O'REILLY®

Wydanie VI

C# 6.0 w pigułce

Joseph Albahari, Ben Albahari

Helion

Tytuł oryginału: C# 6.0 in a Nutshell, 6th Edition

Tłumaczenie: Łukasz Piwko (wstęp, rozdz. 1 – 11, 20 – 24)
Robert Górczyński (12 – 18, 26 – 27)
Jakub Hubisz (rozdz. 19, 25)

ISBN: 978-83-283-2423-7

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/c6pig6
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/c6pig6.zip

Printed in Poland.

Spis treści

7

Kolekcje

Platforma .NET Framework zapewnia standardowy zestaw typów do sortowania i obsługiwania kolekcji obiektów. Wśród nich można znaleźć listy o zmiennym rozmiarze, listy powiązane, sortowane i niesortowane słowniki oraz tablice. Z tych wszystkich typów jedynie tablice należą do języka C#. Pozostałe kolekcje są tylko klasami, których obiekty można tworzyć tak samo jak obiekty wszystkich innych klas.

Typy kolekcji platformy można podzielić na następujące kategorie:

- interfejsy definiujące standardowe protokoły kolekcji;
- gotowe do użycia klasy kolekcji (listy, słowniki itd.);
- klasy bazowe do pisania kolekcji specjalnie dostosowanych do potrzeb konkretnych aplikacji.

W tym rozdziale opisujemy wszystkie te kategorie oraz dodatkowo poświęcamy nieco miejsca typom wykorzystywanym do porównywania i porządkowania elementów.

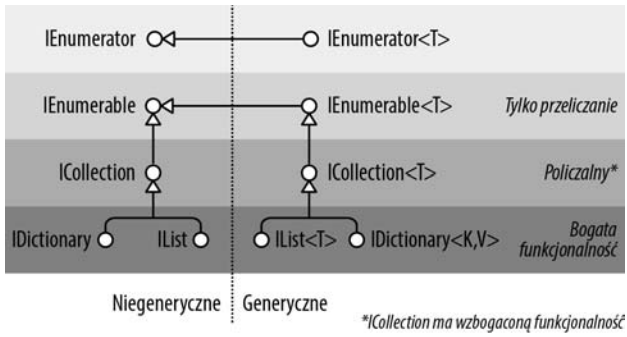
Oto wykaz przestrzeni nazw, w których znajdują się różne kolekcje.

Przestrzeń nazw	Zawartość
`System.Collections`	Niegeneryczne klasy i interfejsy kolekcji
`System.Collections.Specialized`	Silnie typizowane niegeneryczne klasy kolekcji
`System.Collections.Generic`	Generyczne klasy i interfejsy kolekcji
`System.Collections.ObjectModel`	Klasy pośrednie i bazowe do tworzenia niestandardowych kolekcji
`System.Collections.Concurrent`	Kolekcje bezpieczne wątkowo (zob. rozdział 23.)

Przeliczalność

Istnieje wiele różnych rodzajów kolekcji, od prostych struktur danych, przez tablice i listy powiązane po złożone struktury, takie jak drzewa czerwono-czarne i tablice skrótów. Choć konstrukcje te znacznie różnią się pod względem budowy zarówno wewnętrznych, jak i zewnętrznych cech, prawie wszystkie z nich można przeglądać. Platforma .NET zapewnia tę możliwość przez dwa interfejsy

(`IEnumerable` i `IEnumerator` oraz ich generyczne odpowiedniki), dzięki którym różne struktury danych udostępniają jednakowy interfejs API do przeglądania ich zawartości. Wymienione interfejsy należą do szerszego zbioru przedstawionego na rysunku 7.1.



Rysunek 7.1. Interfejsy kolekcji

Interfejsy IEnumerable i IEnumerator

Interfejs `IEnumerator` definiuje podstawowy niskopoziomowy protokół określający sposób przeglądania elementów kolekcji do przodu. Jego deklaracja wygląda tak:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Metoda `MoveNext` przesuwa bieżący element zwany kursorem o jedną pozycję dalej i zwraca fałsz, jeśli był to ostatni element tej kolekcji. Metoda `Current` zwraca bieżący element (zazwyczaj rzutowany z typu `object` na bardziej konkretny typ). Metoda `MoveNext` musi zostać wywołana przed pobraniem pierwszego elementu — zasadę tę wprowadzono, aby było możliwe tworzenie pustych kolekcji. Metoda `Reset`, jeśli jest zaimplementowana, przenosi kursor z powrotem na początek, aby można było od nowa przejrzeć kolekcję. Metoda ta znajduje zastosowanie głównie przy współpracy z technologią COM. Raczej nie wywołuje się jej bezpośrednio, ponieważ nie jest wszędzie obsługiwana (ani niezbędna, gdyż w większości przypadków równie dobrze można utworzyć nowy enumerator).

Tylko nieliczne kolekcje *implementują* enumeratory. Większość *udostępnia* enumeratory przez interfejs `IEnumerable`:

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Dzięki definicji tylko jednej metody zwracającej enumerator interfejs `IEnumerable` zapewnia elastyczność umożliwiającą dostarczenie logiki iteracji w innej klasie. Ponadto dzięki temu kolekcję może przeglądać kilku konsumentów jednocześnie i w żaden sposób nie będą sobie przeszkadzać. Interfejs `IEnumerable` można traktować jak „IEnumeratorProvider” i jest to najbardziej podstawowy interfejs implementowany przez klasy kolekcji.

Poniżej znajduje się przykład niskopoziomowego wykorzystania interfejsów `IEnumerable` i `IEnumerator`:

```
string s = "Cześć";

// klasa string implementuje interfejs IEnumerable, więc możemy wywołać metodę GetEnumerator():
IEnumerator rator = s.GetEnumerator();

while (rator.MoveNext())
{
  char c = (char) rator.Current;
  Console.Write (c + ".");
}

// wynik: C.z.e.ś.ć.
```

Jednak taki bezpośredni sposób wywoływania metod na enumeratorach należy do rzadkości, ponieważ w języku C# istnieje wygodny skrót składniowy w postaci instrukcji `foreach`. Oto ten sam przykład napisany z użyciem tej właśnie instrukcji:

```
string s = "Cześć"; // klasa string implementuje interfejs IEnumerable

foreach (char c in s)
  Console.Write (c + ".");
```

Interfejsy IEnumerable<T> i IEnumerator<T>

Interfejsy `IEnumerator` i `IEnumerable` są prawie zawsze implementowane w parze ze swoimi rozszerzonymi generycznymi wersjami:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
  T Current { get; }
}

public interface IEnumerable<T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
```

Dzięki definicjom typizowanych wersji metod `Current` i `GetEnumerator` interfejsy te wzmacniają bezpieczeństwo typów, eliminują konieczność pakowania elementów typów wartościowych oraz są wygodniejsze w użyciu. Tablice automatycznie implementują interfejs `IEnumerable<T>` (gdzie `T` jest typem elementów przechowywanych w tablicy).

Dzięki zwiększonemu bezpieczeństwu dla typów wywołanie poniższej metody z tablicą znaków spowoduje błąd kompilacji:

```
void Test (IEnumerable<int> numbers) { ... }
```

W klasach kolekcji standardowo udostępnia się publicznie elementy interfejsu `IEnumerable<T>`, a „ukrywa” elementy niegenerycznego interfejsu `IEnumerable` przez zastosowanie jawnej implementacji tego pierwszego. Dzięki temu, jeśli programista bezpośrednio wywoła metodę `GetEnumerator()`, to otrzyma bezpieczny pod względem typów `IEnumerator<T>`. Zdarzają się jednak przypadki złamania tej zasady ze względu na zgodność ze starym kodem (typy generyczne wprowadzono dopiero w C# 2.0). Dobrym przykładem są tablice, które muszą zwracać niegeneryczny (można ładniej powiedzieć: klasyczny) `IEnumerator`, aby nie spowodować awarii starych partii kodu. Aby otrzymać generyczny `IEnumerator<T>`, należy dokonać rzutowania, by udostępnić jawny interfejs:

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable <int>)data).GetEnumerator();
```

Na szczęście dzięki instrukcji `foreach` rzadko jest to konieczne.

Interfejsy IEnumerable<T> i IDisposable

`IEnumerator<T>` dziedziczy po `IDisposable`. Dzięki temu enumeratory mogą przechowywać referencje do takich zasobów jak połączenia z bazą danych i zwalniać je po zakończeniu lub przerwaniu pracy. Instrukcja `foreach` „ma świadomość" tego szczegółu i tłumaczy taki kod:

```
foreach (var element in somethingEnumerable) { ... }
```

na następujący logiczny ekwiwalent:

```
using (var rator = somethingEnumerable.GetEnumerator())
  while (rator.MoveNext())
  {
    var element = rator.Current;
    ...
  }
```

Kiedy używać interfejsów niegenerycznych

Biorąc pod uwagę zwiększone bezpieczeństwo typowe generycznych interfejsów kolekcji, takich jak `IEnumerable<T>`, można zadać pytanie: czy niegeneryczna wersja `IEnumerable` (lub `ICollection` albo `IList`) w ogóle jest do czegoś potrzebna?

Jeśli chodzi o interfejs `IEnumerable`, to musi być implementowany w połączeniu z `IEnumerable<T>`, ponieważ ten drugi korzysta z pierwszego. Jednak niezmiernie rzadko pisze się implementacje tych interfejsów od początku — najczęściej można wykorzystać metody iteracyjne, `Collection<T>` oraz LINQ.

A co z konsumentem? Prawie zawsze wystarczają interfejsy generyczne. Wprawdzie niegeneryczne wersje też są czasami przydatne, choć raczej do zapewnienia spójności typów wszystkich elementów kolekcji. Poniższa metoda np. **rekurencyjnie** liczy elementy w każdej kolekcji:

```
public static int Count (IEnumerable e)
{
  int count = 0;
  foreach (object element in e)
  {
    var subCollection = element as IEnumerable;
    if (subCollection != null)
      count += Count (subCollection);
    else
      count++;
  }
  return count;
}
```

Ponieważ w języku C# występuje kowariancja interfejsów generycznych, można się spodziewać, że ta metoda powinna przyjmować typ `IEnumerable<object>`. To jednak nie udałoby się z elementami typów wartościowych i ze starymi kolekcjami, które nie implementują interfejsu `IEnumerable<T>` — przykładem jest `ControlCollection` z Windows Forms.

Tak na marginesie: uważny czytelnik mógł zauważyć w naszym przykładzie potencjalny błąd — **cykliczne** referencje spowodują nieskończoną rekurencję i awarię metody. Problem ten można łatwo usunąć, używając kolekcji `HashSet` (zob. sekcję „Klasy HashSet<T> i SortedSet<T>").

Blok `using` zapewnia odpowiednie załatwienie zasobów — szerzej na temat interfejsu `IDisposable` piszemy w rozdziale 12.

Implementowanie interfejsów przeliczeniowych

Interfejsy `IEnumerable` i `IEnumerable<T>` można zaimplementować z następujących powodów:

- aby umożliwić korzystanie z instrukcji `foreach`;
- aby zapewnić możliwość współpracy ze wszystkim, co oczekuje standardowej kolekcji;
- aby spełnić wymagania bardziej zaawansowanego interfejsu kolekcji;
- aby obsługiwać inicjalizatory kolekcji.

Aby zaimplementować interfejs `IEnumerable` lub `IEnumerable<T>`, należy dostarczyć enumerator. Można to zrobić na jeden z trzech sposobów:

- jeżeli klasa „opakowuje" inną kolekcję, można zwrócić enumerator tej opakowanej kolekcji;
- przez iterator za pomocą instrukcji `yield return`;
- tworząc własną implementację interfejsu `IEnumerable` lub `IEnumerable<T>`.

Można też utworzyć podklasę istniejącej kolekcji. Klasa `Collection<T>` służy właśnie do tego celu (zob. podrozdział „Kolekcje i pośredniki z możliwością dostosowywania"). Inną możliwością jest użycie operatorów zapytań LINQ, o których mowa w następnym rozdziale.

Zwrócenie enumeratora innej kolekcji jest zaledwie kwestią wywołania metody `GetEnumerator` na wewnętrznej kolekcji. Jednak takie rozwiązanie jest możliwe jedynie w najprostszych przypadkach, gdy elementy wewnętrznej kolekcji są dokładnie tym, czym powinny być. Bardziej elastyczne rozwiązanie polega na napisaniu iteratora przy użyciu instrukcji `yield return`. **Iterator** to element języka C# pomocny w pisaniu kolekcji na podobnej zasadzie, jak instrukcja `foreach` jest pomocna w ich konsumowaniu. Iterator automatycznie rozwiązuje kwestię implementacji interfejsów `IEnumerable` i `IEnumerator` lub ich generycznych wersji. Oto prosty przykład:

```
public class MyCollection : IEnumerable
{
  int[] data = { 1, 2, 3 };

  public IEnumerator GetEnumerator()
  {
    foreach (int i in data)
      yield return i;
  }
}
```

Zwróć uwagę na zawartą w tym kodzie „czarną magię" — metoda `GetEnumerator` nie wygląda tak, jakby miała zwracać enumerator! Kompilator, parsując instrukcję `yield return`, tworzy ukrytą zagnieżdżoną klasę enumeratora, a następnie refaktoryzuje metodę `GetEnumerator` w taki sposób, aby tworzyła i zwracała egzemplarz tej klasy. Iteratory są proste i potężne (i często znajdują zastosowanie w implementacji standardowych operatorów zapytań LINQ to Object).

Zgodnie z tą linią możemy też zaimplementować generyczny interfejs `IEnumerable<T>`:

```
public class MyGenCollection : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };

  public IEnumerator<int> GetEnumerator()
  {
    foreach (int i in data)
      yield return i;
  }

  IEnumerator IEnumerable.GetEnumerator() // jawna implementacja
  { // ukrywa go
    return GetEnumerator();
  }
}
```

Jako że interfejs `IEnumerable<T>` dziedziczy po `IEnumerable`, musimy zaimplementować zarówno generyczną, jak i niegeneryczną wersję metody `GetEnumerator`. Dodatkowo zgodnie ze standardową praktyką zaimplementowaliśmy też jawnie niegeneryczną wersję. Może ona po prostu wywoływać generyczną metodę `GetEnumerator`, ponieważ interfejs `IEnumerator<T>` dziedziczy po `IEnumerator`.

Napisana przez nas klasa mogłaby zostać wykorzystana jako baza do napisania bardziej zaawansowanej kolekcji. Jeśli jednak potrzebna jest tylko prosta implementacja interfejsu `IEnumerable<T>`, instrukcja `yield return` umożliwia skorzystanie z prostszego rozwiązania. Zamiast pisać klasę, logikę iteracji można umieścić w metodzie zwracającej generyczną kolekcję typu `IEnumerable<T>` i resztę zostawić kompilatorowi. Oto przykład:

```
public class Test
{
  public static IEnumerable <int> GetSomeIntegers()
  {
    yield return 1;
    yield return 2;
    yield return 3;
  }
}
```

A oto przykład użycia naszej metody:

```
foreach (int i in Test.GetSomeIntegers())
  Console.WriteLine (i);

// wynik
1
2
3
```

Ostatnim sposobem napisania metody `GetEnumerator` jest napisanie klasy bezpośrednio implementującej interfejs `IEnumerator`. Dokładnie to robi kompilator niejawnie, przy rozpoznawaniu iteratorów. (Na szczęście nieczęsto trzeba posuwać się tak daleko we własnym kodzie). Poniżej znajduje się przykład kolekcji z wpisanymi na stałe wartościami 1, 2 i 3:

```
public class MyIntList : IEnumerable
{
  int[] data = { 1, 2, 3 };
```

```
  public IEnumerator GetEnumerator()
  {
    return new Enumerator (this);
  }

  class Enumerator : IEnumerator // definicja wewnętrznej klasy
  {                              // dla enumeratora
    MyIntList collection;
    int currentIndex = -1;

    public Enumerator (MyIntList collection)
    {
      this.collection = collection;
    }

    public object Current
    {
      get
      {
        if (currentIndex == -1)
          throw new InvalidOperationException ("Enumeracja nie została rozpoczęta!");
        if (currentIndex == collection.data.Length)
          throw new InvalidOperationException ("Za końcem listy!");
        return collection.data [currentIndex];
      }
    }

    public bool MoveNext()
    {
      if (currentIndex >= collection.data.Length - 1) return false;
      return ++currentIndex < collection.data.Length;
    }

    public void Reset() { currentIndex = -1; }
  }
}
```

Implementacja metody `Reset` jest nieobowiązkowa — ewentualnie można zgłaszać wyjątek `NotSupportedException`.

Podkreślmy, że pierwsze wywołanie metody `MoveNext` powinno powodować przejście do pierwszego (a nie drugiego) elementu listy.

Aby uzyskać funkcjonalność zbliżoną do iteratora, musimy jeszcze dodatkowo zaimplementować interfejs `IEnumerator<T>`. Poniżej przedstawiamy przykład z pominięciem testów granic dla uproszczenia:

```
class MyIntList : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };

  // Generyczny enumerator jest zgodny zarówno z IEnumerable, jak i IEnumerable<T>.
  // Implementujemy niegeneryczną metodę GetEnumerator jawnie, aby uniknąć konfliktów nazw.

  public IEnumerator<int> GetEnumerator() { return new Enumerator(this); }
  IEnumerator IEnumerable.GetEnumerator() { return new Enumerator(this); }
```

```
    class Enumerator : IEnumerator<int>
    {
      int currentIndex = -1;
      MyIntList collection;

      public Enumerator (MyIntList collection)
      {
        this.collection = collection;
      }

      public int Current => collection.data [currentIndex];

      object IEnumerator.Current => Current;

      public bool MoveNext() => ++currentIndex < collection.data.Length;

      public void Reset() => currentIndex = -1;
      // Jeśli nie jest potrzebna metoda Dispose, to dobrym zwyczajem jest jej
      // jawne zaimplementowanie tak, aby była niedostępna w interfejsie publicznym.
      void IDisposable.Dispose() {}
    }
  }
```

Przykład z użyciem typów generycznych jest szybszy, ponieważ metoda `IEnumerator<int>.Current` nie wymaga rzutowania z `int` na `object`, a więc eliminuje pakowanie.

Interfejsy ICollection i IList

Choć interfejsy enumeracyjne stanowią protokół iteracji kolekcji tylko do przodu, nie zapewniają możliwości sprawdzania rozmiaru kolekcji, dostępu do składowych za pomocą indeksów, przeszukiwania struktur danych ani ich modyfikowania. Wszystkie te funkcje zapewniają interfejsy .NET Framework `ICollection`, `IList` oraz `IDictionary`. Każdy z nich występuje w wersji generycznej i niegenerycznej, choć te drugie istnieją głównie ze względu na zgodność ze starym kodem.

Hierarchia dziedziczenia tych interfejsów jest pokazana na rysunku 7.1. Najprościej można je podsumować w następujący sposób:

`IEnumerable<T>` (i `IEnumerable`)
: Zapewnia minimalną funkcjonalność (tylko przeglądanie).

`ICollection<T>` (i `ICollection`)
: Zapewnia średni poziom funkcjonalności (np. własność `Count`).

`IList <T>/IDictionary <K,V>` i ich niegeneryczne wersje
: Zapewnia najwyższy poziom funkcjonalności (wliczając dostęp „swobodny” przy użyciu indeksów i kluczy).

Konieczność *implementowania* któregokolwiek z tych interfejsów jest rzadkością. Gdy trzeba napisać klasę kolekcji, to prawie zawsze można wykorzystać do tego klasę bazową `Collection<T>` (zob. podrozdział „Kolekcje i pośredniki z możliwością dostosowywania”). W niektórych przypadkach inną możliwość zapewnia też technologia LINQ.

Różnice między wersjami generycznymi i niegenerycznymi są większe niż można się spodziewać, zwłaszcza w przypadku interfejsu `ICollection`. Przyczyny tego są w głównej mierze historyczne — typy generyczne powstały później, więc interfejsy generyczne tworzono z pewnym bagażem doświadczenia, dzięki czemu udało się dobrać inne (i lepsze) składowe. Dlatego właśnie interfejs `ICollection<T>` nie rozszerza interfejsu `ICollection`, `IList<T>` nie rozszerza interfejsu `IList`, a `IDictionary<TKey, TValue>` nie rozszerza interfejsu `IDictionary`. Oczywiście klasa kolekcji może implementować obie wersje interfejsu, jeśli jest to korzystne (a często jest).

Innym, mniej oczywistym powodem, dla którego `IList<T>` nie rozszerza interfejsu `IList`, jest to, że rzutowanie na `IList<T>` zwracałoby interfejs ze składowymi `Add(T)` i `Add(object)`. To z kolei oznaczałoby zniweczenie bezpieczeństwa typowego, ponieważ można byłoby wywołać metodę `Add` z obiektem dowolnego typu.

W tym podrozdziale są opisane interfejsy `ICollection<T>`, `IList<T>` i ich niegeneryczne wersje. Opis interfejsów słownikowych znajduje się w podrozdziale „Słowniki".

W obrębie platformy .NET Framework słowa **kolekcja** i **lista** są używane bez *dającej się uchwycić* logiki. Na przykład interfejs `IList<T>` jest bardziej funkcjonalną wersją interfejsu `ICollection<T>`, więc można oczekiwać, że klasa `List<T>` będzie tak samo bardziej funkcjonalna niż klasa `Collection<T>`. Jednak tak nie jest. Dlatego terminy **kolekcja** i **lista** najlepiej traktować jako synonimy, chyba że chodzi o konkretny typ.

Interfejsy ICollection<T> i ICollection

`ICollection<T>` to standardowy interfejs dla policzalnych kolekcji obiektów. Zapewnia możliwość sprawdzenia rozmiaru kolekcji (`Count`), sprawdzenia, czy dany element znajduje się w kolekcji (`Contains`), skopiowania kolekcji do tablicy (`ToArray`) oraz stwierdzenia, czy strukturę można modyfikować (`IsReadOnly`). W przypadku kolekcji z możliwością zapisu dostępne są także operacje `Add`, `Remove` i `Clear`. A ponieważ interfejs ten rozszerza `IEnumerable<T>`, kolekcje można przeglądać za pomocą instrukcji `foreach`:

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }

  bool Contains (T item);
  void CopyTo (T[] array, int arrayIndex);
  bool IsReadOnly { get; }

  void Add(T item);
  bool Remove (T item);
  void Clear();
}
```

Niegeneryczny interfejs `ICollection` także opisuje policzalną kolekcję, ale nie zapewnia funkcji do modyfikowania listy ani sprawdzania, czy zawiera określony element:

```
public interface ICollection : IEnumerable
{
  int Count { get; }
  bool IsSynchronized { get; }
```

```
  object SyncRoot { get; }
  void CopyTo (Array array, int index);
}
```

Niegeneryczny interfejs dodatkowo definiuje własności przydatne w synchronizacji (rozdział 14.) — wstawiono je do wersji generycznej, ponieważ bezpieczeństwo wątków nie jest już uważane za cechę wewnętrzną kolekcji.

Implementacja obu interfejsów jest łatwa. W przypadku implementacji interfejsu tylko do odczytu `ICollection<T>` metody `Add`, `Remove` i `Clear` powinny zgłaszać wyjątek `NotSupportedException`.

Z reguły interfejsy te implementuje się łącznie z `IList` lub `IDictionary`.

Interfejsy IList<T> i IList

`IList<T>` to standardowy interfejs kolekcji indeksowanych pozycyjnie. Oprócz składników odziedziczonych z interfejsów `ICollection<T>` i `IEnumerable<T>` zawiera funkcje pozwalające odczytywać, zapisywać (za pomocą indeksatora), wstawiać oraz usuwać elementy wg pozycji:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  T this [int index] { get; set; }
  int IndexOf (T item);
  void Insert (int index, T item);
  void RemoveAt (int index);
}
```

Metody `IndexOf` wykonują przeszukiwanie liniowe listy i zwracają wartość `-1`, jeśli nie znajdą szukanego elementu.

Niegeneryczna wersja interfejsu `IList` ma więcej składowych, ponieważ mniej dziedziczy po `ICollection`:

```
public interface IList : ICollection, IEnumerable
{
  object this [int index] { get; set }
  bool IsFixedSize { get; }
  bool IsReadOnly { get; }
  int Add (object value);
  void Clear();
  bool Contains (object value);
  int IndexOf (object value);
  void Insert (int index, object value);
  void Remove (object value);
  void RemoveAt (int index);
}
```

Metoda `Add` niegenerycznego interfejsu `IList` zwraca liczbę całkowitą będącą indeksem nowo dodanego elementu. Dla porównania metoda `Add` interfejsu `ICollection<T>` ma typ zwrotny `void`.

Ogólna klasa `List<T>` jest typową implementacją interfejsów `IList<T>` i `IList`. W języku C# także tablice implementują zarówno generyczną, jak i niegeneryczną wersję (choć metody dodające i usuwające elementy są ukryte przez jawną implementację interfejsu i w razie wywołania zgłaszają wyjątek `NotSupportedException`).

Przy próbie uzyskania dostępu do wielowymiarowej tablicy za pomocą indeksatora interfejsu `IList` zgłaszany jest wyjątek `ArgumentException`. Jest to pułapka dla programistów piszących takie metody jak poniższa:

```
public object FirstOrNull (IList list)
{
  if (list == null || list.Count == 0) return null;
  return list[0];
}
```

Na pierwszy rzut oka może się wydawać, że to bardzo solidna funkcja, ale zgłosi wyjątek, jeśli ktoś wywoła ją na tablicy wielowymiarowej. W razie potrzeby w czasie działania programu można testować, czy argument nie jest tablicą wielowymiarową, za pomocą poniższego wyrażenia (szerzej na ten temat piszemy w rozdziale 19.):

```
list.GetType().IsArray && list.GetType().GetArrayRank()>1
```

Interfejs IReadOnlyList<T>

W celu zapewniania możliwości współpracy z kolekcjami tylko do odczytu Windows Runtime w .NET Framework 4.5 wprowadzono nowy interfejs kolekcji o nazwie `IReadOnlyList<T>`. Jest on jednak przydatny sam w sobie i można go traktować jak okrojoną wersję interfejsu `IList<T>` udostępniającego tylko składowe potrzebne do wykonywania operacji odczytu na listach:

```
public interface IReadOnlyList<out T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }
  T this[int index] { get; }
}
```

Ponieważ parametr typu jest wykorzystywany wyłącznie na pozycjach wyjściowych, oznaczono go jako kowariantny. Dzięki temu np. listę kotów można traktować jako listę zwierząt tylko do odczytu. Natomiast w interfejsie `IList<T>` parametr `T` nie jest oznaczony jako kowariantny, ponieważ znajduje zastosowanie zarówno na pozycjach wejściowych, jak i wyjściowych.

Interfejs `IReadOnlyList<T>` reprezentuje **widok** listy tylko do odczytu. Nie oznacza to jednak, że podstawowa implementacja także musi być tylko do odczytu.

Zgodnie z logiką interfejs `IList<T>` powinien dziedziczyć po `IReadOnlyList<T>`. Jednak firma Microsoft nie mogła wprowadzić takiej zmiany, ponieważ wymagałaby ona przeniesienia składowych z `IList<T>` do `IReadOnlyList<T>`, co spowodowałoby problemy z programami działającymi pod kontrolą CLR 4.5 (konsumenci musieliby ponownie skompilować swoje programy, aby uniknąć błędów wykonawczych). Dlatego implementatorzy interfejsu `IList<T>` muszą ręcznie dodawać do swoich kolekcji implementację interfejsu `IReadOnlyList<T>`.

Interfejs `IReadOnlyList<T>` odpowiada typowi Windows Runtime `IVectorView<T>`.

Klasa Array

Klasa `Array` to podstawa wszystkich jedno- i wielowymiarowych tablic oraz jeden z podstawowych typów implementujących standardowe interfejsy kolekcji. Ponieważ zapewnia unifikację typów, wszystkie tablice dysponują takim samym zestawem metod, niezależnie od ich deklaracji i typu elementów.

Ze względu na wielkie znaczenie tablic w języku C# utworzono specjalną składnię do ich deklarowania i inicjalizowania, której opis znajduje się w rozdziałach 2. i 3. Gdy programista deklaruje tablicę za pomocą składni C#, system CLR niejawnie generuje podtyp klasy `Array`, tworząc **pseudotyp** o odpowiednich wymiarach i typie elementów. Ten typ implementuje typizowane generyczne interfejsy kolekcji, np. `IList<string>`.

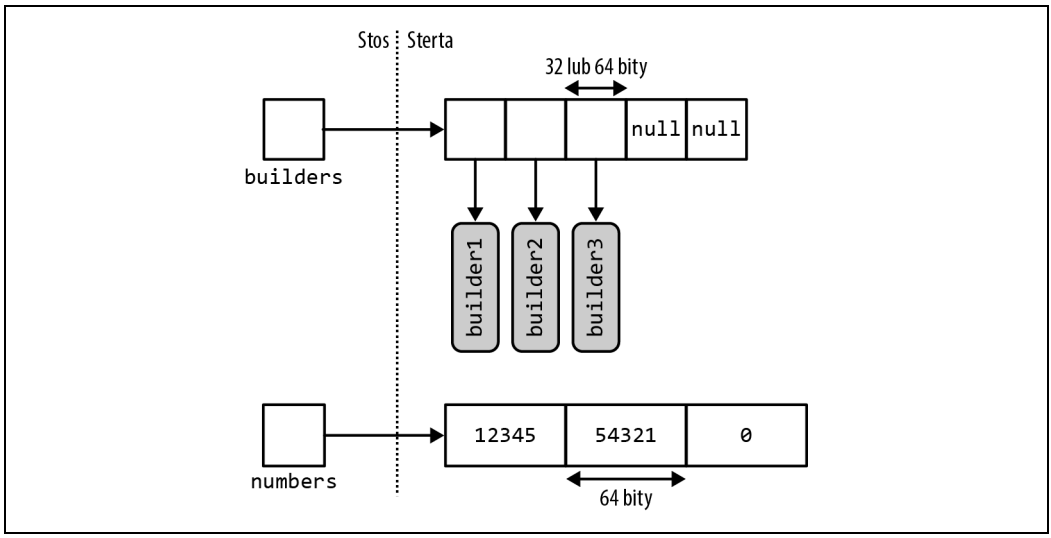
Ponadto system CLR traktuje typy tablicowe w specjalny sposób, przypisując tworzonym obiektom ciągły obszar pamięci. Dzięki temu indeksowanie jest bardzo szybkie, ale za cenę braku możliwości zwiększenia rozmiaru struktury w późniejszym czasie.

Klasa `Array` implementuje interfejsy kolekcji do `IList<T>` w wersjach generycznych i niegenerycznych. Natomiast interfejs `IList<T>` jest zaimplementowany jawnie w celu utrzymania porządku w interfejsie publicznym tablicy, który nie może udostępniać takich metod jak `Add` czy `Remove`, ponieważ zgłaszają one wyjątki, jeśli wywoła się je na kolekcji o niezmiennym rozmiarze. Wprawdzie klasa `Array` udostępnia statyczną metodę o nazwie `Resize` do zmieniania rozmiaru tablicy, ale jej działanie polega na utworzeniu nowej tablicy i skopiowaniu do niej wszystkich elementów. Nie dość, że ta operacja jest bardziej nieefektywna, to na dodatek wszystkie znajdujące się w różnych miejscach programu referencje nadal będą wskazywać starą strukturę. Jeśli potrzebny jest obiekt o zmiennym rozmiarze, to lepiej użyć klasy `List<T>` (opisanej w następnym podrozdziale).

W tablicy można przechowywać elementy typów zarówno wartościowych, jak i referencyjnych. Elementy wartościowe są przechowywane bezpośrednio w tablicy, więc struktura zawierająca trzy liczby całkowite (każda po 8 bajtów) zajmuje ciągły obszar pamięci o rozmiarze 24 bajtów. Natomiast elementy typów referencyjnych zajmują w tablicy tylko tyle miejsca, ile potrzeba do przechowywania referencji (4 bajty w środowisku 32-bitowym i 8 bajtów w środowisku 64-bitowym). Na rysunku 7.2 pokazano, co dzieje się w pamięci po wykonaniu poniższego programu:

```
StringBuilder[] builders = new StringBuilder [5];
builders [0] = new StringBuilder ("builder1");
builders [1] = new StringBuilder ("builder2");
builders [2] = new StringBuilder ("builder3");

long[] numbers = new long [3];
numbers [0] = 12345;
numbers [1] = 54321;
```

Jako że `Array` to klasa, same tablice są typami referencyjnymi, niezależnie od rodzaju przechowywanych w nich elementów. Oznacza to, że wynikiem wykonania instrukcji `tablicaB = tablicaA` będzie powstanie dwóch zmiennych odnoszących się do tej samej tablicy. Jednocześnie test równości dwóch różnych tablic zawsze będzie miał wynik negatywny, chyba że programista użyje własnego



Rysunek 7.2. Tablice w pamięci

komparatora. Jeden taki komparator wprowadzono w .NET Framework 4.0, aby umożliwić porównywanie elementów w tablicach lub krotkach. Dostęp do niego można uzyskać przez typ `StructuralComparisons`:

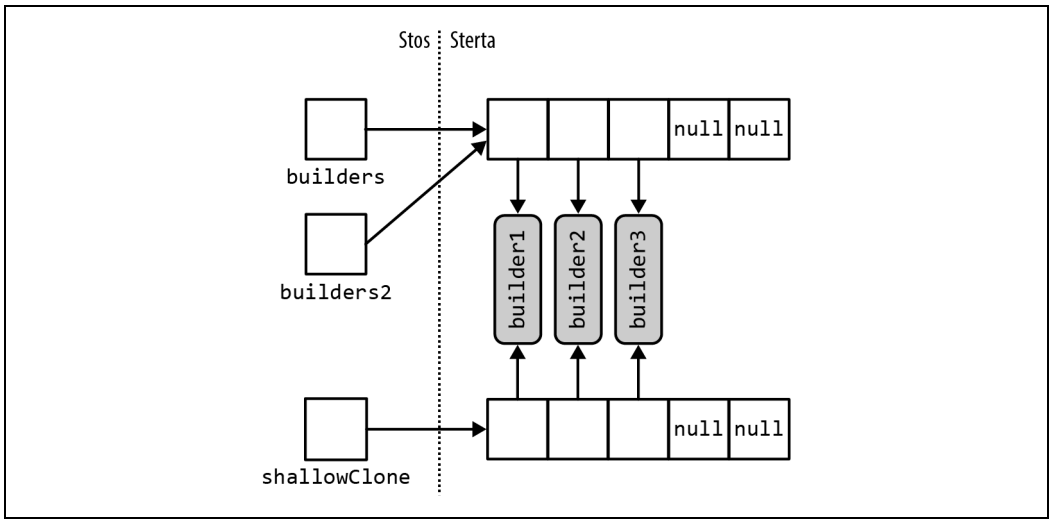
```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };

Console.WriteLine (a1 == a2); // fałsz
Console.WriteLine (a1.Equals (a2)); // fałsz

IStructuralEquatable se1 = a1;
Console.WriteLine (se1.Equals (a2,
  StructuralComparisons.StructuralEqualityComparer)); // prawda
```

Tablice można kopiować za pomocą metody `Clone`: `tablicaB = tablicaA.Clone()`. Jednak w ten sposób można wykonać kopię płytką, czyli obejmującą tylko reprezentowany przez tablicę obszar pamięci. Jeżeli struktura zawiera elementy typów wartościowych, to zostaną one skopiowane. Ale jeżeli tablica zawiera obiekty typów referencyjnych, to skopiowane zostaną tylko referencje (w efekcie powstaną dwie tablice, których składowe odnoszą się do tych samych obiektów). Na rysunku 7.3 przedstawiono skutek dodania do naszego przykładu poniższego kodu:

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone();
```

Jeśli potrzebna jest kopia głęboka — z duplikatami wszystkich obiektów typu referencyjnego — należy przejrzeć tablicę za pomocą pętli i każdy element sklonować ręcznie. Te same zasady dotyczą także innych typów kolekcji .NET.

Choć klasa `Array` jest głównie przeznaczona do pracy z indeksatorami 32-bitowymi, to dzięki kilku metodom przyjmującym parametry `Int32` i `Int64` do pewnego stopnia obsługuje także indeksatory 64-bitowe (co teoretycznie zwiększa jej przestrzeń adresową do 2^{64} elementów). W praktyce te przeciążone wersje metod są jednak bezużyteczne, ponieważ maksymalny rozmiar obiektu w systemie CLR — w tym także tablic — wynosi 2 GB (zarówno w środowisku 32-bitowym, jak i 64-bitowym).



Rysunek 7.3. Płytkie kopiowanie tablicy

Wiele z metod klasy `Array`, które mogą się wydawać metodami egzemplarzowymi, w istocie jest metodami statycznymi. Projektanci tej klasy podjęli dziwną decyzję pod tym względem i przez to programista szukający odpowiedniej metody musi sprawdzać zarówno wśród metod statycznych, jak i egzemplarzowych.

Tworzenie i indeksowanie tablic

Najprostszym sposobem na utworzenie i zindeksowanie tablicy jest użycie konstrukcji języka C#:

```
int[] myArray = { 1, 2, 3 };
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

Ewentualnie tablicę można utworzyć dynamicznie za pomocą metody `Array.CreateInstance`, która pozwala na określenie typu elementów i liczby wymiarów oraz tworzenie tablic indeksowanych od innego numeru niż zero (przez określenie dolnej granicy). Tablice indeksowane w ten sposób są jednak niezgodne z CLS (ang. *Common Language Specification*).

Dostęp do elementów dynamicznie utworzonej tablicy zapewniają metody `GetValue` i `SetValue` (działają też na zwykłych tablicach):

```
// utworzenie tablicy łańcuchów o długości 2
Array a = Array.CreateInstance (typeof(string), 2);
a.SetValue ("Witaj,", 0);                          // → a[0] = "Witaj,";
a.SetValue ("świecie", 1);                         // → a[1] = "świecie";
string s = (string) a.GetValue (0);                // → s = a[0];
// istnieje też możliwość rzutowania na tablice C#
string[] cSharpArray = (string[]) a;
string s2 = cSharpArray [0];
```

Tworzone dynamicznie tablice indeksowane od zera można rzutować na tablice C# takiego samego lub zgodnego typu (zgodnego wg zasad wariancji tablic). Jeśli np. klasa `Apple` jest pochodną klasy `Fruit`, to tablicę typu `Apple[]` można rzutować na typ `Fruit[]`. W tym momencie niektórzy mogą

się zastanawiać, dlaczego do unifikacji tablicy nie użyto typu `object[]`, tylko klasy `Array`. Przyczyną jest to, że typ `object[]` jest niezgodny z wielowymiarowymi tablicami typów wartościowych (a także indeksowanych nie od zera). Tablicy typu `int[]` nie można rzutować na `object[]`. Dlatego potrzebujemy klasy `Array` w celu zapewnienia kompletnej unifikacji.

Metody `GetValue` i `SetValue` działają też na tablicach stworzonych przez kompilator oraz są przydatne przy pisaniu metod działających na tablicach dowolnego typu i rzędu. Dla tablic wielowymiarowych przyjmują *tablicę* indeksatorów:

```
public object GetValue (params int[] indices)
public void SetValue (object value, params int[] indices)
```

Poniższa metoda drukuje pierwszy element tablicy, niezależnie od tego, ile ma ona wymiarów:

```
void WriteFirstValue (Array a)
{
  Console.Write (a.Rank + "-dimensional; ");

  // Tablica indeksatorów automatycznie zainicjalizuje wszystkie wartości zerami, więc przekazanie
  // jej do metody GetValue lub SetValue spowoduje pobranie lub ustawienie pierwszego elementu tablicy.

  int[] indexers = new int[a.Rank];
  Console.WriteLine ("Pierwsza wartość to " + a.GetValue (indexers));
}

void Demo()
{
  int[] oneD = { 1, 2, 3 };
  int[,] twoD = { {5,6}, {8,9} };

  WriteFirstValue (oneD); // jednowymiarowa; pierwsza wartość to 1
  WriteFirstValue (twoD); // dwuwymiarowa; pierwsza wartość to 5
}
```

Do pracy z tablicami nieznanego typu, ale znanego rzędu można zastosować prostsze i efektywniejsze rozwiązanie z wykorzystaniem typów generycznych:

```
void WriteFirstValue<T> (T[] array)
{
  Console.WriteLine (array[0]);
}
```

Metoda `SetValue` zgłasza wyjątek, jeżeli element jest nieodpowiedniego dla danej tablicy typu.

Gdy tworzona jest tablica, zarówno za pomocą składni języka, jak i metody `Array.CreateInstance`, jej elementy zostają automatycznie zainicjalizowane. Jeżeli elementy są typu referencyjnego, początkowo przypisywana jest im wartość `null`. Natomiast w przypadku elementów typów wartościowych zostaje wywołany konstruktor domyślny odpowiedniego typu (co w istocie oznacza wyzerowanie składowych). Ponadto klasa `Array` zawiera metodę `Clear`, za pomocą której w dowolnym momencie można wyzerować tablicę:

```
public static void Clear (Array array, int index, int length);
```

Metoda ta nie zmienia rozmiaru tablicy, co kłóci się z jej typowym zastosowaniem (np. w `ICollec` ↪`tion<T>.Clear`), polegającym na redukcji struktury do zera elementów.

Przeglądanie zawartości tablicy

Zawartość tablicy można łatwo przejrzeć za pomocą instrukcji `foreach`:

```
int[] myArray = { 1, 2, 3};
foreach (int val in myArray)
Console.WriteLine (val);
```

Ewentualnie można też użyć statycznej metody `Array.ForEach` o następującej definicji:

```
public static void ForEach<T> (T[] array, Action<T> action);
```

Wykorzystuje ona delegat `Action` o następującej sygnaturze:

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

Tak wygląda pierwszy przykład napisany z użyciem metody `Array.ForEach`:

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

Długość i liczba wymiarów tablicy

W klasie `Array` dostępne są następujące metody i własności do sprawdzania długości i liczby wymiarów tablicy:

```
public int GetLength (int dimension);
public long GetLongLength (int dimension);

public int Length { get; }
public long LongLength { get; }

public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);

public int Rank { get; } // zwraca liczbę wymiarów tablicy
```

Metody `GetLength` i `GetLongLength` zwracają długość wybranego wymiaru (0, jeśli tablica jest jednowymiarowa), natomiast metody `Length` i `LongLength` zwracają liczbę wszystkich elementów tablicy z wszystkich wymiarów łącznie.

Metody `GetLowerBound` i `GetUpperBound` służą do pracy z tablicami indeksowanymi od innego numeru niż zero. Metoda `GetUpperBound` zwraca wynik równy sumie wartości `GetLowerBound` i `GetLength` dla wybranego wymiaru.

Przeszukiwanie tablic

Klasa `Array` udostępnia bogaty wybór metod do znajdowania elementów w tablicach jednowymiarowych:

Metody `BinarySearch`
: Do szybkiego szukania elementu w posortowanej tablicy.

Metody `IndexOf/LastIndex`
: Do szukania elementu w nieposortowanej tablicy.

`Find`, `FindLast`, `FindIndex`, `FindLastIndex`, `FindAll`, `Exists`, `TrueForAll`
: Do szukania w nieposortowanych tablicach elementów spełniających warunek `Predicate<T>`.

Żadna z metod przeszukujących nie zgłasza wyjątku, jeśli nie znajdzie szukanej wartości. Zamiast tego metody zwracające liczbę całkowitą zwracają `-1` (dla tablic indeksowanych od zera), a metody zwracające typ generyczny zwracają domyślną wartość tego typu (np. `0` w przypadku `int` i `null` w przypadku `string`).

Metody przeszukiwania binarnego są szybkie, ale działają tylko na posortowanych tablicach i wymagają porównywania elementów pod względem *kolejności*, a nie *równości*. W efekcie metody te przyjmują obiekty typu `IComparer` i `IComparer<T>` określające definicję porządku w danym przypadku (zob. sekcję „Dołączanie protokołów równości i porządkowania"). Definicja ta musi być zgodna z komparatorem użytym pierwotnie do posortowania tablicy. Jeżeli nie zostanie przekazany żaden komparator, zostanie zastosowany domyślny algorytm porządkujący dla danego typu oparty na jego implementacji interfejsu `IComparable/IComparable<T>`.

Metody `IndexOf` i `LastIndexOf` służą do prostego przeglądania elementów tablicy i zwracają numer pierwszej (lub ostatniej) pozycji podanej wartości.

Metody działające na bazie predykatów przyjmują delegaty i wyrażenia lambda pozwalające stwierdzić, czy dana wartość „pasuje" do szukanej. Predykat to po prostu delegat przyjmujący obiekt i zwracający prawdę lub fałsz:

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

W poniższym przykładzie szukamy w tablicy łańcuchów imienia zawierającego literę `a`:

```
static void Main()
{
  string[] names = { "Robert", "Jacek", "Juliusz" };
  string match = Array.Find (names, ContainsA);
  Console.WriteLine (match); // Jacek
}
static bool ContainsA (string name) { return name.Contains ("a"); }
```

To jest ten sam kod, tylko skrócony dzięki użyciu metody anonimowej:

```
string[] names = { "Robert", "Jacek", "Juliusz" };
string match = Array.Find (names, delegate (string name)
  { return name.Contains ("a"); } );
```

Wyrażenie lambda pozwala jeszcze bardziej skrócić kod:

```
string[] names = { "Robert", "Jacek", "Juliusz" };
string match = Array.Find (names, n => n.Contains ("a")); // Jacek
```

Metoda `FindAll` zwraca tablicę wszystkich elementów spełniających warunek predykatu. Metoda ta, równoznaczna z `Enumerable.Where` z przestrzeni nazw `System.Linq`, zwraca tylko tablicę pasujących elementów, a nie obiekt typu `IEnumerable<T>`.

Metoda `Exists` zwraca prawdę, jeżeli którykolwiek element tablicy spełnia warunki danego predykatu, i jest równoważna metodzie `Any` z `System.Linq.Enumerable`.

Metoda `TrueForAll` zwraca prawdę, jeżeli wszystkie elementy spełniają warunek predykatu, i jest równoważna z `All` z `System.Linq.Enumerable`.

Sortowanie

Klasa `Array` ma następujące wbudowane metody sortujące:

```
// do sortowania pojedynczej tablicy

public static void Sort<T> (T[] array);
public static void Sort (Array array);

// do sortowania pary tablic

public static void Sort<TKey,TValue> (TKey[] keys, TValue[] items);
public static void Sort (Array keys, Array items);
```

Każda z tych metod jest dodatkowo przeciążona i przyjmuje:

```
int index // indeks, od którego ma być rozpoczęte sortowanie
int length // liczba elementów do posortowania
IComparer<T> comparer // obiekt definiujący sposób porównywania
Comparison<T> comparison // delegat definiujący sposób porównywania
```

Poniżej znajduje się najprostszy przykład użycia metody `Sort`:

```
int[] numbers = { 3, 2, 1 };
Array.Sort (numbers); // teraz tablica ma wartości { 1, 2, 3 }
```

Metody przyjmujące pary tablic przekładają elementy w każdej z tablic łącznie i opierają decyzje porządkowe na pierwszej tablicy. W następnym przykładzie zarówno liczby, jak i odpowiadające im słowa są sortowane w porządku numerycznym:

```
int[] numbers = { 3, 2, 1 };
string[] words = { "trzy", "dwa", "jeden" };
Array.Sort (numbers, words);

// teraz kolejność elementów w tablicy numbers to { 1, 2, 3 }
// teraz kolejność elementów w tablicy words to { "jeden", "dwa", "trzy" }
```

Metoda `Array.Sort` wymaga, aby znajdujące się w tablicy elementy implementowały interfejs `IComparable` (zob. podrozdział „Określanie kolejności” w rozdziale 6.). Oznacza to, że większość wbudowanych typów języka C# (takich jak użyte w poprzednim przykładzie liczby całkowite) można sortować. Jeżeli elementy nie mają określonego wewnętrznego porządku albo programista chce zmienić domyślną kolejność, konieczne jest przekazanie metodzie `Sort` własnego dostawcy `comparison` określającego względne ustawienie dwóch elementów. Można to zrobić na dwa sposoby:

- przy użyciu pomocniczego obiektu implementującego interfejs `IComparer`/`Icomparer<T>` (zob. podrozdział „Dołączanie protokołów równości i porządkowania”);
- przy użyciu delegatu `Comparison`:

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

Delegat `Comparison` posługuje się taką samą semantyką jak metoda `IComparer<T>.CompareTo`: jeśli `x` jest przed `y`, zostaje zwrócona ujemna liczba całkowita; jeśli `x` jest za `y`, zostaje zwrócona dodatnia liczba całkowita; jeśli `x` i `y` zajmują to samo miejsce, zostaje zwrócone `0`.

W poniższym przykładzie sortujemy tablicę liczb całkowitych w taki sposób, że najpierw ustawiane są liczby nieparzyste:

```
int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort (numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);
// zawartość tablicy numbers to { 1, 3, 5, 2, 4 }
```

Zamiast metody Sort można też użyć operatorów LINQ OrderBy i ThenBy. W odróżnieniu od metody Array.Sort operatory LINQ nie zmieniają pierwotnej tablicy, tylko wysyłają posortowane wyniki do nowej sekwencji IEnumerable<T>.

Odwracanie kolejności elementów

Poniższe metody klasy Array odwracają kolejność wszystkich lub niektórych elementów tablicy:

```
public static void Reverse (Array array);
public static void Reverse (Array array, int index, int length);
```

Kopiowanie

Klasa Array zawiera cztery metody do kopiowania płytkiego: Clone, CopyTo, Copy oraz ConstrainedCopy. Dwie pierwsze są metodami egzemplarzowymi, a pozostałe — statycznymi.

Metoda Clone zwraca nową tablicę (stanowiącą płytką kopię oryginału). Metody CopyTo i Copy kopiują ciągły podzbiór elementów tablicy. Kopiowanie wielowymiarowej prostokątnej tablicy wymaga przeprowadzenia mapowania wielowymiarowych indeksów na liniowe. Na przykład środkowy element (position[1,1]) w tablicy o wymiarach 3×3 ma indeks 4, który wynika z obliczeń 1*3+1. Zakresy źródłowy i docelowy mogą się bez problemu nakładać.

Metoda ConstrainedCopy wykonuje operację **atomową**, tzn. jeśli nie można skopiować wszystkich potrzebnych elementów (np. z powodu błędnego typu), to cała operacja jest wycofywana.

Ponadto klasa Array zawiera metodę AsReadOnly, która zwraca opakowanie uniemożliwiające ponowne przypisywanie wartości elementom.

Konwertowanie i zmienianie rozmiarów tablic

Metoda Array.ConvertAll tworzy i zwraca nową tablicę typu TOutput. Wywołuje w tym celu przekazany jej delegat Converter. Jej definicja wygląda następująco:

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

Poniżej znajduje się przykład konwersji tablicy liczb zmiennoprzecinkowych na tablicę liczb całkowitych:

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll (reals, r => Convert.ToInt32 (r));

// zawartość tablicy wholes to { 1, 2, 2 }
```

Metoda Resize tworzy nową tablicę i kopiuje do niej wszystkie elementy, a następnie zwraca tę nową strukturę przez parametr referencyjny. Mimo to referencje do pierwotnej tablicy przechowywane w innych obiektach pozostają niezmienione.

Przestrzeń `System.Linq` zawiera dodatkowy asortyment metod do konwersji tablic. Wszystkie one zwracają obiekty typu `IEnumerable<T>`, który można przekonwertować na tablicę za pomocą metody `ToArray` z klasy `Enumerable`.

Listy, kolejki, stosy i zbiory

Platforma zapewnia podstawowy zestaw konkretnych klas kolekcji implementujących opisane wcześniej interfejsy. W tym podrozdziale skupiamy się na kolekcjach **listowych** (których nie należy mylić ze strukturami słownikowymi opisanymi w podrozdziale „Słowniki"). Tak jak w przypadku opisanych wcześniej interfejsów, często mamy do wyboru zarówno generyczne, jak i niegeneryczne wersje każdego typu. Pod względem elastyczności i wydajności lepsze są klasy generyczne, przez co ich niegeneryczne odpowiedniki są w zasadzie niepotrzebne z wyjątkiem sytuacji, kiedy trzeba zapewnić zgodność ze starym kodem. Jest więc inaczej niż w przypadku interfejsów kolekcji, które bywają przydatne także w wersjach niegenerycznych.

Z opisanych w tym podrozdziale klas najczęściej używana jest generyczna klasa `List`.

Klasy List<T> i ArrayList

Generyczna klasa `List` i niegeneryczna klasa `ArrayList` umożliwiają tworzenie tablic obiektów o dynamicznym rozmiarze i należą do najczęściej używanych klas kolekcji. Klasa `ArrayList` implementuje interfejs `IList`, podczas gdy `List<T>` implementuje zarówno `IList`, jak i `IList<T>` (jak również nową wersję tylko do odczytu o nazwie `IReadOnlyList<T>`). Inaczej niż w przypadku tablic, wszystkie interfejsy są zaimplementowane publicznie i zgodnie z oczekiwaniami metody takie jak `Add` czy `Remove` są dostępne.

Wewnętrznie klasy `List<T>` i `ArrayList` utrzymują tablicę obiektów, którą w razie osiągnięcia maksimum pojemności zamieniają na większą. Dodawanie elementów na końcu to efektywna operacja (ponieważ zazwyczaj na końcu jest wolne miejsce), natomiast wstawianie elementów w środku pomiędzy innymi elementami może być powolne (ponieważ trzeba przesunąć wszystkie elementy znajdujące się za miejscem wstawiania). Jeśli chodzi o przeszukiwanie, to podobnie jak w przypadku tablic operacja jest efektywna, jeżeli przeprowadza się ją za pomocą metody `BinarySearch` na posortowanej liście. W innych sytuacjach wyszukiwanie jest mało wydajne, gdyż wymaga sprawdzenia każdego elementu po kolei.

Klasa `List<T>` jest do kilku razy szybsza od `ArrayList`, gdy `T` jest typem wartościowym, ponieważ `List<T>` nie wymaga pakowania ani odpakowywania elementów.

Klasy `List<T>` i `ArrayList` zawierają konstruktory przyjmujące istniejące kolekcje elementów — kopiują one wszystkie elementy z przekazanej kolekcji do nowej struktury:

```
public class List<T> : IList<T>, IReadOnlyList<T>
{
  public List ();
  public List (IEnumerable<T> collection);
  public List (int capacity);
```

```
    // Add+Insert
    public void Add (T item);
    public void AddRange (IEnumerable<T> collection);
    public void Insert (int index, T item);
    public void InsertRange (int index, IEnumerable<T> collection);

    // Remove
    public bool Remove (T item);
    public void RemoveAt (int index);
    public void RemoveRange (int index, int count);
    public int RemoveAll (Predicate<T> match);

    // indeksowanie
    public T this [int index] { get; set; }
    public List<T> GetRange (int index, int count);
    public Enumerator<T> GetEnumerator();

    // eksportowanie, kopiowanie i konwertowanie
    public T[] ToArray();
    public void CopyTo (T[] array);
    public void CopyTo (T[] array, int arrayIndex);
    public void CopyTo (int index, T[] array, int arrayIndex, int count);
    public ReadOnlyCollection<T> AsReadOnly();
    public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput>
                                              converter);

    // inne
    public void Reverse(); // odwraca kolejność elementów listy
    public int Capacity { get;set; } // wymusza rozszerzenie wewnętrznej tablicy
    public void TrimExcess(); // obcina wewnętrzną tablicę do potrzebnego rozmiaru
    public void Clear(); // usuwa wszystkie elementy, tak że Count=0
}
public delegate TOutput Converter <TInput, TOutput> (TInput input);
```

Oprócz tych składowych klasa `List<T>` zawiera egzemplarzowe wersje wszystkich metod przeszukiwania i sortowania klasy `Array`.

Poniżej znajduje się przykład demonstrujący sposób użycia własności i metod klasy `List`. Przykłady przeszukiwania i sortowania zamieściliśmy w podrozdziale „Klasa Array”.

```
List<string> words = new List<string>(); // nowa lista typu string

words.Add ("melon");
words.Add ("awokado");
words.AddRange (new[] { "banan", "pomarańcza" } );
words.Insert (0, "liczi"); // wstawianie na początku
words.InsertRange (0, new[] { "pomelo", "nashi" }); // wstawianie na początku

words.Remove ("melon");
words.RemoveAt (3); // usunięcie czwartego elementu
words.RemoveRange (0, 2); // usunięcie dwóch pierwszych elementów

// usunięcie wszystkich łańcuchów zaczynających się literą 'n'
words.RemoveAll (s => s.StartsWith ("n"));

Console.WriteLine (words [0]); // pierwsze słowo
Console.WriteLine (words [words.Count - 1]); // ostatnie słowo
foreach (string s in words) Console.WriteLine (s); // wszystkie słowa
List<string> subset = words.GetRange (1, 2); // drugie i trzecie słowo
```

```
string[] wordsArray = words.ToArray(); // tworzy nową tablicę typizowaną

// kopiowanie pierwszych dwóch elementów na koniec istniejącej tablicy
string[] existing = new string [1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCastWords = words.ConvertAll (s => s.ToUpper());
List<int> lengths = words.ConvertAll (s => s.Length);
```

Niegeneryczna klasa `ArrayList` jest wykorzystywana głównie ze względu na zgodność z kodem napisanym dla platformy 1.x. Używający jej programista musi się posługiwać niezręcznymi rzutowaniami, jak pokazano w poniższym przykładzie:

```
ArrayList al = new ArrayList();
al.Add ("hello");
string first = (string) al [0];
string[] strArr = (string[]) al.ToArray (typeof (string));
```

Kompilator nie może zweryfikować takich operacji rzutowania, przez co np. poniższy kod przejdzie kompilację i spowoduje awarię podczas wykonywania programu:

```
int first = (int) al [0]; // wyjątek wykonawczy
```

Klasa `ArrayList` pod względem funkcjonalności przypomina `List<object>`. Obie przydają się, gdy potrzebna jest lista elementów różnego typu niemających wspólnego typu bazowego (innego niż `object`). Klasa `ArrayList` może być lepszym wyborem, gdy podczas pracy z listą używa się refleksji (rozdział 19.). Techniki refleksji łatwiej jest stosować w odniesieniu do niegenerycznego typu `ArrayList` niż do typu generycznego `List<object>`.

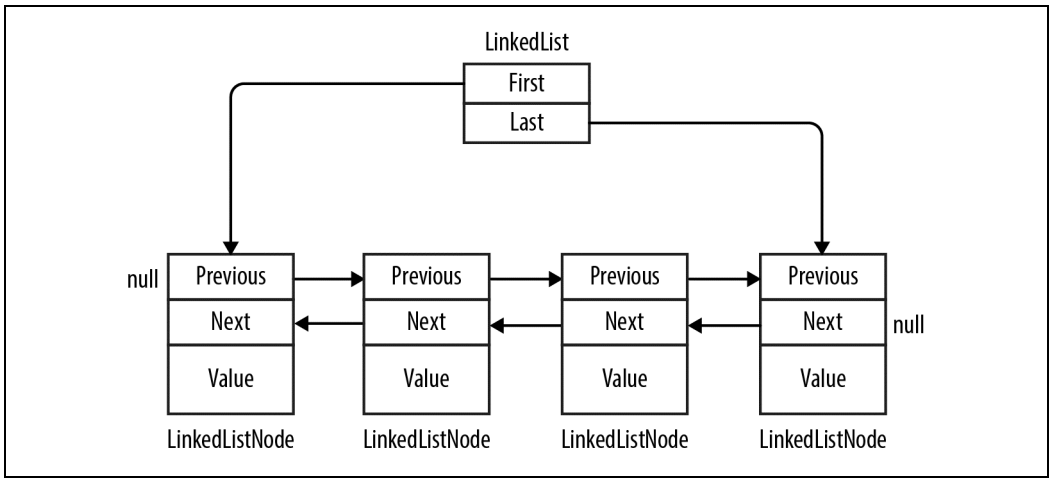
Jeśli do programu zaimportuje się przestrzeń nazw `System.Linq`, to listę `ArrayList` można przekonwertować na listę generyczną za pomocą metod `Cast` i `ToList`:

```
ArrayList al = new ArrayList();
al.AddRange (new[] { 1, 5, 9 } );
List<int> list = al.Cast<int>().ToList();
```

`Cast` i `ToList` to metody rozszerzeniowe klasy `System.Linq.Enumerable`.

Klasa LinkedList<T>

Generyczna klasa `LinkedList<T>` stanowi implementację listy powiązanej dwustronnie (rysunek 7.4). Lista powiązana dwustronnie to łańcuch węzłów, w którym każdy węzeł zawiera referencję do poprzedniego i następnego węzła oraz do samego siebie. Największą zaletą tej struktury jest to, że każdy element można efektywnie wstawić w dowolnym miejscu listy, ponieważ wymaga to tylko utworzenia nowego węzła i zaktualizowania kilku referencji. Jednak znalezienie miejsca do wstawienia elementu może być czasochłonne, ponieważ nie istnieje żaden mechanizm bezpośredniego indeksowania takiej listy. Trzeba się przyjrzeć każdemu węzłowi i nie można korzystać z algorytmów wyszukiwania binarnego.



Rysunek 7.4. Klasa LinkedList<T>

Klasa `LinkedList<T>` implementuje interfejsy `IEnumerable<T>` i `ICollection<T>` (oraz ich niegeneryczne wersje). Nie implementuje natomiast interfejsu `IList<T>`, ponieważ nie obsługuje dostępu do elementów za pomocą indeksów. Implementacja węzłów ma postać następującej klasy:

```
public sealed class LinkedListNode<T>
{
  public LinkedList<T> List { get; }
  public LinkedListNode<T> Next { get; }
  public LinkedListNode<T> Previous { get; }
  public T Value { get; set; }
}
```

Nowy węzeł można dodać na pozycji określonej względem innego węzła lub na początku albo na końcu listy. W klasie `LinkedList<T>` znajdują się następujące metody do wykonywania takich operacji:

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst (T value);

public void AddLast (LinkedListNode<T> node);
public LinkedListNode<T> AddLast (T value);

public void AddAfter (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value);

public void AddBefore (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore (LinkedListNode<T> node, T value);
```

Istnieje też podobny zestaw metod do usuwania elementów:

```
public void Clear();

public void RemoveFirst();
public void RemoveLast();

public bool Remove (T value);
public void Remove (LinkedListNode<T> node);
```

Klasa `LinkedList<T>` zawiera też kilka wewnętrznych pól do rejestrowania liczby przechowywanych elementów oraz referencji do pierwszego i ostatniego elementu. Pola te są dostępne w postaci następujących publicznych własności:

```
public int Count { get; } // szybkie
public LinkedListNode<T> First { get; } // szybkie
public LinkedListNode<T> Last { get; } // szybkie
```

Dodatkowo klasa `LinkedList<T>` udostępnia następujące metody wyszukiwania (każda wymaga, aby lista była wewnętrznie przeliczalna):

```
public bool Contains (T value);
public LinkedListNode<T> Find (T value);
public LinkedListNode<T> FindLast (T value);
```

Ponadto klasa `LinkedList<T>` obsługuje kopiowanie elementów do tablicy, aby można się było posługiwać indeksami, oraz umożliwia tworzenie enumeratorów, aby można się było posługiwać pętlą `foreach`:

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator();
```

Oto przykład użycia klasy `LinkedList<string>`:

```
var tune = new LinkedList<string>();
tune.AddFirst ("do"); // do
tune.AddLast ("sol"); // do - sol

tune.AddAfter (tune.First, "re"); // do - re - sol
tune.AddAfter (tune.First.Next, "mi"); // do - re - mi - sol
tune.AddBefore (tune.Last, "fa"); // do - re - mi - fa - sol

tune.RemoveFirst(); // re - mi - fa - sol
tune.RemoveLast(); // re - mi - fa

LinkedListNode<string> miNode = tune.Find ("mi");
tune.Remove (miNode); // re - fa
tune.AddFirst (miNode); // mi - re - fa

foreach (string s in tune) Console.WriteLine (s);
```

Klasy Queue<T> i Queue

Klasy `Queue<T>` i `Queue` to struktury danych typu FIFO (ang. *first-in, first-out* — pierwszy na wejściu, pierwszy na wyjściu) udostępniające m.in. metody `Enqueue` (do dodawania elementów na końcu kolejki) i `Dequeue` (do pobierania i usuwania elementów z początku kolejki). Ponadto dostępna jest metoda `Peek` zwracająca element z początku struktury bez jego usuwania oraz własność `Count` umożliwiająca sprawdzenie przed zdejmowaniem elementów, czy jakieś w ogóle istnieją.

Choć kolejki są przeliczalne, nie implementują interfejsów `IList<T>` ani `IList`, ponieważ do składowych nie można odwoływać się bezpośrednio za pomocą indeksów. W razie potrzeby dostępna jest jednak metoda `ToArray` służąca do kopiowania elementów do tablicy zapewniającej swobodny dostęp do swojej zawartości:

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
```

```
  public Queue();
  public Queue (IEnumerable<T> collection); // kopiuje istniejące elementy
  public Queue (int capacity); // pozwala ograniczyć automatyczne zmiany rozmiaru
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public T Dequeue();
  public void Enqueue (T item);
  public Enumerator<T> GetEnumerator(); // aby można było korzystać z pętli foreach
  public T Peek();
  public T[] ToArray();
  public void TrimExcess();
}
```

Poniżej przedstawiamy przykład użycia klasy `Queue<int>`:

```
var q = new Queue<int>();
q.Enqueue (10);
q.Enqueue (20);
int[] data = q.ToArray(); // eksport zawartości do tablicy
Console.WriteLine (q.Count); // "2"
Console.WriteLine (q.Peek()); // "10"
Console.WriteLine (q.Dequeue()); // "10"
Console.WriteLine (q.Dequeue()); // "20"
Console.WriteLine (q.Dequeue()); // spowoduje zgłoszenie wyjątku (pusta kolejka)
```

Wewnętrzna implementacja kolejek opiera się na tablicy, której rozmiar jest zmieniany zgodnie z zapotrzebowaniem — podobną implementację ma generyczna klasa `List`. Kolejka utrzymuje indeksy wskazujące bezpośrednio na pierwszy i ostatni element, dzięki czemu operacje dokładania i wyjmowania elementów są bardzo szybkie (z wyjątkiem sytuacji, gdy spowodują zmianę rozmiaru struktury).

Klasy Stack<T> i Stack

Klasy `Stack<T>` i `Stack` to struktury danych typu LIFO (ang. *last-in, first-out* — ostatni na wejściu, pierwszy na wyjściu) udostępniające m.in. metody `Push` (do dodawania elementu na wierzchu stosu) i `Pop` (do pobierania i usuwania elementów z wierzchu stosu). Dostępne są też niedestrukcyjna metoda `Peek`, jak również własność `Count` i metoda `ToArray`, umożliwiające eksport danych do struktury pozwalającej na ich swobodne przeglądanie:

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Stack();
  public Stack (IEnumerable<T> collection); // kopiuje istniejące elementy
  public Stack (int capacity); // pozwala ograniczyć automatyczne zmiany rozmiaru
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public Enumerator<T> GetEnumerator(); // aby można było korzystać z pętli foreach
  public T Peek();
  public T Pop();
  public void Push (T item);
  public T[] ToArray();
  public void TrimExcess();
}
```

Poniżej znajduje się przykład użycia klasy `Stack<int>`:

```
var s = new Stack<int>();
s.Push (1); // s = 1
s.Push (2); // s = 1,2
s.Push (3); // s = 1,2,3
Console.WriteLine (s.Count); // drukuje 3
Console.WriteLine (s.Peek()); // drukuje 3, s = 1,2,3
Console.WriteLine (s.Pop()); // drukuje 3, s = 1,2
Console.WriteLine (s.Pop()); // drukuje 2, s = 1
Console.WriteLine (s.Pop()); // drukuje 1, s = <empty>
Console.WriteLine (s.Pop()); // powoduje wyjątek
```

Wewnętrzna implementacja stosów bazuje na tablicy, której rozmiar zmienia się w razie potrzeby, podobnie jak w przypadku klas `Queue<T>` i `List<T>`.

Klasa BitArray

Klasa `BitArray` to kolekcja wartości typu `bool` z możliwością dynamicznej zmiany rozmiaru. Pozwala efektywniej wykorzystać pamięć niż zwykła tablica lub struktura `List` wartości typu `bool`, ponieważ do przechowywania każdego elementu potrzebuje tylko jednego bitu, podczas gdy normalnie wartość typu `bool` zajmuje jeden bajt.

Indeksator struktury `BitArray` odczytuje i zapisuje pojedyncze bity:

```
var bits = new BitArray(2);
bits[1] = true;
```

Dostępne są cztery metody bitowe: `And`, `Or`, `Xor` oraz `Not`. Wszystkie oprócz ostatniej przyjmują obiekty typu `BitArray`:

```
bits.Xor (bits); // bitowe lub wykluczające na samym sobie
Console.WriteLine (bits[1]); // fałsz
```

Klasy HashSet<T> i SortedSet<T>

Klasy `HashSet<T>` i `SortedSet<T>` to generyczne kolekcje, które wprowadzono odpowiednio w .NET Framework 3.5 i 4.0. Obie wyróżniają się następującymi cechami:

- Metody `Contains` charakteryzują się dużą szybkością działania dzięki posługiwaniu się algorytmem wyszukiwania wykorzystującym wartości skrótu.
- Nie przechowują duplikatów i niepostrzeżenie ignorują żądania dodania elementów takich samych jak elementy istniejące.
- Nie ma możliwości odwołania się do elementu po jego pozycji.

Klasa `SortedSet<T>` przechowuje elementy w określonym porządku, a `HashSet<T>` nie przechowuje.

Wspólne cechy tych dwóch klas są dziełem implementacji interfejsu `ISet<T>`.

Z powodów historycznych klasa `HashSet<T>` znajduje się w zestawie *System.Core.dll*, a klasa `SortedSet<T>` i interfejs `ISet<T>` w zestawie *System.dll*.

Implementację klasy HashSet<T> stanowi tablica skrótów przechowująca tylko klucze. Natomiast struktura SortedSet<T> jest zaimplementowana jako drzewo czerwono-czarne.

Obie kolekcje implementują interfejs ICollection<T> i udostępniają metody, jakich można się spodziewać, a więc np.: Contains, Add i Remove. Dodatkowo istnieje też wykorzystująca predykat metoda RemoveWhere do usuwania elementów.

Poniżej przedstawiamy przykład utworzenia zbioru HashSet<char> z istniejącej kolekcji, sprawdzenia, czy wybrany element jest dostępny w zbiorze, oraz jej przejrzenia (zwróć uwagę na usunięcie duplikatów):

```
var letters = new HashSet<char> ("gdyby kózka nie skakała");

Console.WriteLine (letters.Contains ('g')); // prawda
Console.WriteLine (letters.Contains ('j')); // fałsz

foreach (char c in letters) Console.Write (c); // gdyb kózaniesł
```

(Przekazanie obiektu typu string do konstruktora klasy HashSet<char> nie spowodowało błędu, ponieważ klasa string implementuje interfejs IEnumerable<char>).

Najciekawsze metody to te operujące na zbiorach. Poniższe metody mają charakter **destrukcyjny** w tym sensie, że modyfikują zbiór:

```
public void UnionWith (IEnumerable<T> other); // dodaje
public void IntersectWith (IEnumerable<T> other); // usuwa
public void ExceptWith (IEnumerable<T> other); // usuwa
public void SymmetricExceptWith (IEnumerable<T> other); // usuwa
```

Natomiast te metody tylko sprawdzają coś w zbiorze, więc nie mają charakteru destrukcyjnego:

```
public bool IsSubsetOf (IEnumerable<T> other);
public bool IsProperSubsetOf (IEnumerable<T> other);
public bool IsSupersetOf (IEnumerable<T> other);
public bool IsProperSupersetOf (IEnumerable<T> other);
public bool Overlaps (IEnumerable<T> other);
public bool SetEquals (IEnumerable<T> other);
```

Metoda UnionWith dodaje wszystkie elementy z drugiego zbioru do pierwszego (wykluczając duplikaty). Metoda IntersectWith usuwa elementy, których nie ma w obu zbiorach. Z naszego zbioru znaków możemy np. wydobyć wszystkie samogłoski w następujący sposób:

```
var letters = new HashSet<char> ("gdyby kózka nie skakała");
letters.IntersectWith ("aeioóuy");
foreach (char c in letters) Console.Write (c); // yóaie
```

Metoda ExceptWith usuwa określone elementy ze zbioru źródłowego. W poniższym przykładzie pozbawiamy nasz zbiór wszystkich samogłosek:

```
var letters = new HashSet<char> ("gdyby kózka nie skakała");
letters.ExceptWith ("aeioóu");
foreach (char c in letters) Console.Write (c); // gdb kznsł
```

Metoda SymmetricExceptWith usuwa wszystkie elementy oprócz tych, które występują tylko w jednym lub drugim zbiorze:

```
var letters = new HashSet<char> ("gdyby kózka nie skakała");
letters.SymmetricExceptWith ("toby nogi nie złamała");
foreach (char c in letters) Console.Write (c); // dkóstom
```

Ponieważ klasy `HashSet<T>` i `SortedSet<T>` implementują interfejs `IEnumerable<T>`, ich metody przyjmują jako argumenty inne typy zbiorów (lub kolekcji).

Klasa `SortedSet<T>` ma wszystkie składowe klasy `HashSet<T>` oraz dodatkowo:

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; }
```

Ponadto konstruktor klasy `SortedSet<T>` opcjonalnie przyjmuje obiekt typu `IComparer<T>` (a nie *komparator równości*).

Oto jeden z efektów załadowania naszego przykładowego ciągu liter do zbioru `SortedSet<char>`:

```
var letters = new SortedSet<char> ("gdyby kózka nie skakała");
foreach (char c in letters) Console.Write (c); // abdegiknsyzół
```

Teraz możemy pobrać wszystkie litery z przedziału od `f` do `j`:

```
foreach (char c in letters.GetViewBetween ('f', 'j'))
Console.Write (c); // gi
```

Słowniki

Słownik jest kolekcją, w której każdy przechowywany element stanowi parę klucz – wartość. Tego typu struktur danych najczęściej używa się do przeszukiwania i sortowania list.

Platforma .NET zawiera definicję standardowego protokołu słownikowego w postaci interfejsów `IDictionary` i `IDictionary<TKey, TValue>` oraz definicji kilku klas słownikowych ogólnego przeznaczenia. Różnią się one między sobą następującymi cechami:

- niektóre sortują elementy, a inne nie;
- niektóre pozwalają na dostęp do elementów wg pozycji (indeksu) i klucza, a inne nie;
- niektóre są generyczne, a inne nie;
- niektóre pozwalają na szybkie pobieranie elementów wg klucza dużych słowników, a inne nie.

W tabeli 7.1 przedstawiono zestawienie klas słownikowych i dzielących je różnic. Wartości czasowe są wyrażone w milisekundach dla 50 000 operacji na słowniku z kluczami i są wartościami całkowitoliczbowymi wykonywanymi w komputerze PC z procesorem 1,5 GHz. (Różnice między generycznymi i niegenerycznymi wersjami wynikają z konieczności stosowania pakowania w jednym przypadku i występują tylko w odniesieniu do typów wartościowych).

Tabela 7.1. Klasy słownikowe

Typ	Struktura wewnętrzna	Indeksowanie	Narzut pamięciowy (średnia liczba bajtów na element)	Szybkość — losowe wstawianie	Szybkość — wstawianie sekwencyjne	Szybkość — pobieranie elementów wg klucza
Niesortowane						
`Dictionary <K,V>`	Tablica skrótów	Nie	22	30	30	20
`Hashtable`	Tablica skrótów	Nie	38	50	50	30
`ListDictionary`	Lista powiązana	Nie	36	50 000	50 000	50 000
`OrderedDictionary`	Tablica skrótów + tablica	Tak	59	70	70	40
Sortowane						
`SortedDictionary <K,V>`	Drzewo czerwono-czarne	Nie	20	130	100	120
`SortedList <K,V>`	Dwie tablice	Tak	2	3300	30	40
`SortedList`	Dwie tablice	Tak	27	4500	100	180

W notacji wielkiego O czasy pobierania elementów wg klucza wynoszą:

- `O(1)` dla `Hashtable`, `Dictionary` oraz `OrderedDictionary`;
- `O(log n)` dla `SortedDictionary` i `SortedList`;
- `O(n)` dla `ListDictionary` (i typów niesłownikowych, takich jak `List<T>`).

Parametr *n* reprezentuje liczbę elementów w kolekcji.

Interfejs IDictionary<TKey,TValue>

Interfejs `IDictionary<TKey,TValue>` to standardowy protokół dla wszystkich kolekcji przechowujących pary klucz – wartość. Stanowi on rozszerzenie interfejsu `ICollection<T>`, do którego dodaje metody i własności umożliwiające dostęp do elementów na podstawie kluczy dowolnego typu:

```
public interface IDictionary <TKey, TValue> :
  ICollection <KeyValuePair <TKey, TValue>>, IEnumerable
{
  bool ContainsKey (TKey key);
  bool TryGetValue (TKey key, out TValue value);
  void Add (TKey key, TValue value);
  bool Remove (TKey key);
```

```
  TValue this [TKey key] { get; set; } // główny indeksator — wg klucza
  ICollection <TKey> Keys { get; } // zwraca tylko klucze
  ICollection <TValue> Values { get; } // zwraca tylko wartości
}
```

W .NET Framework 4.5 dodano interfejs o nazwie `IReadOnlyDictionary<TKey, ↪TValue>` definiujący podzbiór składowych słownika tylko do odczytu. Odpowiada on typowi `IMapView<K,V>` Windows Runtime i został wprowadzony głównie jako jego odpowiednik.

Aby dodać element do słownika, należy wywołać metodę `Add` albo użyć metody dostępowej indeksu — ta druga dodaje element do słownika, jeśli nie ma w nim jeszcze takiego klucza (a jeśli jest, to dokonuje modyfikacji elementu). W żadnej implementacji słownika nie może się powtórzyć ani jeden klucz, więc dwukrotne wywołanie metody `Add` z takim samym kluczem powoduje wyjątek.

Aby pobrać element ze słownika, należy się posłużyć indeksatorem albo metodą `TryGetValue`. Jeżeli dany klucz nie istnieje, indeksator zgłasza wyjątek, podczas gdy metoda `TryGetValue` zwraca fałsz. Istnienie elementu w strukturze można sprawdzić za pomocą metody `ContainsKey`. Jeśli jednak później dany element trzeba pobrać, będzie to oznaczało konieczność wykonania dwóch takich samych operacji wyszukiwania.

Operacja enumeracji struktury typu `IDictionary<TKey,TValue>` zwraca sekwencję struktur `KeyValuePair`:

```
public struct KeyValuePair <TKey, TValue>
{
  public TKey Key { get; }
  public TValue Value { get; }
}
```

Istnieje możliwość przeglądania tylko kluczy lub tylko wartości przy użyciu własności `Keys` i `Values`.

W następnej sekcji przedstawiamy przykład użycia tego interfejsu w połączeniu z generyczną klasą `Dictionary`.

Interfejs IDictionary

Niegeneryczny interfejs `IDictionary` jest bardzo podobny do generycznego `IDictionary<TKey,TValue>`, ale różni się od niego w dwóch ważnych aspektach. Należy wiedzieć o tych różnicach, ponieważ interfejs `IDictionary` występuje w starym kodzie (włącznie z samą platformą .NET Framework):

- Próba pobrania przez indeksator nieistniejącego klucza kończy się zwróceniem wartości `null` (a nie wyjątkiem).
- Do sprawdzania, czy dany element znajduje się w strukturze, służy metoda `Contains`, a nie `ContainsKey`.

Efektem przeliczania niegenerycznego `IDictionary` jest zwrot sekwencji struktur `DictionaryEntry`:

```
public struct DictionaryEntry
{
  public object Key { get; set; }
  public object Value { get; set; }
}
```

Klasy Dictionary<TKey,TValue> i Hashtable

Generyczna klasa `Dictionary` należy do najczęściej używanych kolekcji (obok `List<T>`). Do przechowywania kluczy i wartości wykorzystuje tablicę skrótów (ang. *hashtable*), dzięki czemu jest szybka i efektywna.

Niegeneryczna wersja klasy `Dictionary<TKey,TValue>` nazywa się `Hashtable`. Nie istnieje niegeneryczna klasa o nazwie `Dictionary`. Dlatego pisząc `Dictionary`, mamy na myśli generyczną klasę `Dictionary<TKey,TValue>`.

Klasa `Dictionary` implementuje zarówno generyczny, jak i niegeneryczny interfejs `IDictionary`, przy czym generyczna wersja jest udostępniona publicznie. W istocie `Dictionary` to „podręcznikowa" implementacja generycznego interfejsu `IDictionary`.

Oto przykład jej użycia:

```
var d = new Dictionary<string, int>();

d.Add("Jeden", 1);
d["Dwa"] = 2; // dodaje element do słownika, ponieważ nie ma w nim jeszcze "dwa"
d["Dwa"] = 22; // modyfikuje słownik, ponieważ "dwa" już w nim jest
d["Trzy"] = 3;

Console.WriteLine (d["Dwa"]); // drukuje "22"
Console.WriteLine (d.ContainsKey ("Jeden")); // prawda (szybka operacja)
Console.WriteLine (d.ContainsValue (3)); // prawda (wolna operacja)
int val = 0;
if (!d.TryGetValue ("jedeN", out val))
  Console.WriteLine ("Brak"); // "Brak" (wielkość liter ma znaczenie)

// trzy różne sposoby przeglądania słownika:

foreach (KeyValuePair<string, int> kv in d) // Jeden; 1
  Console.WriteLine (kv.Key + "; " + kv.Value); // Dwa; 22
                                                // Trzy; 3

foreach (string s in d.Keys) Console.Write (s); // JedenDwaTrzy
Console.WriteLine();
foreach (int i in d.Values) Console.Write (i); // 1223
```

Wewnętrzna tablica skrótów konwertuje klucz każdego elementu na całkowitoliczbową wartość skrótu, która jest pseudoniepowtarzalna, a następnie za pomocą specjalnego algorytmu konwertuje ten skrót na klucz mieszający. Wewnętrznie za pomocą tego klucza wybierany jest „kubełek", do którego należy dany wpis. Jeśli kubełek zawiera więcej niż jedną wartość, to jest przeszukiwany algorytmem liniowym. Dobra funkcja mieszająca nie dąży do zwracania wyłącznie niepowtarzalnych skrótów (co w większości przypadków byłoby niemożliwe), tylko do równomiernego rozmieszczenia skrótów w 32-bitowej przestrzeni całkowitoliczbowej. W ten sposób unika się groźby powstania bardzo małej liczby bardzo dużych (i nieefektywnych) kubełków.

Klucze w słowniku mogą być każdego typu, pod warunkiem że można je porównywać i tworzyć z nich wartości skrótu. Domyślnie klucze porównuje się za pomocą metody klucza `object.Equals`, a pseudoniepowtarzalna wartość skrótu jest obliczana przez metodę klucza `GetHashCode`. Można to zmienić przez przesłonięcie tych metod lub dostarczenie obiektu implementującego interfejs

`IEqualityComparer` podczas tworzenia słownika. Możliwość tę wykorzystuje się często, gdy stosowane są klucze łańcuchowe w celu użycia komparatora nierozróżniającego wielkości liter:

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

Szerzej na ten temat piszemy jeszcze w podrozdziale „Dołączanie protokołów równości i porządkowania".

Tak jak w przypadku wszystkich innych typów kolekcji, wydajność słownika można nieco podnieść przez określenie w konstruktorze oczekiwanego rozmiaru struktury, co pozwala ograniczyć potrzebę wykonywania operacji zmiany rozmiaru (lub w ogóle jej uniknąć).

Niegeneryczna wersja nazywa się `Hashtable` i pod względem funkcjonalnym jest podobna do generycznej, tylko udostępnia niegeneryczny interfejs `IDictionary`, o którym była mowa wcześniej.

Wadą klas `Dictionary` i `Hashtable` jest to, że nie sortują przechowywanych elementów. Nie zachowują nawet kolejności dodawania wpisów. Jak wszystkie słowniki, struktury te nie pozwalają na zapisanie dwóch takich samych kluczy.

Gdy w .NET Framework 2.0 wprowadzono generyczne kolekcje, programiści CLR postanowili nadać im nazwy zgodne z tym, co reprezentują (`Dictionary`, `List`), a nie odnoszące się do ich wewnętrznej implementacji (`Hashtable`, `ArrayList`). Choć jest to dobra decyzja, dzięki której w przyszłości będzie można bez przeszkód zmieniać podstawową implementację, wybrane nazwy nie pozwalają się zorientować, jaką *wydajnością* (a informacja ta jest często najważniejszym kryterium wyboru) charakteryzują się poszczególne kolekcje.

Klasa OrderedDictionary

`OrderedDictionary` to niegeneryczny słownik przechowujący elementy w kolejności ich dodawania. W tej strukturze elementy dostępne są zarówno wg indeksu, jak i wg klucza.

`OrderedDictionary` nie jest słownikiem *posortowanym*.

Klasa `OrderedDictionary` jest kombinacją klas `Hashtable` i `ArrayList`, tzn. zawiera całą funkcjonalność pierwszej i kilka dodatkowych funkcji, takich jak `RemoveAt` i indeksator całkowitoliczbowy. Ponadto struktura ta udostępnia własności `Keys` i `Values` zwracające elementy w pierwotnym porządku.

Klasę tę wprowadzono w .NET 2.0, ale wciąż nie wiadomo dlaczego nie ma jej wersji generycznej.

Klasy ListDictionary i HybridDictionary

Klasa `ListDictionary` przechowuje dane w liście powiązanej jednostronnie. Nie sortuje elementów, ale zapisuje je w kolejności dodawania. Struktura ta działa bardzo wolno, gdy jest duża. Jedyny sens jej istnienia to wysoka wydajność dla bardzo małych list (zawierających mniej niż dziesięć elementów).

Klasa `HybridDictionary` to `ListDictionary` automatycznie konwertująca się na `Hashtable` po osiągnięciu określonego rozmiaru w celu uniknięcia problemów wydajnościowych. Chodzi o to, by jak

najoszczędniej operować pamięcią, gdy słownik jest mały, oraz by zachować dobrą wydajność, kiedy się powiększy. Biorąc jednak pod uwagę czas potrzebny na przeprowadzenie konwersji — oraz to, że klasa `Dictionary` też nie jest zbyt ciężka ani wolna — nikomu nie zaszkodzi użycie słownika `Dictionary` w zamian.

Obie te klasy występują tylko w wersji niegenerycznej.

Słowniki sortowane

Platforma .NET zawiera dwie klasy słownikowe o wewnętrznej strukturze sprawiającej, że ich zawartość jest zawsze posortowana wg klucza:

- `SortedDictionary<TKey,TValue>`,
- `SortedList<TKey,TValue>`[1].

(W tej sekcji skracamy zapis `<TKey,TValue>` do postaci `<,>`).

Implementacja klasy `SortedDictionary<,>` bazuje na drzewie czerwono-czarnym — strukturze danych, która charakteryzuje się taką samą wydajnością w każdej operacji wstawiania i pobierania elementu.

Implementację klasy `SortedList<,>` stanowi uporządkowana para tablic, co zapewnia szybkie pobieranie danych (za pomocą binarnego algorytmu wyszukiwania), ale mało efektywne wstawianie elementów (ponieważ trzeba przesunąć istniejące elementy, aby zrobić miejsce dla dodawanego).

Klasa `SortedDictionary<,>` jest znacznie szybsza od `SortedList<,>` we wstawianiu elementów w losowej kolejności (zwłaszcza gdy struktura danych jest duża). Za to `SortedList<,>` dodatkowo umożliwia odnoszenie się do elementów wg indeksu lub wg klucza. Mając posortowaną listę, można bezpośrednio przejść do **n**-tego elementu sekwencji (posługując się indeksatorem lub własnościami `Keys` i `Values`). Aby zrobić to samo ze strukturą `SortedDictionary<,>`, należy ręcznie przejrzeć *n* elementów. (Ewentualnie można napisać klasę łączącą posortowany słownik z listą).

Żadna z tych trzech kolekcji nie pozwala na duplikowanie kluczy (tak jest we wszystkich słownikach).

Poniżej przedstawiamy przykład, w którym za pomocą refleksji ładujemy wszystkie metody zdefiniowane w klasie `System.Object` do posortowanej listy, w której klucze stanowią nazwy metod, a następnie przeglądamy te klucze i ich wartości:

```
// MethodInfo znajduje się w przestrzeni nazw System.Reflection

var sorted = new SortedList <string, MethodInfo>();

foreach (MethodInfo m in typeof (object).GetMethods())
  sorted [m.Name] = m;

foreach (string name in sorted.Keys)
  Console.WriteLine (name);

foreach (MethodInfo m in sorted.Values)
  Console.WriteLine (m.Name + " zwraca obiekt typu " + m.ReturnType);
```

[1] stnieje też identyczna pod względem funkcjonalności niegeneryczna wersja o nazwie `SortedList`.

Oto wynik pierwszego przeglądu:

```
Equals
GetHashCode
GetType
ReferenceEquals
ToString
```

A to jest wynik drugiej pętli:

```
Equals zwraca obiekt typu System.Boolean
GetHashCode zwraca obiekt typu System.Int32
GetType zwraca obiekt typu System.Type
ReferenceEquals zwraca obiekt typu System.Boolean
ToString zwraca obiekt typu System.String
```

Zauważ, że zawartość wstawiliśmy do słownika za pomocą indeksatora. Gdybyśmy zamiast niego użyli metody `Add`, zgłosiłaby wyjątek, ponieważ klasa `object`, której dotyczy stosowana przez nas refleksja, przeciąża metodę `Equals`, a do słownika nie można dodać dwa razy takiego samego klucza. Dzięki użyciu indeksatora drugi wpis po prostu zastąpi poprzedni i nie wystąpi żaden błąd.

Jeśli trzeba zapisać kilka elementów pod takim samym kluczem, to elementy wartości można zdefiniować jako listy:

```
SortedList <string, List<MethodInfo>>
```

Wracając do naszego przykładu: poniższa instrukcja pobiera obiekt `MethodInfo`, którego klucz to `GetHashCode`. Dokładnie tak samo pracowalibyśmy ze zwykłym słownikiem:

```
Console.WriteLine (sorted ["GetHashCode"]); // Int32 GetHashCode()
```

Wszystko, co do tej pory zrobiliśmy, dałoby się wykonać także z klasą `SortedDictionary<,>`. Ale poniższe dwa wiersze kodu zadziałają tylko z listą posortowaną, ponieważ pobieramy w nich ostatni klucz i ostatnią wartość:

```
Console.WriteLine (sorted.Keys [sorted.Count - 1]); // ToString
Console.WriteLine (sorted.Values[sorted.Count - 1].IsVirtual); // prawda
```

Kolekcje i pośredniki z możliwością dostosowywania

Opisane w poprzednich podrozdziałach klasy kolekcji są wygodne w użyciu, ponieważ można bezpośrednio tworzyć ich egzemplarze. Niestety, nie ma możliwości kontrolowania sposobu dodawania ani usuwania elementów. Możliwość taka jest jednak potrzebna w aplikacjach, w których używane są ściśle typizowane kolekcje, np.:

- aby uruchomić zdarzenie w reakcji na dodanie lub usunięcie elementu;
- aby zaktualizować własności z powodu dodania lub usunięcia elementu;
- aby wykryć „niedozwoloną” operację dodawania lub usuwania i zgłosić wyjątek (gdy np. operacja łamie jakieś zasady biznesowe).

Platforma .NET Framework zawiera klasy kolekcji przeznaczone do użytku właśnie w takich sytuacjach. Znajdują się one w przestrzeni nazw `System.Collections.ObjectModel` i tak naprawdę są

pośrednikami lub opakowaniami implementującymi interfejs `IList<T>` lub `IDictionary<,>` przez przekazywanie metod do podstawowej kolekcji. Każda operacja `Add`, `Remove` i `Clear` zostaje przekierowana przez metodę wirtualną, która po przesłonięciu stanowi rodzaj „bramy”.

Klasy kolekcji z możliwością dostosowywania są powszechnie wykorzystywane do tworzenia publicznie dostępnych kolekcji, takich jak np. kolekcja kontrolek publicznych klasy `System.Windows.Form`.

Klasy Collection<T> i CollectionBase

Klasa `Collection<T>` to modyfikowalne opakowanie klasy `List<T>`.

Implementuje ona interfejsy `IList<T>` i `IList` oraz definiuje cztery dodatkowe metody wirtualne i chronioną własność:

```
public class Collection<T> :
  IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
  // ...
  protected virtual void ClearItems();
  protected virtual void InsertItem (int index, T item);
  protected virtual void RemoveItem (int index);
  protected virtual void SetItem (int index, T item);

  protected IList<T> Items { get; }
}
```

Metody wirtualne stanowią bramę, przez którą można „się podpiąć”, aby zmienić lub rozszerzyć normalną funkcjonalność listy. Chroniona własność `Items` umożliwia implementatorowi uzyskanie bezpośredniego dostępu do „wewnętrznej listy” — w ten sposób można dokonywać zmian wewnątrz bez uruchamiania metod wirtualnych.

Metody wirtualne nie muszą być przesłonięte. Można je zostawić nietknięte, aż nadejdzie potrzeba modyfikacji domyślnego działania listy. Poniżej znajduje się schematyczny przykład ilustrujący typowy sposób użycia klasy `Collection<T>`:

```
public class Animal
{
  public string Name;
  public int Popularity;

  public Animal (string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : Collection <Animal>
{
  // AnimalCollection to już w pełni funkcjonalna lista zwierząt.
  // Nie jest potrzebny żaden dodatkowy kod.
}

public class Zoo // klasa, która będzie udostępniać AnimalCollection,
{ // normalnie zawierałaby dodatkowe składowe
  public readonly AnimalCollection Animals = new AnimalCollection();
}
```

```
class Program
{
  static void Main()
  {
    Zoo zoo = new Zoo();
    zoo.Animals.Add (new Animal ("Kangur", 10));
    zoo.Animals.Add (new Animal ("Król Lew", 20));
    foreach (Animal a in zoo.Animals) Console.WriteLine (a.Name);
  }
}
```

W tej postaci klasa `AnimalCollection` nie przewyższa funkcjonalnością prostej klasy `List<Animal>`, ponieważ została stworzona tylko po to, by można ją było w przyszłości rozszerzyć. Dla przykładu do klasy `Animal` dodamy teraz własność `Zoo`, aby można było określić, w którym zoo mieszka dane zwierzę, i dodatkowo przesłonimy wszystkie metody wirtualne klasy `Collection<Animal>`, aby zapewnić automatyczną obsługę tej własności:

```
public class Animal
{
  public string Name;
  public int Popularity;
  public Zoo Zoo { get; internal set; }
  public Animal(string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : Collection <Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
  protected override void InsertItem (int index, Animal item)
  {
    base.InsertItem (index, item);
    item.Zoo = zoo;
  }
  protected override void SetItem (int index, Animal item)
  {
    base.SetItem (index, item);
    item.Zoo = zoo;
  }
  protected override void RemoveItem (int index)
  {
    this [index].Zoo = null;
    base.RemoveItem (index);
  }
  protected override void ClearItems()
  {
    foreach (Animal a in this) a.Zoo = null;
    base.ClearItems();
  }
}

public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
```

Klasa `Collection<T>` zawiera też konstruktor przyjmujący obiekty implementujące interfejs `IList<T>`. Jednak inaczej niż jest w przypadku innych klas kolekcji, przekazana lista nie zostanie *skopiowana*, tylko nowy egzemplarz klasy `Collection<T>` będzie *pośrednikiem* do opakowywanej listy, więc wszelkie wprowadzane w niej zmiany będą widoczne przez tego pośrednika (choć *bez* uruchamiania metod wirtualnych klasy `Collection<T>`).

Klasa CollectionBase

Klasa `CollectionBase` to niegeneryczna wersja klasy `Collection<T>`, którą wprowadzono w wersji 1.0 platformy. Zapewnia bardzo podobną funkcjonalność jak `Collection<T>`, tylko jest mniej wygodna w użyciu. Zamiast szablonowych metod: `InsertItem`, `RemoveItem`, `SetItem` i `ClearItem`, klasa `Collec`
`↪tionBase` zawiera metody „zaczepowe", które podwajają tylko zestaw potrzebnych metod: `OnInsert`, `OnInsertComplete`, `OnSet`, `OnSetComplete`, `OnRemove`, `OnRemoveComplete`, `OnClear` oraz `OnClearComplete`. Jako że klasa `CollectionBase` nie jest generyczna, tworząc jej podklasę, należy dodatkowo zaimplementować metody typizowane — przynajmniej indeksator i metodę `Add`.

Klasy KeyedCollection<TKey,TItem> i DictionaryBase

Klasa `KeyedCollection<TKey,TItem>` jest podklasą klasy `Collection<TItem>`. W pewnych obszarach rozszerza jej funkcjonalność, a w innych ogranicza. Jeśli chodzi o rozszerzenia, to dodaje możliwość odnoszenia się do elementów wg klucza, tak jak w słownikach. Likwiduje natomiast możliwość tworzenia pośredników listy wewnętrznej.

Kolekcja `KeyedCollection` przypomina w pewnym stopniu klasę `OrderedDictionary`, ponieważ także jest kombinacją liniowej listy i tablicy skrótów. Jednak w odróżnieniu od niej nie implementuje interfejsu `IDictionary` i nie rozpoznaje pojęcia *pary* klucz – wartość. Klucze są tworzone z samych elementów przez abstrakcyjną metodę `GetKeyForItem`. Oznacza to, że kolekcję taką można przeglądać w taki sam sposób jak zwykłą listę.

Klasę `KeyedCollection<TKey,TItem>` można traktować jak `Collection<TItem>` z szybkim wyszukiwaniem po kluczach.

Jako że jest ona podklasą klasy `Collection<>`, dziedziczy wszystkie jej składniki z wyjątkiem możliwości podawania istniejącej listy konstruktorowi. Dodatkowe składowe, jakie definiuje, to:

```
public abstract class KeyedCollection <TKey, TItem> : Collection <TItem>

  // ...

  protected abstract TKey GetKeyForItem(TItem item);
  protected void ChangeItemKey(TItem item, TKey newKey);

  // szybkie wyszukiwanie wg klucza to dodatek do wyszukiwania indeksowego
  public TItem this[TKey key] { get; }

  protected IDictionary<TKey, TItem> Dictionary { get; }
}
```

Metodę `GetKeyForItem` należy przesłonić tak, aby prawidłowo obliczała klucze elementów utworzonego obiektu. Metodę `ChangeItemKey` należy wywołać, jeżeli zmieni się własność przechowująca klucz elementu, aby zaktualizować słownik wewnętrzny. Własność `Dictionary` zwraca wewnętrzny

słownik wykorzystywany do wyszukiwania, tworzony w chwili dodania pierwszego elementu. W razie potrzeby można zmienić tę zasadę w konstruktorze tak, aby utworzenie słownika następowało dopiero po przekroczeniu pewnego progu (do tego czasu elementy są wyszukiwane liniowo). Dobrym powodem do tego, by nie określać progu utworzenia słownika, jest to, że może on być przydatny do utworzenia kolekcji `ICollection<>` kluczy przy użyciu własności `Keys` klasy `Dictionary`.

Klasy `KeyedCollection<,>` najczęściej używa się do tworzenia kolekcji elementów dostępnych poprzez indeksy i nazwy. W ramach przykładu zaimplementujemy naszą klasę `AnimalCollection` jako `KeyedCollection<string,Animal>`:

```
public class Animal
{
  string name;
  public string Name
  {
    get { return name; }
    set {
      if (Zoo != null) Zoo.Animals.NotifyNameChange (this, value);
      name = value;
    }
  }
  public int Popularity;
  public Zoo Zoo { get; internal set; }

  public Animal (string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}

public class AnimalCollection : KeyedCollection <string, Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }

  internal void NotifyNameChange (Animal a, string newName)
  {
    this.ChangeItemKey (a, newName);
  }

  protected override string GetKeyForItem (Animal item)
  {
    return item.Name;
  }

  // poniższe metody byłyby zaimplementowane, jak w poprzednim przykładzie
  protected override void InsertItem (int index, Animal item)...
  protected override void SetItem (int index, Animal item)...
  protected override void RemoveItem (int index)...
  protected override void ClearItems()...
}

public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
}
```

```
class Program
{
  static void Main()
  {
    Zoo zoo = new Zoo();
    zoo.Animals.Add (new Animal ("Kangur", 10));
    zoo.Animals.Add (new Animal ("Król Lew", 20));
    Console.WriteLine (zoo.Animals [0].Popularity); // 10
    Console.WriteLine (zoo.Animals ["Mr Sea Lion"].Popularity); // 20
    zoo.Animals ["Kangaroo"].Name = "Mr Roo";
    Console.WriteLine (zoo.Animals ["Mr Roo"].Popularity); // 10
  }
}
```

DictionaryBase

`DictionaryBase` to niegeneryczna wersja klasy `KeyedCollection`. Jest to stara klasa, którą zbudowano w całkiem odmienny sposób niż nowsze — implementuje interfejs `IDictionary` i podobnie jak `CollectionBase` wykorzystuje nieporęczne metody zaczepowe: `OnInsert`, `OnInsertComplete`, `OnSet`, `OnSetComplete`, `OnRemove`, `OnRemoveComplete`, `OnClear` oraz `OnClearComplete` (i dodatkowo `OnGet`). Największą zaletą implementacji słownika `IDictionary` zamiast stosowania podejścia klasy `Keyed➥Collection` jest to, że nie trzeba tworzyć podklasy, aby uzyskać klucze. Ponieważ jednak klasa `DictionaryBase` istnieje przede wszystkim po to, by utworzyć jej podklasę, ta zaleta nie ma znaczenia. Ulepszony model klasy `KeyedCollection` z pewnością powstał dlatego, że klasa ta została napisana kilka lat później, kiedy programiści mogli spojrzeć na nią z pewnej perspektywy. Dlatego też klasę `DictionaryBase` należy traktować głównie jako środek do zapewnienia zgodności ze starym kodem.

Klasa ReadOnlyCollection<T>

Klasa `ReadOnlyCollection<T>` jest opakowaniem, albo *pośrednikiem*, zapewniającym widok tylko do odczytu kolekcji. Jest przydatna, gdy klasa ma publicznie udostępnić kolekcję, której nie można modyfikować z zewnątrz, a którą można zmieniać wewnętrznie.

Kolekcja tylko do odczytu przyjmuje kolekcję wejściową w konstruktorze, do którego ma stałą referencję. Nie tworzy ona statycznej kopii przekazanej kolekcji, więc wszelkie późniejsze zmiany kolekcji wejściowej są widoczne przez opakowanie.

W ramach przykładu powiedzmy, że chcemy utworzyć klasę publicznie udostępniającą listę łańcuchów tylko do odczytu o nazwie `Names`:

```
public class Test
{
  public List<string> Names { get; private set; }
}
```

To dopiero połowa pracy. Choć inne typy nie mogą nic przypisać do własności `Names`, nadal mogą wywoływać na liście metody `Add`, `Remove` i `Clear`. Rozwiązaniem problemu jest klasa `ReadOnly➥Collection<T>`:

```
public class Test
{
  List<string> names;
  public ReadOnlyCollection<string> Names { get; private set; }
```

```
  public Test()
  {
    names = new List<string>();
    Names = new ReadOnlyCollection<string> (names);
  }

  public void AddInternally() { names.Add ("test"); }
}
```

Teraz tylko składowe klasy `Test` mogą zmieniać listę nazw:

```
Test t = new Test();

Console.WriteLine (t.Names.Count); // 0
t.AddInternally();
Console.WriteLine (t.Names.Count); // 1

t.Names.Add ("test"); // błąd kompilacji
((IList<string>) t.Names).Add ("test"); // NotSupportedException
```

Dołączanie protokołów równości i porządkowania

W podrozdziałach „Sprawdzanie równości" i „Określanie kolejności" w rozdziale 6. opisaliśmy standardowe protokoły platformy .NET umożliwiające porównywanie typów i obliczanie ich skrótów. Typ implementujący te protokoły może prawidłowo funkcjonować w słowniku i liście posortowanej bez żadnego dodatkowego przygotowania. Mówiąc konkretniej:

- Typ, dla którego metody `Equals` i `GetHashCode` zwracają sensowne wyniki, może być wykorzystywany jako klucz w strukturach `Dictionary` i `Hashtable`.
- Typ implementujący interfejsy `IComparable` i `IComparable<T>` może być używany jako klucz we wszystkich *sortowanych* słownikach i listach.

Domyślna implementacja operacji sprawdzania równości i porównywania typu zazwyczaj odzwierciedla jego „naturalne" cechy. Ale programiście nie zawsze zależy na domyślnym znaczeniu tych operacji. Czasami potrzebny jest np. słownik, w którym klucze są typu `string` i nie ma znaczenia wielkość liter. Innym razem ktoś może potrzebować listy klientów posortowanej wg kodów pocztowych. Dlatego na platformie .NET Framework stworzono dodatkowy zestaw protokołów „dołączanych" o dwojakim przeznaczeniu:

- Dzięki nim można włączać alternatywne sposoby określania równości i kolejności elementów.
- Umożliwiają korzystanie ze słowników lub posortowanych kolekcji z typem kluczy, które w swej istocie nie są porównywalne.

Protokoły dołączane obejmują następujące interfejsy:

`IEqualityComparer` i `IEqualityComparer<T>`

- Odpowiada za *sprawdzanie równości i obliczanie skrótów*.
- Rozpoznawany przez `Hashtable` i `Dictionary`.

`IComparer` i `IComparer<T>`

- Odpowiada za *porównywanie pod względem kolejności*.
- Rozpoznawany przez słowniki i kolekcje posortowane; także `Array.Sort`.

Każdy interfejs występuje w wersji generycznej i niegenerycznej. Dodatkowo interfejsy `IEquality`
`↪Comparer` mają domyślną implementację w klasie o nazwie `EqualityComparer`.

Ponadto w .NET Framework 4.0 dodano dwa nowe interfejsy: `IStructuralEquatable` i `IStructural`
`↪Comparable`, które umożliwiają wykonywanie porównań strukturalnych na klasach i tablicach.

IEqualityComparer i EqualityComparer

Interfejs `IEqualityComparer` włącza niestandardowe operacje sprawdzania równości i obliczania skrótów i jest przeznaczony głównie do pracy z klasami `Dictionary` i `Hashtable`.

Przypomnimy pokrótce wymagania, jakie powinien spełniać słownik skrótów. Dla każdego klucza musi odpowiadać na dwa pytania:

- Czy jest taki sam jak inny?
- Jaka jest jego całkowitoliczbowa wartość skrótu?

Komparator sprawdzający równość odpowiada na te pytania przez implementację interfejsów `IEqualityComparer`:

```
public interface IEqualityComparer<T>
{
  bool Equals (T x, T y);
  int GetHashCode (T obj);
}

public interface IEqualityComparer // wersja niegeneryczna
{
  bool Equals (object x, object y);
  int GetHashCode (object obj);
}
```

Aby napisać własny komparator, należy zaimplementować jeden z tych interfejsów lub oba (implementacja obu zapewnia największą elastyczność). Ponieważ jednak jest to dość czasochłonne zajęcie, ewentualnie można też utworzyć podklasę abstrakcyjnej klasy `EqualityComparer`, której definicja wygląda następująco:

```
public abstract class EqualityComparer<T> : IEqualityComparer,
                                            IEqualityComparer<T>
{
  public abstract bool Equals (T x, T y);
  public abstract int GetHashCode (T obj);

  bool IEqualityComparer.Equals (object x, object y);
  int IEqualityComparer.GetHashCode (object obj);

  public static EqualityComparer<T> Default { get; }
}
```

Klasa `EqualityComparer` implementuje oba interfejsy, więc programiście pozostaje już tylko przesłonięcie dwóch abstrakcyjnych metod.

Semantyka metod `Equals` i `GetHashCode` jest taka sama jak metod `object.Equals` i `object.GetHashCode` opisanych w rozdziale 6. W poniższym przykładzie definiujemy klasę `Customer` z dwoma polami i piszemy komparator porównujący imiona i nazwiska:

```
public class Customer
{
  public string LastName;
  public string FirstName;

  public Customer (string last, string first)
  {
    LastName = last;
    FirstName = first;
  }
}
public class LastFirstEqComparer : EqualityComparer <Customer>
{
  public override bool Equals (Customer x, Customer y)
   => x.LastName == y.LastName && x.FirstName == y.FirstName;

  public override int GetHashCode (Customer obj)
   => (obj.LastName + ";" + obj.FirstName).GetHashCode();
}
```

Aby pokazać, jak to działa, utworzymy dwóch klientów:

```
Customer c1 = new Customer ("Barański", "Jan");
Customer c2 = new Customer ("Barański", "Jan");
```

Jako że nie przesłoniliśmy metody `object.Equals`, zastosowanie ma normalna semantyka referencyjna:

```
Console.WriteLine (c1 == c2); // fałsz
Console.WriteLine (c1.Equals (c2)); // fałsz
```

To samo dzieje się, gdy użyjemy naszych obiektów w słowniku, nie określając własnego komparatora:

```
var d = new Dictionary<Customer, string>();
d [c1] = "Jan";
Console.WriteLine (d.ContainsKey (c2)); // fałsz
```

A teraz podajemy własny komparator:

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Jan";
Console.WriteLine (d.ContainsKey (c2)); // prawda
```

W tym przypadku musielibyśmy uważać, aby nie zmienić pól `FirstName` ani `LastName` klienta podczas używania obiektu w słowniku. W przeciwnym razie zmieniłaby się wartość skrótu i słownik przestałby działać.

Metoda EqualityComparer<T>.Default

Metoda `EqualityComparer<T>.Default` zwraca ogólny komparator, którego można używać zamiast statycznej metody `object.Equals`. Jej zaletą jest to, że najpierw sprawdza, czy typ `T` implementuje interfejs `IEquatable<T>`, i jeśli tak, to wywołuje tamtą implementację, pozwalając uniknąć pakowania. Jest to szczególnie przydatne w metodach generycznych:

```
static bool Foo<T> (T x, T y)
{
  bool same = EqualityComparer<T>.Default.Equals (x, y);
  ...
```

IComparer i Comparer

Komparatory umożliwiają dołączanie do posortowanych słowników i kolekcji niestandardowej logiki porównawczej.

Dla nieposortowanego słownika, np. `Dictionary` i `Hashtable`, komparator jest bezużyteczny — w takim przypadku potrzebny jest `IEqualityComparer`, aby możliwe było obliczanie wartości skrótów. Analogicznie komparator równości jest nieprzydatny dla posortowanych słowników i kolekcji.

Oto definicje interfejsu `IComparer`:

```
public interface IComparer
{
  int Compare(object x, object y);
}
public interface IComparer <in T>
{
  int Compare(T x, T y);
}
```

Tak jak w przypadku komparatorów równości, istnieje klasa abstrakcyjna, którą można rozszerzyć zamiast pisać implementację interfejsów:

```
public abstract class Comparer<T> : IComparer, IComparer<T>
{
  public static Comparer<T> Default { get; }

  public abstract int Compare (T x, T y); // do implementacji przez programistę
  int IComparer.Compare (object x, object y); // do implementacji przez programistę
}
```

Poniżej znajduje się przykład ilustrujący klasę opisującą życzenia i komparator sortujący życzenia wg ważności:

```
class Wish
{
  public string Name;
  public int Priority;

  public Wish (string name, int priority)
  {
    Name = name;
    Priority = priority;
  }
}

class PriorityComparer : Comparer <Wish>
{
  public override int Compare (Wish x, Wish y)
  {
    if (object.Equals (x, y)) return 0; // test bezpieczeństwa
    return x.Priority.CompareTo (y.Priority);
  }
}
```

Test bezpieczeństwa metody `object.Equals` gwarantuje nam, że nigdy nie zaprzeczymy metodzie `Equals`. Wywołanie statycznej metody `object.Equals` jest w tym przypadku lepszym rozwiązaniem niż wywołanie `x.Equals`, ponieważ metoda ta zadziała także, gdy `x` będzie `null`!

Poniżej przedstawiamy przykład użycia naszego komparatora `PriorityComparer` do posortowania listy:

```
var wishList = new List<Wish>();
wishList.Add (new Wish ("Pokój", 2));
wishList.Add (new Wish ("Bogactwo", 3));
wishList.Add (new Wish ("Miłość", 2));
wishList.Add (new Wish ("3 kolejne życzenia", 1));

wishList.Sort (new PriorityComparer());
foreach (Wish w in wishList) Console.Write (w.Name + " | ");

// WYNIK: 3 kolejne życzenia | Bogactwo | Miłość | Pokój |
```

W następnym przykładzie klasa `SurnameComparer` pozwala na posortowanie nazwisk w kolejności odpowiedniej dla książki telefonicznej:

```
class SurnameComparer : Comparer <string>
{
  string Normalize (string s)
  {
    s = s.Trim().ToUpper();
    if (s.StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }

  public override int Compare (string x, string y)
   => Normalize (x).CompareTo (Normalize (y));
}
```

Oto przykład użycia tego komparatora w posortowanym słowniku:

```
var dic = new SortedDictionary<string,string> (new SurnameComparer());
dic.Add ("MacPhail", "drugi!");
dic.Add ("MacWilliam", "trzeci!");
dic.Add ("McDonald", "pierwszy!");

foreach (string s in dic.Values)
  Console.Write (s + " "); // pierwszy! drugi! trzeci!
```

Klasa StringComparer

`StringComparer` to gotowa klasa do sprawdzania równości i porównywania łańcuchów, która dodatkowo umożliwia określenie języka i podjęcie decyzji, czy ma być rozpoznawana wielkość liter. Klasa ta implementuje interfejsy `IEqualityComparer` i `IComparer` (oraz ich generyczne wersje), więc można jej używać z każdym typem słownika lub posortowanej kolekcji:

```
// CultureInfo znajduje się w przestrzeni nazw System.Globalization

public abstract class StringComparer : IComparer, IComparer <string>,
                                       IEqualityComparer,
                                       IEqualityComparer <string>
{
```

```
  public abstract int Compare (string x, string y);
  public abstract bool Equals (string x, string y);
  public abstract int GetHashCode (string obj);

  public static StringComparer Create (CultureInfo culture,
                                       bool ignoreCase);
  public static StringComparer CurrentCulture { get; }
  public static StringComparer CurrentCultureIgnoreCase { get; }
  public static StringComparer InvariantCulture { get; }
  public static StringComparer InvariantCultureIgnoreCase { get; }
  public static StringComparer Ordinal { get; }
  public static StringComparer OrdinalIgnoreCase { get; }
}
```

Klasa `StringComparer` jest abstrakcyjna, więc egzemplarze otrzymuje się przez metody statyczne i własności. Metoda `StringComparer.Ordinal` odzwierciedla domyślny sposób sprawdzania równości łańcuchów, a metoda `StringComparer.CurrentCulture` dotyczy porównywania pod względem kolejności.

W poniższym przykładzie tworzymy porządkowy słownik nierozróżniający wielkości liter, taki że `dict["Jan"]` i `dict["JAN"]` znaczą to samo:

```
var dict = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

W następnym przykładzie sortujemy tablicę imion wg zasad australijskiej odmiany angielskiego:

```
string[] names = { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo ("en-AU");
Array.Sort<string> (names, StringComparer.Create (ci, false));
```

Ostatni przykład to rozpoznająca kultury wersja komparatora `SurnameComparer`, który napisaliśmy w poprzedniej sekcji (do porównywania nazwisk w sposób odpowiedni dla książki telefonicznej):

```
class SurnameComparer : Comparer <string>
{
  StringComparer strCmp;
  public SurnameComparer (CultureInfo ci)
  {
    // utworzenie rozpoznającego wielkość liter i kultury komparatora łańcuchów
    strCmp = StringComparer.Create (ci, false);
  }
  string Normalize (string s)
  {
    s = s.Trim();
    if (s.ToUpper().StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }

  public override int Compare (string x, string y)
  {
    // bezpośrednie wywołanie metody Compare na naszym komparatorze StringComparer
    return strCmp.Compare (Normalize (x), Normalize (y));
  }
}
```

Interfejsy IStructuralEquatable i IStructuralComparable

Jak napisaliśmy w poprzednim rozdziale, struktury domyślnie implementują **porównywanie strukturalne**, które polega na tym, że dwie struktury zostają uznane za równe, jeśli wszystkie ich pola są sobie równe. Czasami jednak strukturalne porównywanie i równość są przydatne także w innych

typach, np. tablicach i krotkach, do których można je dodać jako protokoły dołączane. W wersji 4.0 platformy wprowadzono więc dwa nowe interfejsy:

```
public interface IStructuralEquatable
{
  bool Equals (object other, IEqualityComparer comparer);
  int GetHashCode (IEqualityComparer comparer);
}

public interface IStructuralComparable
{
  int CompareTo (object other, IComparer comparer);
}
```

Przekazywane komparatory `IEqualityComparer` i `IComparer` są stosowane do każdego elementu w obiekcie złożonym. Możemy to zademonstrować przy użyciu tablic i krotek, które implementują te interfejsy. W poniższym przykładzie sprawdzamy, czy dwie tablice są sobie równe, najpierw za pomocą domyślnej metody `Equals`, a następnie przy użyciu jej wersji z interfejsu `IStructuralEquatable`:

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };
IStructuralEquatable se1 = a1;
Console.Write (a1.Equals (a2)); // fałsz
Console.Write (se1.Equals (a2, EqualityComparer<int>.Default)); // prawda
```

Oto kolejny przykład:

```
string[] a1 = "gdyby kózka nie skakała".Split();
string[] a2 = "GDYBY KÓZKA NIE SKAKAŁA".Split();
IStructuralEquatable se1 = a1;
bool isTrue = se1.Equals (a2, StringComparer.InvariantCultureIgnoreCase);
```

Krotki działają tak samo:

```
var t1 = Tuple.Create (1, "foo");
var t2 = Tuple.Create (1, "FOO");
IStructuralEquatable se1 = t1;
bool isTrue = se1.Equals (t2, StringComparer.InvariantCultureIgnoreCase);
IStructuralComparable sc1 = t1;
int zero = sc1.CompareTo (t2, StringComparer.InvariantCultureIgnoreCase);
```

Różnica, jeśli chodzi o krotki, jest taka, że ich *domyślne* implementacje operacji sprawdzania równości i porównywania mają charakter strukturalny:

```
var t1 = Tuple.Create (1, "FOO");
var t2 = Tuple.Create (1, "FOO");
Console.WriteLine (t1.Equals (t2)); // prawda
```

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

O'REILLY®

C# jest szybki, efektywny, wygodny — to narzędzie w sam raz dla Ciebie!

C# jest jednym z najlepszych projektów firmy Microsoft. Język ten został od podstaw zaprojektowany jako obiektowy. Charakteryzuje się niezwykłą elastycznością i wszechstronnością. Udostępnia wysokopoziomowe abstrakcje, takie jak wyrażenia, zapytania i kontynuacje asynchroniczne, a równocześnie pozwala na korzystanie z niskopoziomowych mechanizmów, jak własne typy wartościowe programisty czy opcjonalne wskaźniki. C# w wersji 6.0 jest kolejną istotną aktualizacją języka. Programista piszący w C# powinien konsekwentnie poznawać te zmiany.

Niniejsza książka jest zwięzłym kompendium wiedzy o C#, CLR oraz o związanej z C# platformie. Opracowano je z myślą o programistach na co najmniej średnim poziomie zaawansowania. W zrozumiały,a równocześnie dogłębny sposób wyjaśniono takie trudne kwestie, jak współbieżność, bezpieczeństwo i domeny aplikacji. Informacje o nowych składnikach języka C# 6.0 i związanej z nim platformy zostały specjalnie oznaczone. Szczególnie istotny z punktu widzenia programisty może okazać się rozdział o nowym kompilatorze Roslyn, zwanym kompilatorem usługowym.

Najważniejsze zagadnienia ujęte w książce:

- składnia, typy oraz zmienne C#
- bezpieczeństwo kodu i dyrektywy preprocesora
- rdzenne technologie i techniki platformy .NET Framework, w tym LINQ, XML, kolekcje, kontrakty kodu, zarządzanie pamięcią, refleksja, programowanie dynamiczne
- kompilator Roslyn — jego architektura, struktura drzewa składni i model semantyczny

Joseph Albahari — jest autorem kilku książek dotyczących C# oraz LINQ. Napisał też LINQPad, popularny program do testowania zapytań LINQ.

Ben Albahari — były kierownik programowy w Microsofcie; współtworzył takie projekty, jak .NET Compact Framework i ADO.NET. Jeden z założycieli firmy Genamics zajmującej się produkcją narzędzi dla programistów C# i J++ oraz oprogramowania do analizy DNA i sekwencjonowania białek. Jest autorem i współautorem kilku książek dotyczących C#.

Helion

księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-2423-7

Informatyka w najlepszym wydaniu

cena: 129,00 zł