

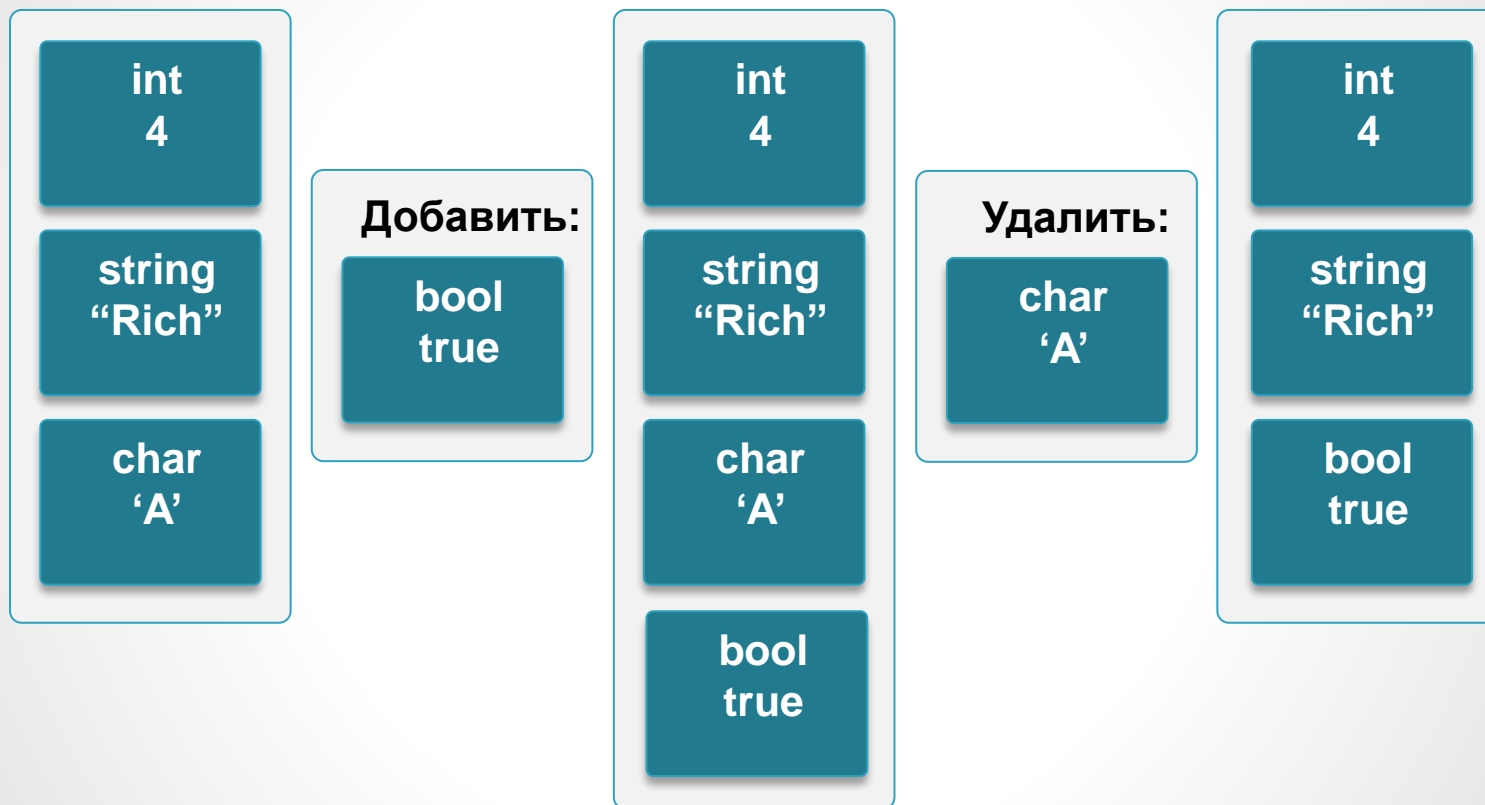
ЛЕКЦИЯ

КОЛЛЕКЦИИ. ОБОБЩЁННЫЕ ТИПЫ

Лектор Крамар Ю.М.

КОЛЛЕКЦИИ. ИХ ТИПЫ

Что такое коллекция?



Использование классов коллекций

- Классы коллекции пространства имен `System.Collections` хранят объекты `System.Object`
- Большинство классов коллекций предоставляют конкретные методы и члены, лежащие в основе их функциональности

Интерфейсы, поддерживаемые классами пространства имен `System.Collections`

ICollection

IDictionary

IEnumerator

ICloneable

IEnumerable

IList

Все коллекции реализуют интерфейс `ICollection`

- `Count`
- `CopyTo()`
- `GetEnumerator()`

Некоторые коллекции реализуют интерфейс `IList`

Использование классов коллекций

Класс `ArrayList` это простой класс коллекция, реализующий массив, который может динамически изменять свой размер

```
ArrayList list = new ArrayList();  
list.Add(3);  
list.Add(4);  
list.Add(6);  
list.Add("String Object");  
list.Remove(6);  
list.RemoveAt(1);  
int temp = ((int)list[0]) * 5;
```

Итерация по коллекции

Все коллекции реализуют интерфейс `ICollection`, определяющий метод `GetEnumerator`, который возвращает объект `Enumerator` (перечислитель), используемый для быстрого перебора всех элементов в коллекции

Тип данных
коллекции

Управляющая
переменная

```
foreach (<type> <control_variable> in <collection>)  
{  
    <foreach_statement_body>  
}
```

Управляющая переменная устанавливается по очереди для каждого элемента коллекции, и операторы в теле цикла `foreach` выполняются для каждого элемента

Область видимости управляющей переменной – оператор `foreach`

Итерация по коллекции

```
ArrayList list = new ArrayList();  
list.Add(99);  
list.Add(10001);  
list.Add(25);  
...  
foreach (int i in list)  
{  
    Console.WriteLine(i);  
}
```

Оператор `foreach` отображает каждое из значений коллекции по очереди в порядке, в котором они встречаются в объекте `ArrayList`

Базовые классы коллекции

ArrayList

- Неупорядоченная коллекция похожая на массив
- Можно добавлять и извлекать элементы с использованием индексирования
- Динамически увеличивается в размерах по мере добавления значений в коллекцию, но автоматически не сжимается при удалении элементов из нее

```
ArrayList al = new ArrayList();  
al.Add("Value 1");  
al.Add("Value 2");  
al.Add("Value 3");  
al.Add("Value 4");  
al.Remove("Value 2"); // Removes "Value 2"  
al.RemoveAt(2); // Removes "Value 4"  
string valueFromCollection = (string)al[1];  
// Returns "Value 3"
```

Value 1

[0]

Value 2

[1]

Value 3

[1]

Value 4

[2]

Базовые классы коллекции

Queue

- FIFO структура данных
- Обеспечивает методы Enqueue и Dequeue вместо методов Add и Remove
- Растет автоматически при добавлении объектов в коллекцию, но автоматически не сжимается при удалении элементов из нее

```
Queue queue = new Queue();  
queue.Enqueue("Value 1");  
queue.Enqueue("Value 2");  
queue.Enqueue("Value 3");  
queue.Enqueue("Value 4");  
string valueFromCollection =  
(string)queue.Dequeue();  
// Returns "Value 1"
```



Базовые классы коллекции

Stack

- FILO структура данных
- Обеспечивает методы Push и Pop вместо методов Add и Remove
- Растет автоматически при добавлении объектов в коллекцию, но автоматически не сжимается при удалении элементов из нее

```
Stack stack = new Stack();
stack.Push("Value 1");
stack.Push("Value 2");
stack.Push("Value 3");
stack.Push("Value 4");
string peekValueFromCollection =
(string)stack.Peek();
// Returns "Value 4"
string valueFromCollection =
(string)stack.Pop();
// Returns "Value 4"
string valueFromCollection2 =
(string)stack.Pop();
// Returns "Value 3"
```

Value 4

Value 3

Value 2

Value 1

Базовые классы коллекции

Hashtable

- Хранит пары ключ–значение в коллекции быстрого доступа
- При добавлении элемента в класс Hashtable с помощью метода Add предоставляются как ключ, так и значение
- Выполнение операции в хеш–таблице начинается с вычисления хеш–функции от ключа
- Каждый объект включает в себя реализацию по умолчанию метода GetHashCode, унаследованный от класса System.Object

```
Hashtable hashtable = new Hashtable();  
hashtable.Add("Key A", "Value 1");  
hashtable.Add("Key B", "Value 2");  
hashtable.Add("Key C", "Value 3");  
hashtable.Add("Key D", "Value 4");  
hashtable.Remove("Key C");  
string valueFromCollection =  
(string)hashtable["Key B"];  
// Returns "Value 2"
```

Value 1	Key A
Value 2	Key B
Value 3	Key C
Value 4	Key D

Использование инициализаторов коллекции

Для добавления элементов в созданный экземпляр класса коллекции обычно используется метод Add

```
ArrayList al = new ArrayList();  
al.Add("Value");  
al.Add("Another Value");
```

Альтернативой написанию нескольких операторов с методом Add является использование инициализатора коллекции

```
ArrayList al = new ArrayList() { "Value", "Another Value" };
```

При добавлении новых объектов в коллекцию можно комбинировать инициализаторы коллекции с инициализаторами объектов

```
ArrayList al = new ArrayList()  
{  
    new Person() {Name="James", Age =45},  
    new Person() {Name="Tom", Age =31}  
};
```

Инициализатор коллекции можно использовать только с коллекцией классов, предоставляющих метод Add

СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ОБОБЩЕННЫХ ТИПОВ

Что такое обобщенный тип?

- Обобщенный тип это тип, специфицирующий один или несколько параметров типа
- Тип параметра указывается при определении класса с помощью угловых скобок
- .NET Framework определяет обобщенные версии некоторых из классов коллекций в пространстве имен `System.Collections.Generic`

```
public class List<T>
```

```
List<int> ages = new List<int>();  
ages.Add(10);  
ages.Add(25);  
...  
int data = ages[0];  
...  
ages.Add("Data"); //СТЕ
```

При создании экземпляра объекта на основе универсального типа необходимо указать тип для замены параметра типа

Извлечь данные из коллекции можно без использования приведения типа

Компиляция обобщенных типов и безопасность типов

Замена параметра типа для указанного типа не просто текстовый механизм замены, компилятор при этом выполняет полную семантическую замену

```
List<string> names = new List<string>();  
names.Add("John");  
...  
string name = names[0];  
List<List<string>> listOfLists = new List<List<string>>();  
listOfLists.Add(names);  
...  
List<string> data = listOfLists[0];
```

Компилятор генерирует различные версии метода Add

```
public void Add(string item);  
public void Add(List<string> item);
```

```
List<int> ages = new List<int>();  
ages.Add(10);
```

```
public void Add(int item);
```


Компилятор не генерирует код упаковки и распаковки для значимых типов

Определение пользовательских обобщенных типов

Для определения пользовательского обобщенного типа нужно добавить один или несколько параметров типа сразу после имени класса в угловых скобках

```
class PrintableCollection<TItem>
{
    TItem[] data;
    int index;
    ...
    public void Insert(TItem item)
    {
        ...
        data[index] = item;
    }
}
```

Имена, указанные для параметров типа, используются в качестве конкретных типов в классе



```
...
PrintableCollection<Person> employeeList =
new PrintableCollection<Person>();
Person employee = new Person(...);
employeeList.Insert(employee);
...
```

```
struct Person
{
    ...
}
```


Определение пользовательских обобщенных типов

Для инициализации членов класса на основе параметра типа значением по умолчанию C# предоставляет ключевое слово `default`

```
class PrintableCollection<TItem>
{
    TItem[] data;
    int index;
    TItem tempData;
    ...
    public PrintableCollection()
    {
        this.tempData = default(TItem);
        ...
    }
    ...
}
```

При компиляции генерируется код, в котором конструкция `default` заменяется значением по умолчанию, зависящим от конкретного типа

Добавление ограничений для обобщенных типов

При определении обобщенного типа можно ограничить типы, которые могут быть использованы в качестве параметров типа, чтобы гарантировать, что они соответствуют определенным критериям или специфическим требованиям типа

```
class PrintableCollection<TItem> where TItem : IPrintable
{
    TItem[] data;
    ...
    public void PrintItems()
    {
        foreach (TItem item in data)
        {
            item.Print();
        }
    }
}
```

Безопасный вызов
метода Print



```
class MyClass<T, Y>
    where T : IComparable, IDisposable
    where Y : T, new()
{ }
```

Добавление ограничений для обобщенных типов

Ограничение	Описание
where T: struct	Аргумент типа должен быть типом значения. Могут быть указаны любые типы значения, кроме Nullable
where T : class	Аргумент типа должны быть ссылочным типом; это относится к любому классу, интерфейсу, делегату или типу массив
where T : new()	Аргумент типа должен иметь public конструктор по умолчанию. Когда ограничение new() используется вместе с другими ограничениями, оно должно быть указано последним
where T : <base class name>	Аргумент типа должны быть наследником указанного базового класса
where T : <interface name>	Аргумент типа должны реализовывать указанный интерфейс. Могут быть указаны несколько ограничений интерфейса. Уточняющий интерфейс может быть универсальным
where T : U	Тип аргумента, который поставляется для T должен вытекать из аргумента, который поставляется для U

ИСПОЛЬЗОВАНИЕ ОБОБЩЕННЫХ МЕТОДОВ И ДЕЛЕГАТОВ

Определение обобщенного метода

Как обобщенные типы, так и обобщенные методы и делегаты содержат параметр типа, который можно использовать в списке параметров и возвращаемом типе для метода или делегата

```
void AddToQueue (Report report)
{
    printQueue.Add(report);
}
```

```
void AddToQueue (ReferenceGuide referenceGuide)
{
    printQueue.Add(referenceGuide);
}
```

Использование обобщенного метода с параметром типа для параметра метода снимает дублирование кода при сохранении безопасности типов

```
void AddToQueue<DocumentType> (DocumentType document)
{
    printQueue.Add(document);
}
```

Определение обобщенного метода

При определении обобщенного метода используются параметры типа

```
ResultType MyMethod<Parameter1Type, ResultType>(Parameter1Type param1)
    where ResultType : new()
{
    ResultType result = new ResultType();
    return result;
}
```

Параметры типа можно использовать в списке параметров метода, типе возвращаемого значения, в любом месте в теле метода

На параметры типа можно добавить ограничения

Использование обобщенных методов

```
T PerformUpdate<T>(T input)
{
    T output = . . . // Update parameter.
    return output;
}
```

При вызове обобщенного метода в дополнение к другим параметрам нужно предоставить параметры типа

```
string result = PerformUpdate<string>("Test");
int result2 = PerformUpdate<int>(1);
```

```
PerformUpdate<string>(1);
```

При компиляции приложения, использующего обобщенный метод, компилятор создает версии обобщенного метода для каждой комбинации параметров типа

```
string PerformUpdate(string input)
{
    string output = . . .
    return output;
}
```

```
int PerformUpdate(int input)
{
    int output = . . .
    return output;
}
```

Использование обобщенных делегатов .NET Framework

Обобщенный делегат определяется с помощью параметров типа аналогично использованию параметров типа в объявлении метода

```
delegate void PrintDocumentDelegate<DocumentType>(DocumentType document);
```

- .NET Framework включает несколько встроенных обобщенных делегатов
- Основными обобщенными делегатами являются Action и Func

- **Action<T>**

Делегат, используемый для инкапсуляции методов, не возвращающих значения

- **Func<T, TResult>**

Делегат, используемый для инкапсуляции методов, возвращающих значение

Использование обобщенных делегатов .NET Framework

Примеры использования делегатов Action и Func

```
Action<string, int> myDelegate = null;
myDelegate += ((param1, param2) =>
{
    Console.WriteLine("{0} : {1}", param1,
param2.ToString());
});
if (myDelegate != null)
{
    myDelegate("Value", 5);
}
```

```
Func<string, int, string> myDelegate = null;
myDelegate += ((param1, param2) =>
{
    return String.Format("{0} : {1}", param1,
param2.ToString());
});
if (myDelegate != null)
{
    string returnedValue;
    returnedValue = myDelegate("Value", 5);
    Console.WriteLine(returnedValue);
}
```

ОПРЕДЕЛЕНИЕ ОБОБЩЕННЫХ ИНТЕРФЕЙСОВ, ВАРИАНТНОСТЬ

Определение обобщенных интерфейсов

При определении обобщенного интерфейса указываются параметры типа, которые используются в членах, определенных в интерфейсе

```
interface IPrinter<DocumentType> where DocumentType : IPrintable
{
    void PrintDocument(DocumentType Document);
    PrintPreview PreviewDocument(DocumentType Document);
}
```

Определение обобщенных интерфейсов

В обобщенном классе можно реализовать обобщенный интерфейс и использовать его для ссылки на класс

```
class Printer<DocumentType> : IPrintable<DocumentType>
    where DocumentType : IPrintable
{
    public void PrintDocument(DocumentType Document)
    {
        // Send document to printer.
        IPrintable doc = (IPrintable) Document;
        PrintService.Print(doc);
    }
    public PrintPreview PreviewDocument(DocumentType Document)
    {
        // Return a new PrintPreview object.
        IPrintable doc = (IPrintable) Document;
        return new PrintPreview(doc)
    }
}
```

Что такое инвариантность?

Все строки являются объектами

```
string myString = "Hello";  
object myObject = myString;
```

```
interface IWrapper<T>  
{  
    void SetData(T data);  
    T GetData();  
}
```

```
class Wrapper<T> : IWrapper<T>  
{  
    private T storedData;  
    void IWrapper<T>.SetData(T  
data)  
    {  
        this.storedData = data;  
    }  
    T IWrapper<T>.GetData()  
    {  
        return this.storedData;  
    }  
}
```

Что такое инвариантность?

```
Wrapper<string> stringWrapper = new Wrapper<string>();  
IWrapper<string> storedStringWrapper = stringWrapper;  
storedStringWrapper.SetData("Hello");  
Console.WriteLine("Stored value is {0}",  
    storedStringWrapper.GetData());
```



Stored value is Hello

```
IWrapper<object> storedObjectWrapper = stringWrapper; //CTE
```

```
IWrapper<object> storedObjectWrapper =  
    (IWrapper<object>)stringWrapper; //RTE
```

Нельзя присвоить объект `IWrapper<A>` ссылке типа `IWrapper`, даже если тип `A` является производным от типа `B`

Определение и реализация ковариантного интерфейса

```
interface IStoreWrapper<T>
{
    void SetData(T data);
}
```

```
interface IRetrieveWrapper<T>
{
    T GetData();
}
```

```
class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>
{
    private T storedData;
    void IStoreWrapper<T>.SetData(T data)
    {
        this.storedData = data;
    }
    T IRetrieveWrapper<T>.GetData()
    {
        return this.storedData;
    }
}
```

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IStoreWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;
Console.WriteLine("Stored value is {0}", retrievedStringWrapper.GetData());
```

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper; //CTE
```

Определение и реализация ковариантного интерфейса

Если параметр типа в обобщенном интерфейсе появляется только в качестве возвращаемого значения методов, можно сообщить компилятору, что некоторые неявные преобразования являются законными и что можно не соблюдать строгую безопасность типов

```
interface IRetrieveWrapper<out T>
{
    T GetData();
}
```

Ковариация позволяет присваивать объект `IRetrieveWrapper<A>` ссылке `IRetrieveWrapper`, пока существует допустимое преобразование из типа `A` в тип `B` или тип `A` является производным от типа `B`

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;

Wrapper<int> intWrapper = new Wrapper<int>();
IStoreWrapper<int> storedIntWrapper = intWrapper;
...
IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;
```


Определение и реализация контравариантного интерфейса

Контравариация позволяет использовать обобщающий интерфейс для ссылки на объект типа В через ссылку на тип А, пока типа В является производным от типа А

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

```
class ObjectComparer : IComparer<object>
{
    int Comparer<object>.Compare(Object x, Object y)
    {
        int xHash = x.GetHashCode();
        int yHash = y.GetHashCode();
        if (xHash == yHash)
            return 0;
        if (xHash < yHash)
            return -1;
        return 1;
    }
}
```

Определение и реализация контравариантного интерфейса

```
Object x = ...;
Object y = ...;
ObjectComparer objectComparer = new ObjectComparer();
IComparer<object> objectComparator = objectComparer;
int result = objectComparator(x, y);
```

```
IComparer<String> stringComparator = objectComparer;
```

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

Ключевое слово `in` сообщает компилятору C#, что в качестве параметра типа методов можно передавать тип `T` или любой тип, производный от `T`

Нельзя использовать тип `T` в качестве возвращаемого типа любых методов

Если тип `A` предоставляет некоторые операции, свойства или поля, и, если тип `B` является производным от типа `A`, он должен поддерживать те же операции (которые могут вести себя иначе, если они были переопределены), свойства и поля

Определение и реализация контравариантного интерфейса

Подводя итоги ковариации и контравариации

Ковариация. Если методы в обобщенном интерфейсе могут возвращать строки, они также могут возвращать объекты. (Все строки являются объектами)

Контравариация. Если методы в обобщенном интерфейсе могут принимать параметры object, они могут принимать параметры string. (Если можно выполнять операции с использованием объекта, значит можно выполнять ту же операцию с использованием строки, потому что все строки являются объектами.)

РЕАЛИЗАЦИЯ ПОЛЬЗОВАТЕЛЬСКОГО КЛАССА КОЛЛЕКЦИИ

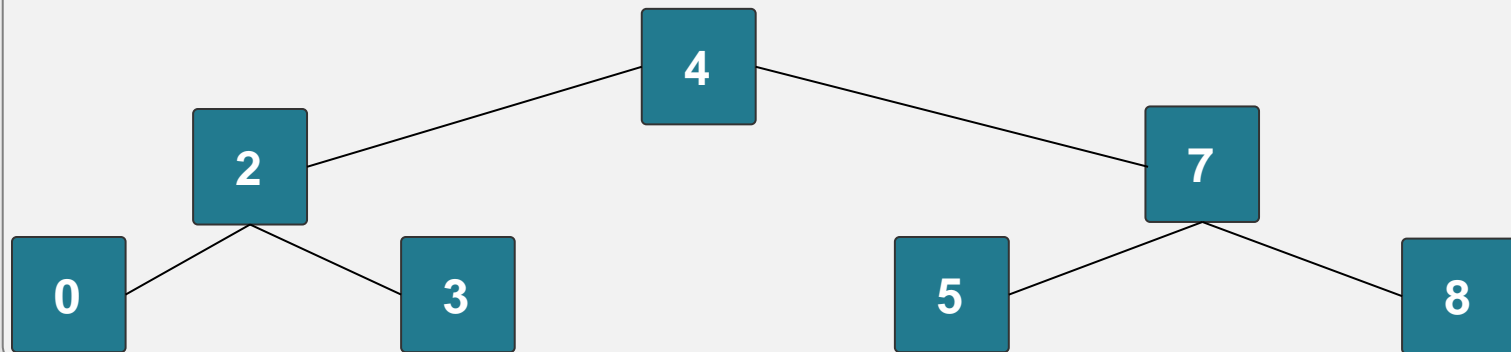
Последовательный список – List<T>



Упорядоченный список – SortedList<T>



Двоичное дерево – необходим пользовательский класс коллекция



Обобщенные интерфейсы коллекций .NET Framework

.NET Framework включает в себя интерфейсы, которые следует реализовывать при разработке пользовательских классов коллекций. Платформа .NET Framework обеспечивает как обобщенные, так и необобщенные версии этих интерфейсов

ICollection<T>

ICollection<T>

ICollection<T>

ICollection<T>

IDictionary<TKey, TValue>

IComparer<T>

ICollection<T> : IEnumerable<T>

- Add()
- Clear()
- Contains()
- CopyTo()
- Remove()
- Count
- IsReadOnly

Обобщенные интерфейсы коллекций .NET Framework

`IList<T> : ICollection<T>`

- `IndexOf()`
- `Insert()`
- `RemoveAt()`
- `Item`

`IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>`

- `Add()`
- `ContainsKey()`
- `GetEnumerator()`
- `Remove()`
- `TryGetValue ()`
- `Item`
- `Keys`
- `Values`

Обобщенные классы коллекции .NET Framework

Пространство имен `System.Collections.Generic` содержит классы, определяющие обобщенные коллекции, которые позволяют пользователям создавать строго типизированные коллекции, обеспечивающие повышенную производительность и безопасность типов по сравнению с необобщенными строго типизированными коллекциями. Платформа .NET Framework представляет

- типы для работы с коллекциями–списками
- типы для работы с коллекциям–множествами
- типы для работы с коллекциями–словарями

`List<T>`

`Queue<T>`

`HashSet<T>`

`LinkedList<T>`

`Stack<T>`

`SortedSet<T>`

`Dictionary<TKey, TValue>`

`SortedDictionary<TKey, TValue>`

Обобщенные классы коллекции

.NET Framework

`List<T> : IList<T>, IList`

- `Add()`
- `AddRange()`
- `Insert()`
- `InsertRange()`
- `Remove()`
- `RemoveAt()`
- `RemoveRange()`
- `RemoveAll()`
- `GetRange()`
- `Contains()`
- `GetEnumerator()`
- `BinarySearch()`
- `Count`
- `Capacity`
- `Item`

Обобщенные классы коллекции

.NET Framework

`LinkedList<T> : ICollection<T>, ICollection`

- `AddAfter()`
- `AddBefore()`
- `AddFirst()`
- `AddLast()`
- `Remove()`
- `RemoveFirst()`
- `RemoveLast()`
- `Find()`
- `Clear()`
- `Contains()`
- `CopyTo()`
- `FindLast()`
- `Count`
- `First`
- `Last`

Обобщенные классы коллекции

.NET Framework

Queue<T> : IEnumerable<T>, ICollection

- Dequeue()
- Enqueue()
- Peek()
- ToArray()
- TrimExcess()
- Contains()
- Clear()
- CopyTo()
- Count

Обобщенные классы коллекции

.NET Framework

`Stack<T> : IEnumerable<T>, ICollection`

- `Pop()`
- `Push()`
- `Peek()`
- `ToArray()`
- `TrimExcess()`
- `Contains()`
- `Clear()`
- `CopyTo()`
- `Count`

Обобщенные классы коллекции .NET Framework

```
// List<Person>
List<Person> db;
db = new List<Person>();
db.Add(new Person { Id = 1, LastName="Ivanov", FirstName = "Ivan" });
db.Add(new Person { Id = 2, LastName="Petrov", FirstName = "Petr"});
db.Add(new Person { Id = 3, LastName="Sidorov", FirstName = "Sidor" });

foreach (Person item in db)
{
    Console.WriteLine("LastName - {0} FirstName {1}", item.LastName,
item.FirstName);
}

// Dictionary<TKey, TValue>
Dictionary<String, Int32> ListCourse = new Dictionary<string,int>();

ListCourse.Add("History", 5);
ListCourse.Add("Logic", 4);

foreach (KeyValuePair<String, Int32> item in ListCourse)
{
    Console.WriteLine("{0}, Mark {1}", item.Key, item.Value);
}
```

Реализация класса коллекции DoubleEndedQueue

```
class DoubleEndedQueue<T> :  
ICollection<T>, IList<T>  
{  
    private List<T> items;  
  
    // Type specific methods.  
    public DoubleEndedQueue() {...}  
  
    public EnqueueItemAtStart() {...}  
  
    public DequeueItemFromStart()  
    {...}  
  
    public EnqueueItemAtEnd() {...}  
  
    public DequeueItemFromEnd() {...}  
  
    // Interface methods.  
    public void Add(T item) {...}  
  
    public void Clear() {...}  
  
    public bool Contains(T item) {...}  
  
    public void CopyTo(T[] array, int  
    arrayIndex) {...}
```

```
    public int Count  
    { get {...} }  
  
    public bool IsReadOnly  
    { get {...} }  
  
    public bool Remove(T item) {...}  
  
    public IEnumerator<T>  
    GetEnumerator() {...}  
  
    IEnumerator  
    IEnumerable.GetEnumerator() {...}  
  
    public int IndexOf(T item) {...}  
  
    public void Insert (int index, T item)  
    {...}  
  
    public void RemoveAt(int index)  
    {...}  
    public T this[int index]  
    {  
        get {...}  
        set {...}  
    }  
}
```

Реализация класса коллекции IntelligentDictionary

```
class IntelligentDictionary<TKey, TValue>
: IDictionary<TKey, TValue>
{
    public IntelligentDictionary() {...}

    public TKey AddItem(TKey key, TValue
value) {...}

    public void Add(TKey key, TValue
value) {...}

    public bool ContainsKey(TKey
key){...}

    public ICollection<TKey> Keys
{ get {...} }

    public bool Remove(TKey key) {...}

    public bool TryGetValue (TKey key,
out TValue value) {...}

    public ICollection<TValue> Values
{ get {...} }

    public TValue this[TKey key]
{ get {...}
  set {...} }
```

```
public void Add(KeyValuePair<TKey,
TValue> item) {...}

public void Clear() {...}

public bool Contains(KeyValuePair<TKey,
TValue> item)
{...}

public void CopyTo(KeyValuePair<TKey,
TValue>[] array, int arrayIndex)
{...}

public int Count { get {...} }

public bool IsReadOnly { get {...} }

public bool Remove(KeyValuePair<TKey,
TValue> item) {...}

public IEnumerator<KeyValuePair<TKey,
TValue>> GetEnumerator() {...}

IEnumerator IEnumerable.GetEnumerator()
{...}
}
```

ПЕРЕЧИСЛИТЕЛИ И НУМЕРАТОРЫ КОЛЛЕКЦИИ

integerCollectionObject



Enumerator



```
foreach (int datum in integerCollectionObject)
{
    //Обработка данных
}
```

Интерфейс IEnumerable<T>

Для поддержки перечисления класс коллекция должен реализовывать интерфейс IEnumerable<T>

Интерфейс IEnumerable<T> определяет единственный метод GetEnumerator, который возвращает объект IEnumerator<T>, предоставляющий логику, требуемую оператору foreach

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

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

предельно
дополнительные перечислители, для этого необходимо предоставить дополнительные методы или свойства для того, чтобы приложения имели доступ к этим перечислителям

Для поддержки инициализаторов коллекции, необходимо реализовать в типе интерфейс IEnumerable и определить метод Add

Интерфейс IEnumerable<T>

```
// public interface IEnumerable<out T> : IEnumerable
class CustomCollectionClass<T> : IEnumerable<T>
{
    public IEnumerator<T> GetEnumerator()
    {
        ...
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }

    // Additional enumerators return
    // instances of the IEnumerable<T>
    // interface.
    public IEnumerable<T> Backwards()
    {
        ...
    }
    ...
}
```

Интерфейс IEnumerable<T>

```
CustomCollectionClass<int> intCollection = new  
CustomCollectionClass<int>();  
intCollection.Add(3);  
intCollection.Add(5);  
intCollection.Add(8);  
intCollection.Add(2);  
intCollection.Add(9);  
intCollection.Add(1);  
intCollection.Add(0);
```

Конструкция foreach использует перечислитель по умолчанию

```
foreach (int temp in intCollection)  
{  
    ...  
}
```

Конструкция foreach использует дополнительный перечислитель Backwards

```
foreach (int temp in intCollection.Backwards())  
{  
    ...  
}
```

Интерфейс IEnumerator<T>

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

Типизированное свойство; возвращает из коллекции текущий элемент

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

Извлекает текущий элемент из коллекции

Перемещает перечислитель к следующему элементу коллекции

Сбрасывает внутреннее состояние перечислителя так, что последующий вызов метода MoveNext располагает перечислитель снова перед первым элементом в коллекции

```
foreach (int datum in integerCollectionObject)
{
    ...
    datum = datum / 2;
}
```

Реализация перечислителя вручную

При реализации интерфейса `IEnumerator<T>` вручную, необходимо предоставить собственные реализации для свойства `Current` и методов `MoveNext` и `Reset`

```
class CustomCollectionClass<T> : IEnumerator<T>
```

```
{
```

```
    T[] vals = new T[10];
```

```
    int pointer = -1;
```

```
    public T Current
```

```
    {
```

```
        get
```

```
        {
```

```
            if (pointer != -1)
```

```
                return vals[pointer];
```

```
            }
```

```
            else
```

```
            {
```

```
                throw new InvalidOperationException();
```

```
            }
```

```
        }
```

```
    }
```

Данные хранятся в массиве

Хранит текущую позицию
перечислителя

Определение свойства только
для чтения Current для
возвращения текущего элемента

Проверка
корректности
текущего
значения

Реализация перечислителя вручную

```
object IEnumerator.Current
{
    get { return (object)Current; }
}
public bool MoveNext()
{
    if (pointer < (vals.Length - 1))
    {
        pointer++;
        return true;
    }
    else
    {
        return false;
    }
}
public void Reset()
{
    pointer = -1;
}
public void Dispose() { }
```

Возвращает свойство Current
типобезопасного
обобщенного интерфейса
IEnumerator<T>

Продвигает
перечислитель к
следующему элементу
коллекции

Сбрасывает внутреннее
состояние перечислителя

Реализация интерфейса IDisposable

Реализация перечислителя с помощью итератора

Итератор является блоком кода, возвращающим (yields) упорядоченную последовательность значений коллекции

Итератор не является членом перечисляемого класса, он только описание перечисляемой последовательности, которую компилятор C# может использовать для создания своего собственного перечислителя

Итератор отличает присутствие одного или нескольких операторов yield:

- `yield return <выражение>` возвращает следующее значение последовательности
- `yield break` прекращает генерацию последовательности


При создании итератора для класса или структуры реализация всего интерфейса `IEnumerator` не требуется – когда компилятор обнаруживает итератор, он автоматически создает методы `Current`, `MoveNext` и `Dispose` интерфейса `IEnumerator` или `IEnumerator<T>`

Итератор вызывается из клиентского кода с помощью оператора `foreach`

Реализация перечислителя с помощью итератора

```
class BasicCollection<T>
{
    private List<T> data = new List<T>();
    public void FillList(params T[] items)
    {
        foreach (var datum in items)
        {
            data.Add(datum);
        }
    }
    public IEnumerator<T> GetEnumerator()
    {
        foreach (var datum in data)
        {
            yield return datum;
        }
    }
    . . .
}
```

Компилятор использует этот код для реализации интерфейса `IEnumerator<T>`, который содержит свойство `Current` и методы `MoveNext` и `Reset`



```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");

foreach (string word in bc)
{
    Console.WriteLine(word);
}
```

Реализация перечислителя с помощью итератора

```
class BasicCollection<T>
{
    private List<T> data = new List<T>();
    . . .
    public IEnumerable<T> Reverse
    {
        get
        {
            for (int i = data.Count - 1; i >= 0; i--)
            {
                yield return data[i];
            }
        }
    }
    . . .
}
```

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");

foreach (string word in bc.Reverse)
{
    Console.WriteLine(word);
}
```

Реализация перечислителя с помощью итератора

Итераторы реализуют концепцию отложенных вычислений

Каждое выполнение оператора `yield return` ведет к выходу из метода и возврату значения, но состояние метода, его внутренние переменные и позиция `yield return` запоминаются, чтобы быть восстановленными при следующем вызове

```
public static class Helper
{
    public static IEnumerable<int> GetNumbers()
    {
        int i = 0;
        while (true) yield return i++;
    }
}
```

```
foreach (var n in Helper.GetNumbers())
{
    Console.WriteLine(n);
    if (n == 2) break;
}
```

ИНИЦИАЛИЗАТОРЫ КОЛЛЕКЦИИ

Инициализаторы коллекции

Вместо того чтобы явно вызывать метод `Add()`, при создании коллекции можно указать список инициализаторов. После этого компилятор организует автоматические вызовы метода `Add()`, используя значения из этого списка. Синтаксис в данном случае ничем не отличается от инициализации массива.

```
List<char> lst = new List<char>() { 'C', 'A', 'E', 'B', 'D', 'F' };
SortedList<int, string> lst =
new SortedList<int, string>() { {1, "один"}, {2, "два" }, {3,
"три"} };
SortedList people = new SortedList()
{ {"Lara", new Person {Name = "Lara", Age = 32}},
  {"Rechard", new Person {Name = "Rechard", Age = 35}}
};
```

Инициализаторы коллекций нельзя использовать в коллекциях типа `Stack`, `Stack<T>`, `Queue` или `Queue<T>`, поскольку в них метод `Add()` не поддерживается.

ШАБЛОН ИТЕРАТОР

Шаблон Итератор

Шаблон Iterator (также известный как **Cursor**) — шаблон проектирования, относится к паттернам поведения. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящих в состав агрегации.

Когда необходим итератор:

1. Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры
2. Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора
3. Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта

Например, такие элементы как дерево, связанный список, хэш-таблица и массив могут быть пролистаны (и модифицированы) с помощью паттерна (объекта) Итератор.

Шаблон Итератор

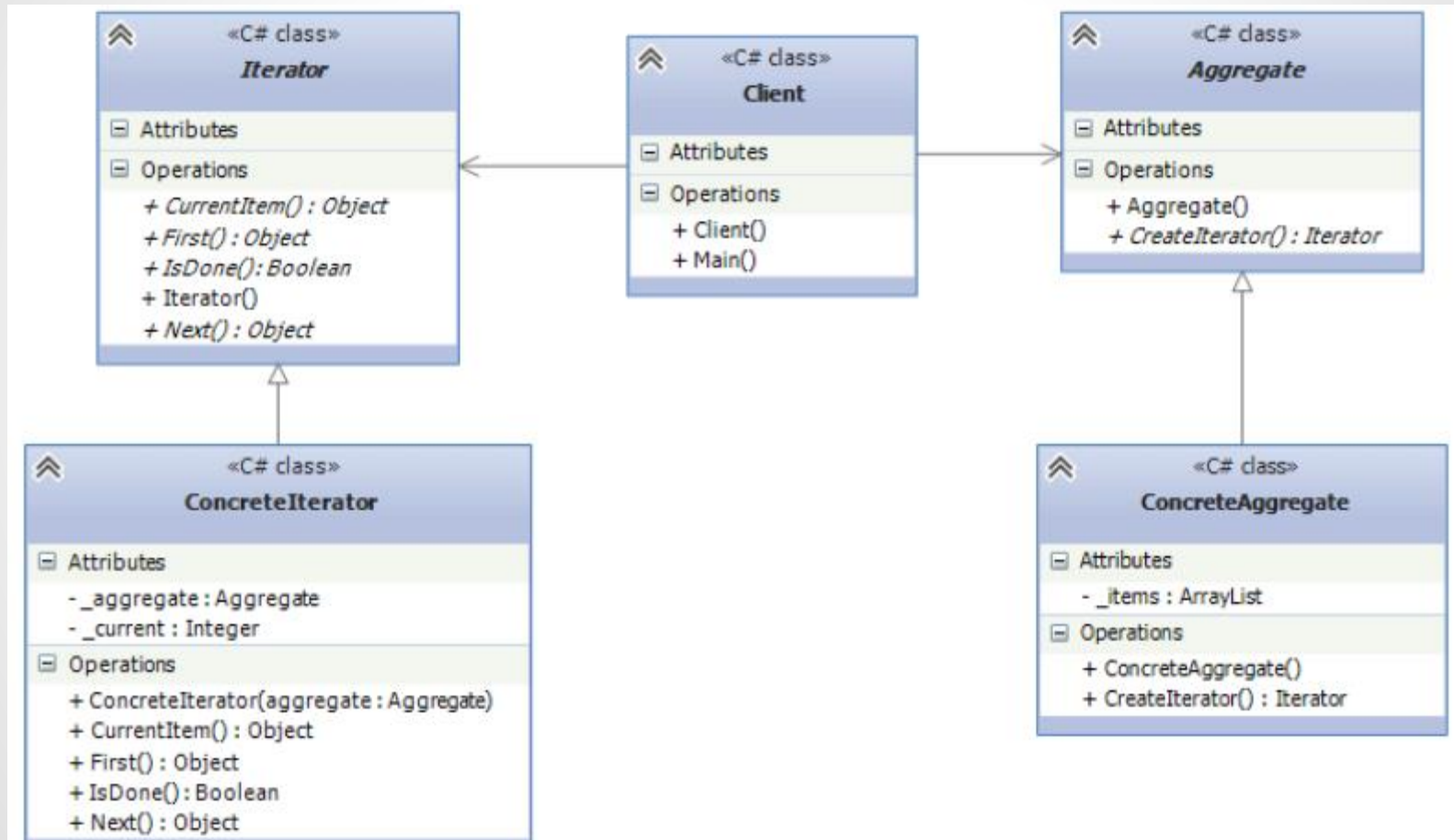
Перебор элементов выполняется объектом итератора, а не самой коллекцией. Это упрощает интерфейс и реализацию коллекции, а также способствует более логичному распределению обязанностей. Особенностью полноценно реализованного итератора является то, что код, использующий итератор, может ничего не знать о типе итерируемого агрегата.

Дополнительно можно обеспечить возможность работы с диапазонами итераторов при отсутствии знания о типе итерируемого агрегата, например:

```
Iterator itBegin = aggregate.begin();
Iterator itEnd = aggregate.end();
func(itBegin, itEnd);
void func(Iterator itBegin, Iterator itEnd)
{
    for( Iterator it = itBegin, it != itEnd; ++it )
    { ... }
}
```


Шаблон Итератор

Demo PatternIterator



Шаблон Итератор в C#

Так как паттерн Итератор предоставляет абстрактный интерфейс для последовательного доступа ко всем элементам составного объекта без раскрытия его внутренней структуры, то этот интерфейс можно использовать кодом, не знающим об устройстве агрегаций, например оператором `foreach`, который перебирает объекты в массиве или коллекции. При этом встроенных классов коллекций существует множество, и каждая из них отличается по своей структуре и поведению.

Ключевым моментом, который позволяет осуществить перебор коллекций с помощью `foreach`, является применение этими классами коллекций паттерна итератор в виде пары интерфейсов `IEnumerable` / `IEnumerator`.

Шаблон Итератор в C#

Интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

```
public interface IEnumerator
{
    bool MoveNext(); // перемещение на одну позицию вперед в
    // контейнере элементов
    object Current {get;} // текущий элемент в контейнере
    void Reset(); // перемещение в начало контейнера
}
```

Интерфейс IEnumerable использует IEnumerator для получения итератора для конкретного типа объекта:

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

Используя данные интерфейсы, можно свести к одному шаблону обработки (с помощью foreach) любые составные объекты

КОЛЛЕКЦИИ. ОБОБЩЁННЫЕ ТИПЫ

Ресурсы:

1. Шилд Г. С# : учебный курс.- СПб.:Питер;К.: Издательская группа BHV, 2002.-512с.
2. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections>