# Обобщенные коллекции. Классы – прототипы

1. Классы-прототипы.
2. Создание собственных классов-прототипов.
3. Обобщенные методы.
4. Параметризированные коллекции.

### 1. Классы-прототипы.

Многие алгоритмы не зависят от типов данных, с которыми они работают. Простейшими примерами могут служить сортировка и поиск. Возможность отделить алгоритмы от типов данных предоставляют **классы-прототипы**(generics) – **параметризированные** классы, имеющие в качестве параметров типы данных. Чаще всего такие классы применяются для хранения данных – в качестве контейнерных классов, или коллекций. Параметризированным коллекциям соответствует пространство имен **System.Collections.Generic.**

Параметризированные типы позволяют создавать классы, структуры, интерфейсы, методы, в которых обрабатываемые данные указываются в виде параметра. Такой класс автоматически становится пригодным для обработки разнотипных данных.

Класс (структура, интерфейс, метод), оперирующий параметризированным типом данных, называется **обобщенным** (обобщенный класс или обобщенный метод) или **классом-прототипом**.

В С# всегда имелась возможность создавать обобщенный код, оперируя ссылками типа **object.** Так как класс object является базовым для всех остальных классов, то по ссылке типа object можно обращаться к объекту любого типа. Таким образом, до появления обобщений для оперирования разнотипными объектами в программах служил обобщенный код, в котором для этой цели использовались ссылки типа object.

Но в таком коде трудно соблюдать **типовую безопасность**, т.к. для преобразования типа object в конкретный тип данных требуется приведение типов. А это служит потенциальным источником ошибок из-за того, что приведение типов может быть выполнено неверно. Обобщения, обеспечивают типовую безопасность и, кроме того, исключают необходимость выполнять приведение типов для преобразования объекта или другого типа обрабатываемых данных.

Таким образом, обобщения расширяют возможности повторного использования кода и позволяют делать это надежно и просто.

Обобщения — это не совсем новая конструкция; подобные концепции присутствуют и в других языках. Например, схожие с обобщениями черты имеют шаблоны С++. Однако между шаблонами С++ и обобщениями .NET есть большая разница. В С++ при создании экземпляра шаблона с конкретным типом необходим исходный код шаблонов. В отличие от шаблонов С++, обобщения являются не только конструкцией языка С#, но также определены для CLR. Это позволяет создавать экземпляры шаблонов с определенным типом-параметром на языке Visual Basic, даже если обобщенный класс определен на С#.

#### Основные преимущества использования обобщений:

1. **Производительность.** Использование типов значений с не обобщенными классами коллекций вызывает **упаковку (boxing)** и **распаковку (unboxing)** при преобразовании в ссылочный тип и обратно.

Классы С# являются ссылочными типами, а структуры — типами значений. .NET позволяет легко преобразовывать типы значений в ссылочные, поэтому их можно использовать там, где ожидаются объекты (т.е. ссылочные типы). Например, объекту можно присвоить значение типа int.

Преобразование типа значений в ссылочный тип называется упаковкой (boxing). Упаковка происходит автоматически, когда метод ожидает параметр ссылочного типа, а ему передается тип значений. С другой стороны, упакованный тип значений может быть обратно преобразован к простому типу значений с помощью распаковки (unboxing). При распаковке требуется операция приведения.

**Безопасность.** Обобщения автоматически обеспечивают типовую безопасность всех операций, т.к. исключают необходимость обращаться к приведению типов и проверять соответствие типов в коде вручную.

**Повторное использование двоичного кода.** Обобщенный класс может быть определен однажды, и на его основе могут быть созданы экземпляры многих типов. При этом не нужно иметь доступ к исходным текстам.

**"Разрастание" кода.** Т. к. определение обобщенного класса включается в сборку, создание на его основе конкретных классов специфических типов не приводит к дублированию кода в IL. Однако когда обобщенные классы компилируются JIT-компилятором в родной машинный код, для каждого конкретного типа значения создается новый класс. Ссылочные типы при этом разделяют общую реализацию одного родного класса. Причина в том, что в случае ссылочных типов каждый элемент представлен в памяти 4-байтным адресом (на 32-разрядных системах) и машинные реализации обобщенного класса с различными ссылочными типами-параметрами не отличаются друг от друга. В отличие от этого, типы значений содержатся в памяти целиком, и поскольку каждый из них требует разного объема памяти, то для каждого из них создаются свои экземпляры классов на основе обобщенного.

Синтаксис объявления обобщенного класса:

class имя\_класса<список\_параметров\_типа> {...}

Ссылка на обобщенный класс:

имя\_класса<список\_аргументов\_типа> имя\_переменной =

new имя\_класса<список\_параметров\_типа> (список\_аргументов\_конструктора);

Пример 1.

class MyCollection<T1, Т2>

{

T1[] beg;

Т2 atr;

int size;

public MyCollection(int s, T1 obj, Т2 a)

{

size = s;

atr = a;

beg = new T1[size];

for (int i = 0; i < size; i++)

beg[i] = obj;

}

public void Show()

{

Console.WriteLine(atr);

for (int i = 0; i < size; i++)

Console.WriteLine(beg[i].ToString());

Console.WriteLine();

}

public void PrintType()

{

Console.WriteLine("Тип объекта1:" + typeof(T1));

Console.WriteLine("Тип объекта2:" + typeof(Т2));

}

}

class Program

{

static void Main(string[] args)

{

MyCollection<int,double> mci = new MyCollection<int,double>(5,1,1.5);

mci.Show();

mci.PrintType();

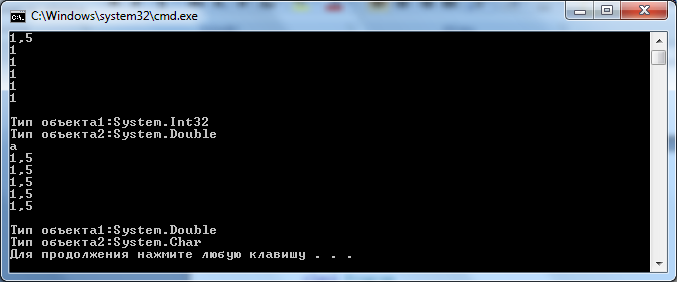
MyCollection<double,char> mcd = new MyCollection<double, char>(5, 1.5,'a');

mcd.Show();

mcd.PrintType();

}

}



Таким образом, обобщения автоматически обеспечивают типовую безопасность всех операций, затрагивающих параметризированный класс. В ходе выполнения этих операций обобщения исключают необходимость обращаться к приведению типов и проверять соответствие типов в коде вручную.

Когда для класса MyCollection указывается аргумент типа, например int или double, то создается **закрыто сконструированный тип**. Например, MyCollection <int, double> является закрыто сконструированным типом.

Обобщенный тип, такой как MyCollection <T, Е>, является абстракцией. И только после того, как будет сконструирован конкретный вариант, например MyCollection <int, double>, создается конкретный тип.

Конструкция, подобная MyCollection <T, Е>, называется **открыто сконструированным типом**, поскольку в ней указывается параметр типа Т.

Если в программе используются обобщения, то очень полезно, когда переменные обобщенных типов легко можно отличить от необобщенных.

**Рекомендации по именованию обобщенных типов:**

1. Имена обобщенных типов должны начинаться с буквы Т.
2. Если обобщенный тип может быть заменен любым классом, поскольку нет никаких специальных требований, и используется только один обобщенный тип, можно использовать имя Т :

public class List<T> { }

public class LinkedList<T> { }

1. Если к обобщенному типу предъявляются специальные требования (например, что тип должен реализовывать интерфейс или наследоваться от определенного класса), либо же используется два или более обобщенных типа в качестве параметров, то следует применять осмысленные имена типов:

public class SortedList<TKey, TValue> { }

#### Ограниченные типы.

Указывая параметр типа, можно наложить определенное ограничение на этот параметр. Это делается с помощью **оператора where** при указании параметра типа:

class имя\_класса<параметр\_типа> where параметр\_типа : ограничения { // ...},

ограничения указываются списком через запятую.

В С# предусмотрен ряд ограничений на типы данных:

* **Ограничение на базовый класс:** Требует наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса.
* **Ограничение на интерфейс:** Требует реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.
* **Ограничение на конструктор:** Требует предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().
* **Ограничение ссылочного типа:** Требует указывать аргумент ссылочного типа с помощью оператора class.
* **Ограничение типа значения:** Требует указывать аргумент типа значения с помощью оператора struct.

При применении ограничения new(), следует обратить внимание на три важных момента:

* Его можно использовать вместе с другими ограничениями, но последним по порядку.
* Ограничение new () позволяет конструировать объект, используя только конструктор без параметров, даже если доступны другие конструкторы. Иными словами, передавать аргументы конструктору параметра типа не разрешается.
* Ограничение new() нельзя использовать одновременно с ограничением типа значения.

Ограничения ссылочного типа и типа значения позволяют указать на то, что аргумент, обозначающий тип, должен быть либо ссылочного типа, либо типа значения. Эти ограничения оказываются полезными в тех случаях, когда для обобщенного кода важно провести различие между ссылочным типом и типом значения. Если имеются дополнительные ограничения, то class или struct должны быть первым по порядку накладываемым ограничением.

Как правило, пустое значение нельзя присвоить переменной типа значения. Следовательно, в отсутствие ограничения такое присваивание было бы недопустимым, и код не подлежал бы компиляции. Это один из тех случаев, когда для обобщенного кода может оказаться очень важным различие между типами значений и ссылочными типами

### Примеры использования обобщенных типов

#### Часть 1. Иерархия классов

|  |  |
| --- | --- |
|  | **class Person**  **{**  static string[] InitNames={"Иван Иванов", "Петр Петров ","Илья Кузнецов","Сергей Крылов","Артем Артемьев","Елена Смирнова","Елизавета Семенова","Екатерина Соколова","Евгения Павлова","Ксения Кошкина"};  protected static Random rnd=new Random();  public string Name { set; get; }  int age;  public int Age  {  get { return age; }  set { if (value > 0)age = value; else age = 17; }  }  public Person()  {  Name = "NoName";  Age = 17;  }  public Person(string Name, int Age)  {  this.Name = Name;  this.Age = Age;  }  public override string ToString()  {  return Name+", "+Age.ToString();  }  public virtual void Show()  {  Console.WriteLine(this.ToString());  }  public virtual void RandomInit()  {  Name=InitNames[rnd.Next(10)];  Age=rnd.Next(17,35);  }  **}**  **class Student:Person**  **{**  int rating;  public int Rating  {  set  {  if (value >= 0 && value <= 10) rating = value;  else rating = 0;  }  get  {  return rating;  }  }  public Student()  : base()  {  rating = 0;  }  public Student(string Name, int Age, int Rating)  : base(Name, Age)  {  this.Rating = rating;  }  public override string ToString()  {    return base.ToString()+", "+rating.ToString();  }  public override void Show()  {  Console.WriteLine( this.ToString());    }  public override void RandomInit()  {  base.RandomInit();  rating = Person.rnd.Next(1,11);  }  **}** |

#### Часть 2. Коллекция (массив типа Person), в которую можно поместить элементы типа Person и Student

class List

{

Person [] list;

int size;

int current;

public int Size

{

set

{

if (value < 0) value = 0;

size = value;

}

get

{

return size;

}

}

public int Current

{

set

{

if (value < 0) value = 0;

if (value >= size) value = size;

current = value;

}

get

{

return current;

}

}

public List()

{

size=10;

list=new Person[10];

current=0;

}

public void Add(Person obj)

{

if (current < Size)

{

list[current] = obj;

Current++;

}

else

{

Person[] temp = new Person[size + 10];

for (int i = 0; i < Size; i++) temp[i] = list[i];

temp[Size] = obj;

list = temp;

Size += 10;

Current++;

}

}

public void Show()

{

for (int i = 0; i < Current; i++)

Console.WriteLine(list[i].ToString());

}

}

**Демонстрационная программа:**

class Program

{

static void Main(string[] args)

{

List list = new List();

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

list.Add(s);

Person p = new Person();

p.RandomInit();

list.Add(p);

}

list.Show();

**//list.Add(20); ошибка: несоответствие типов**

**//list.Show();**

}



#### Часть 3. Коллекция (массив типа object), в которую можно поместить элементы любого типа.

class List

{

**object [] list;**

int size;

int current;

public int Size

{

set

{

if (value < 0) value = 0;

size = value;

}

get

{

return size;

}

}

public int Current

{

set

{

if (value < 0) value = 0;

if (value >= size) value = size;

current = value;

}

get

{

return current;

}

}

public List()

{

size=10;

list=new object[10];

current=0;

}

public void Add(object obj)

{

if (current < Size)

{

list[current] = obj;

Current++;

}

else

{

object [] temp = new object [size + 10];

for (int i = 0; i < Size; i++) temp[i] = list[i];

temp[Size] = obj;

list = temp;

Size += 10;

Current++;

}

}

public void Show()

{

for (int i = 0; i < Current; i++)

Console.WriteLine(list[i].ToString());

}

}

**Демонстрационная программа:**

static void Main(string[] args)

{

List list = new List();

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

list.Add(s);

Person p = new Person();

p.RandomInit();

list.Add(p);

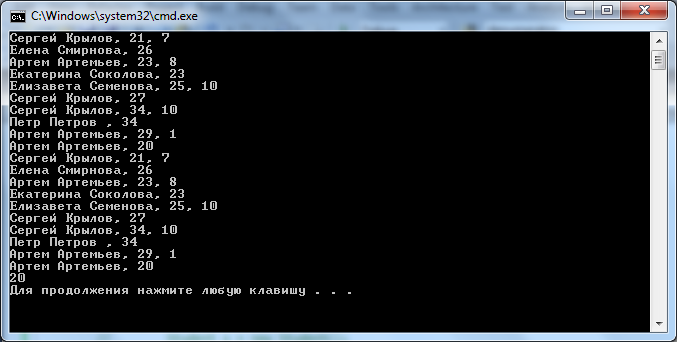
}

list.Show();

**list.Add(20);**

**list.Show();**

}



#### Часть 4. Параметризированная коллекция: тип массива является параметром

class Group<T>

{

**T[] list;**

int size;

int current;

public int Size

{

set

{

if (value < 0) value = 0;

size = value;

}

get

{

return size;

}

}

public int Current

{

set

{

if (value < 0) value = 0;

if (value >= size) value = size;

current = value;

}

get

{

return current;

}

}

public Group()

{

size=10;

**list=new T[10];**

current=0;

}

**public void Add(T obj)**

{

if (current < Size)

{

list[current] = obj;

Current++;

}

else

{

**T[] temp = new T[size + 10];**

for (int i = 0; i < Size; i++) temp[i] = list[i];

temp[Size] = obj;

list = temp;

Size += 10;

Current++;

}

}

public void Show()

{

for (int i = 0; i < Current; i++)

Console.WriteLine(list[i].ToString());

}

}

**Демонстрационная программа:**

static void Main(string[] args)

{

Group<Student> group = new Group<Student>();

for (int i = 0; i < 10; i++)

{

Student s = new Student();

s.RandomInit();

group.Add(s);

}

group.Show();

Group<int> intGr = new Group<int>();

for (int i = 0; i < 10; i++)

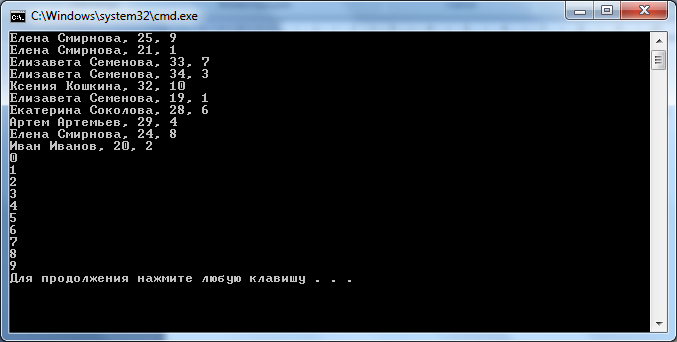
{

intGr.Add(i);

}

intGr.Show();

}



#### Часть 5. Параметризированная коллекция с ограничениями.

1. **Ограничение на базовый класс:** Требует наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса.

class Group<T> where T:Person

{...}

**Демонстрационная программа 1:**

static void Main(string[] args)

{

Group<Student> group = new Group<Student>();

for (int i = 0; i < 10; i++)

{

Student s = new Student();

s.RandomInit();

group.Add(s);

}

group.Show();

**// Group<int> intGr = new Group<int>();//синтаксическая ошибка**

// for (int i = 0; i < 10; i++)

// {

// intGr.Add(i);

// }

// intGr.Show();

}

**Демонстрационная программа 2:**

static void Main(string[] args)

{

Group<Student> group1 = new Group<Student>();

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

group1.Add(s);

//**Person p = new Person();**

**//p.RandomInit();**

**//group1.Add(p); //синтаксическая ошибка**

}

group1.Show();

Group<Person> group2 = new Group<Person>();

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

group2.Add(s);

Person p = new Person();

p.RandomInit();

group2.Add(p);

}

group2.Show();

}

1. **Ограничение на интерфейс:** Требует реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.

interface INameDialog

{

void PrintName();

void Init();

}

**class Student:Person, INameDialog**

{

. . . .

public void Init()

{

Console.WriteLine("Введите имя:");

Name = Console.ReadLine();

}

public void PrintName()

{

Console.WriteLine("Имя:"+nam);

}

}

class Group<T> where T: INameDialog

{

T[] list;

int size;

int current;

. . . .

}

**Демонстрационная программа:**

static void Main(string[] args)

{

Group<Student> group1 = new Group<Student>();

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

group1.Add(s);

//Person p = new Person();

//p.RandomInit();

//group1.Add(p);

}

group1.Show();

//Group<Person> group2 = new Group<Person>(); - ошибка!

//for (int i = 0; i < 5; i++)

//{

// Student s = new Student();

// s.RandomInit();

// group2.Add(s);

// Person p = new Person();

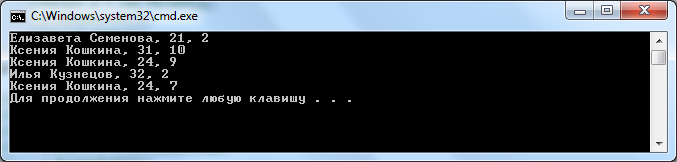
// p.RandomInit();

// group2.Add(p);

//}

//group2.Show();

}



1. **Ограничение на конструктор:** Требует предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().

class Student:Person, INameDialog

{

int rating;

public int Rating

{

set

{

if (value >= 0 && value <= 10) rating = value;

else rating = 0;

}

get

{

return rating;

}

}

//public Student()

// : base()

//{

// rating = 0;

//}

public Student(string Name, int Age, int Rating)

: base(Name, Age)

{

this.Rating = rating;

}

. . . . . .

}

class Group<T> where T: new()

{. . . . }

**Демонстрационная программа:**

static void Main(string[] args)

{

Group<Student> group1 = new Group<Student>(); //ошибка компиляции!!!

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

group1.Add(s);

}

group1.Show();

Group<Person> group2 = new Group<Person>();

for (int i = 0; i < 5; i++)

{

Person p = new Person();

p.RandomInit();

group2.Add(p);

}

group2.Show();

}

1. **Ограничение ссылочного типа:** Требует указывать аргумент ссылочного типа с помощью оператора class.
2. class Group<T> where T: class

{. . . }

**Демонстрационная программа:**

class Program

{

static void Main(string[] args)

{

Random rnd=new Random();

Group<int> group3 = new Group<int>(); //ошибка компиляции

for (int i = 0; i < 5; i++)

{

int number = rnd.Next(1,10);

group3.Add(number);

}

group3.Show();

}

1. **Ограничение типа значения**: Требует указывать аргумент типа значения с помощью оператора struct.

class Group<T> where T: struct

{. . . }

**Демонстрационная программа:**

class Program

{

static void Main(string[] args)

{

Group<Student> group1 = new Group<Student>(); //ошибка компиляции

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

group1.Add(s);

}

group1.Show();

}

### Иерархия обобщенных классов

Обобщенные классы могут входить в иерархию классов аналогично необобщенным классам. Следовательно, обобщенный класс может действовать как базовый или производный класс. Главное отличие между иерархиями обобщенных и необобщенных классов заключается в том, что в случае использования обобщенных классов аргументы типа, необходимые обобщенному базовому классу, должны передаваться всеми производными классами вверх по иерархии аналогично передаче аргументов конструктора.

В производном классе следует непременно указывать параметры типа, требующиеся его обобщенному базовому классу, даже если этот производный класс не обязательно должен быть обобщенным. В производный класс можно добавлять новые параметры типа, если в этом есть необходимость.

**Пример 15\_3**

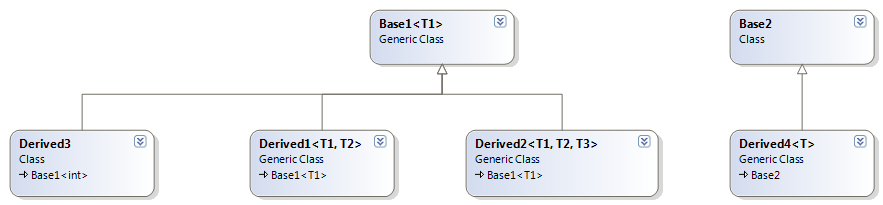


Рисунок . Пример создания сложной иерархии обобщенных классов:

#### Пример программы для диаграммы классов на рисунке 1.

namespace ieraehia

{

class Base1<T1>

{

protected T1 arg1;

public Base1(T1 x)

{

arg1 = x;

}

virtual public void Show()

{

Console.WriteLine("arg1=" + arg1);

Console.WriteLine("type=" + typeof(T1));

}

}

class Derived1<T1, T2> : Base1<T1>

{

protected T2 arg2;

public Derived1(T1 x, T2 y)

: base(x)

{

arg2 = y;

}

override public void Show()

{

base.Show();

Console.WriteLine("arg2=" + arg2);

Console.WriteLine("type=" + typeof(T2));

}

}

class Derived2<T1, T2, T3> : Base1<T1>

{

protected T2 arg2;

protected T3 arg3;

public Derived2(T1 x, T2 y, T3 z)

: base(x)

{

arg2 = y;

arg3 = z;

}

override public void Show()

{

base.Show();

Console.WriteLine("arg2=" + arg2);

Console.WriteLine("type=" + typeof(T2));

Console.WriteLine("arg3=" + arg3);

Console.WriteLine("type=" + typeof(T3));

}

}

class Derived3:Base1<int>

{

protected int arg;

public Derived3(int x,int y):base(x)

{

arg = y;

}

override public void Show()

{

base.Show();

Console.WriteLine("arg=" + arg);

}

}

class Base2

{

protected int arg;

public Base2(int a)

{

arg=a;

}

virtual public void Show()

{

Console.WriteLine("arg=" + arg);

}

}

class Derived4<T>:Base2

{

protected T arg2;

public Derived4( T x, int a):base(a)

{

arg2=x;

}

override public void Show()

{

base.Show();

Console.WriteLine("arg2=" + arg2);

Console.WriteLine("type=" + typeof(T));

}

}

class Program

{

static void Main(string[] args)

{

Base1<int> obj1 = new Base1<int>(1);

Derived1<char, double> obj2=new Derived1<char,double>('a',1.1);

Derived2<int, char, double> obj3 = new Derived2<int, char, double>(1, 'a', 1.1);

Derived3 obj4 = new Derived3(0,100);

Console.WriteLine("Base1<int> obj1:");

obj1.Show();

Console.WriteLine("Derived1<char, double> obj2:");

obj2.Show();

Console.WriteLine(" Derived2<int, char, double> obj3:");

obj3.Show();

Console.WriteLine(" Derived3 obj4:");

obj4.Show();

Base2 obj5 = new Base2(5);

Derived4<int> obj6 = new Derived4<int>(10, 20);

Console.WriteLine(" Base2 obj5 :");

obj5.Show();

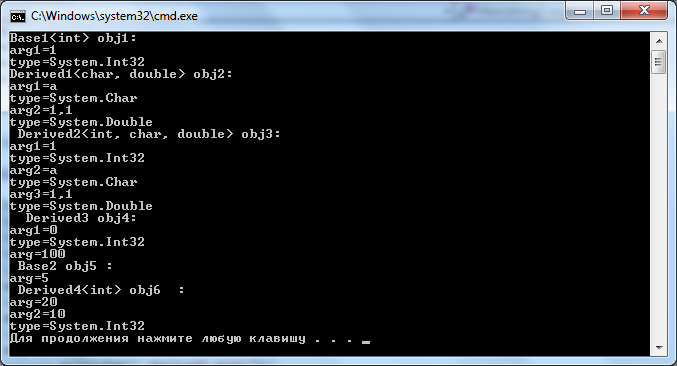
Console.WriteLine(" Derived4<int> obj6 :");

obj6.Show();

}

}

}



### Значения по умолчанию.

При написании обобщенного кода иногда бывает необходимо провести различие между типами значений и ссылочными типами. Такая потребность возникает, если переменной параметра типа должно быть присвоено значение по умолчанию. Для ссылочных типов (class) значением по умолчанию является null, для неструктурных типов значений – 0 или логическое значение false, если это тип bool, а для структур (struct) –объект соответствующей структуры с полями, установленными по умолчанию.

Например, если в следующем объявлении класса Test:

class Test<T>

{

Т obj ;

// инициализируем obj

. . .

}

**Какое значение требуется присвоить переменной obj по умолчанию?**

1. obj = null; // подходит только для ссылочных типов
2. obj =0; // подходит только для числовых типов,

// перечислений, но не для структур

Для решения этой проблемы используется **оператор default**:

*default(тип)*

Эта форма оператора default пригодна для всех аргументов типа ( и для значений или для ссылочных типов).

Пример 15\_6:

class Group<T, E> where T: class

{

T[] list;

int size;

int current;

**public E param;**

. . .

public Group()

{

size=10;

list=new T[10];

current=0;

param = default(E);

}

. . .

}

class Program

{

static void Main(string[] args)

{

Group<Student, int> group1 = new Group<Student, int>();

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

group1.Add(s);

}

group1.Show();

if (group1.param == null) Console.WriteLine("param=null");

if (group1.param == 0) Console.WriteLine("param=0");

Group<Person, string> group2 = new Group<Person, string>();

for (int i = 0; i < 5; i++)

{

Person p = new Person();

p.RandomInit();

group2.Add(p);

}

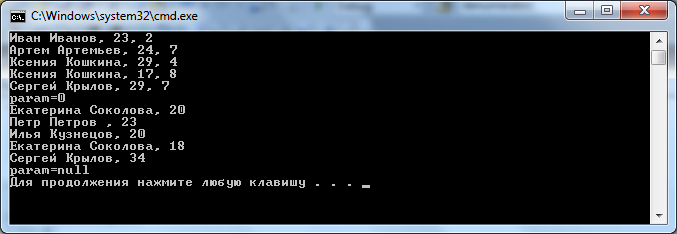
group2.Show();

if (group2.param == null) Console.WriteLine("param=null");

if (group2.param == "") Console.WriteLine("param=0");

}

}



### Статические члены обобщенных классов

Статические члены обобщенного класса разделяются только одним экземпляром класса. Рассмотрим пример 15\_4.

class Group<T> where T:class

{

**static int count = 0;**

T[] list;

int size;

int current;

. . .

public static int GetCount()

{

return count;

}

public Group()

{

size = 10;

list = new T[10];

current = 0;

}

public void Add(T obj)

{

if (current < Size)

{

list[current] = obj;

Current++;

**count++;**

}

else

{

T[] temp = new T[size + 10];

for (int i = 0; i < Size; i++) temp[i] = list[i];

temp[Size] = obj;

list = temp;

Size += 10;

Current++;

**count++;**

}

}

}

class Program

{

static void Main(string[] args)

{

Group<Student> group1 = new Group<Student>();

for (int i = 0; i < 5; i++)

{

Student s = new Student();

s.RandomInit();

group1.Add(s);

}

group1.Show();

Console.WriteLine(Group<Student>.GetCount());

Group<Person> group2 = new Group<Person>();

for (int i = 0; i < 5; i++)

{

Person p = new Person();

p.RandomInit();

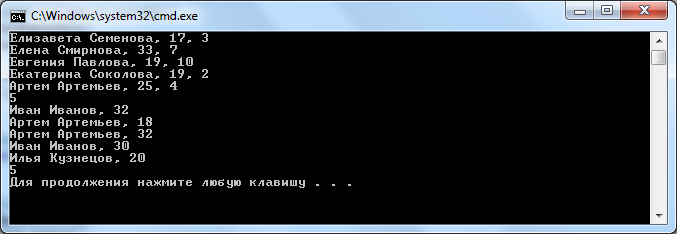
group2.Add(p);

}

group2.Show();

Console.WriteLine(Group<Person>.GetCount());

}

****

### Обобщенные методы

Построение обобщенных методов представляет собой более развитую версию традиционной перегрузки методов. **Перегрузка** — это определение нескольких версий одного метода, отличающихся друг от друга количеством или типами параметров.

Рассмотрим метод для обмена двух переменных.

Пример 15\_5

static void Swap<T>(ref T p1, ref T p2)

{

T temp = p1;

p1 = p2;

p2 = temp;

}

Использование метода:

Person p1 = group2[0];

Person p2 = group2[1];

Console.WriteLine("\nОбмен элементов местами");

p1.Show();

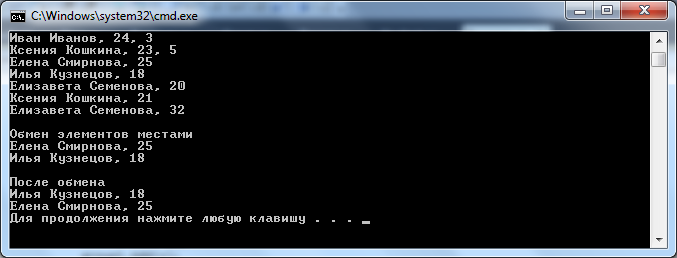
p2.Show();

Swap<Person>(ref p1, ref p2);

Console.WriteLine("\nПосле обмена");

p1.Show();

p2.Show();



### Параметризированные коллекции

Во вторую версию библиотеки .NET добавлены параметризованные коллекции для представления основных структур данных, применяющихся при создании программ, — стека, очереди, списка, словаря и т. д. Эти коллекции, расположенные в пространстве имен **System.Collections.Generic**, дублируют аналогичные коллекции пространства имен **System.Collections.**

**Параметризованные коллекции библиотеки .NET**

|  |  |
| --- | --- |
| **Класс- прототип** | **Обычный класс** |
| Comparer <T> | Comparer |
| Dictionary <K,T> | HashTable |
| LinkedList <T>//двунаправленный список | - |
| List <T> | ArrayList |
| Queue<T> | Queue |
| SortedDictionary <K,T> | SortedList |
| Stack <T> | Stack |

Параметром класса-прототипа является тип данных, с которым он работает (T – это тип, который является параметром коллекции, т. е. вместо него можно подставить любой другой тип данных).

В коллекциях, которые мы рассматривали ранее, хранятся ссылки на объекты типа object. Когда значение структурного типа приводится к типу object (или к типу того интерфейса, который реализован системой) выполняется операция упаковки – явного преобразования из типа значений в тип ссылок. Эта операция выполняется автоматически и не требует вмешательства программиста. Обратной операцией является распаковка – значение объекта присваивается переменной.

У таких коллекций есть два недостатка:

* в одной и той же коллекции можно хранить элементы любого типа, следовательно, ошибки при помещении в коллекцию невозможно проконтролировать на этапе компиляции, а при извлечении элемента требуется его явное преобразование;
* при хранении в коллекции элементов значимых типов выполняется большой объем действий по упаковке и распаковке элементов, что в значительной степени снижает эффективность работы.

Пример 1.

class Person : IComparable

{. . . . }

class Student : Person

{. . . . }

class Program

{

static void Main(string[] args)

{

List<Person> StudentGroup = new List<Person>();//создали коллекцию

StudentGroup.Add(new Student("Иванов",18,1,4));

StudentGroup.Add(new Student("Петров",19,2,5));

StudentGroup.Add(new Student("Сидоров",21,5,4.5));

StudentGroup.Add(new Person("Семенов",23));

//StudentGroup.Add(1);

Console.WriteLine("Студенческая группа");

foreach (Person x in StudentGroup)

x.Show();

List<int> Vector = new List<int>();

Vector.Add(1);

Vector.Add(5);

Vector.Add(4);

Console.WriteLine("\nВектор");

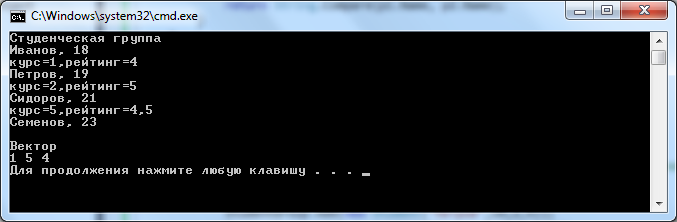
foreach (int y in Vector)

Console.Write(y + " ");

Console.WriteLine();

}

}



Коллекция StudentGroup содержит объекты пользовательских классов иерархии классов Person->Student.

В коллекции, для которой объявлен тип элементов Person, благодаря полиморфизму можно хранить элементы любого производного класса, но не элементы других типов.

Казалось бы, по сравнению с обычными коллекциями это ограничение, а не универсальность, однако на практике коллекции, в которых действительно требуется хранить значения различных, не связанных межу собой типов, почти не используются.

Достоинством же такого ограничения является то, что компилятор может выполнить контроль типов во время компиляции, а не выполнения программы, что повышает ее надежность и упрощает поиск ошибок.

Коллекция Vector состоит из целых чисел, причем для работы с ними не требуются ни операции упаковки и распаковки, ни явные преобразования типа при получении элемента из коллекции, как это было в обычных коллекциях.

В пространстве имен System.Collections.Generic также описаны параметризованные интерфейсы.

**Параметризованные интерфейсы библиотеки .NET.**

|  |  |
| --- | --- |
| Параметризованный интерфейс | Обычный интерфейс |
| ICollection<T> | ICollection |
| IComparable<T> | IComparable |
| IDictionary<Т> | IDictionary |
| IEnumerable<T> | I Enumerable |
| IEnumerator<T> | I Enumerator |
| IList<T> | IList |