Лекция 35

Обобщенные методы и классы

Обобщенные методы и классы

Термин **обобщение**, по существу, означает **параметризированный тип**. С помощью обобщений можно, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных.

Класс, **структура**, **интерфейс**, **метод** или **делегат**, оперирующий параметризированным типом данных, называется **обобщенным**.

Обобщения 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.

Обобщенные методы и классы

Общая форма объявления обобщенного класса:

```
class имя_класса<список_параметров_типа> {
    // ...
}
```

Синтаксис объявления ссылки на обобщенный класс:

```
имя_класса<список_аргументов_типа>
   имя_переменной =
new имя_класса<список_параметров_типа>
   (список_аргументов_конструктора);
```

Ограничения при использовании обобщений:

- Свойства, операторы, индексаторы и события не могут быть обобщенными. Но эти элементы могут использоваться в обобщенном классе, причем с параметрами обобщенного типа этого класса.
- К обобщенному методу нельзя применять модификатор **extern**.
- Типы указателей нельзя использовать в аргументах типа.
- Если обобщенный класс содержит поле типа **static**, то во всех экземплярах объектов одного конструируемого типа совместно используется одно и то же поле типа **static**. Но в экземплярах объектов другого конструируемого типа совместно используется другая копия этого поля. Следовательно, поле типа **static** не может совместно использоваться объектами всех конструируемых типов.

Пример 1

```
class Gen<T> {  // Обобщенный класс
    T ob; // Объявить переменную типа T
    public Gen(T o) { // У конструктора параметр типа T
        ob = o;
    }
    public T GetOb() {
       return ob;    // Возвратить переменную
                  // экземпляра ob, которая относится к типу T
    }
    public void ShowType() {
        Console.WriteLine("К типу T относится " +
   typeof(T)); // показать тип T
    }
}
```

Пример 1

```
class GenericsDemo {
    static void Main() {
        Gen<int> iOb; // Создать переменную ссылки
                      // на объект Gen типа int
        // Создать объект типа Gen<int>
        // и присвоить ссылку на него переменной iOb
        iOb = new Gen<int>(102);
        iOb.ShowType();  // Показать тип данных,
                         // хранящихся в переменной iOb
        int v = iOb.GetOb();   // Получить значение
                               // переменной iOb
        Console.WriteLine("Значение: " + v);
        Console.WriteLine();
```

Пример 1

```
        // Создать объект типа Gen для строк
        Gen<string> strOb = new Gen<string>("Обобщения
    повышают эффективность.");
        strOb.ShowType();   // Показать тип данных,
                            // хранящихся в переменной strOb
        string str = strOb.GetOb();   // Получить значение
                                      // переменной strOb
       Console.WriteLine("Значение: " + str);
    }
}
```

К типу T относится System.Int32
Значение: 102
К типу T относится System.String
Значение: Обобщения повышают эффективность.

Обобщенные методы и классы

Когда для обобщенного класса, например **Gen**, указывается аргумент конкретного типа, например **int** или **string**, то создается так называемый в **C# закрыто сконструированный тип**, например, **Gen<int>**.

А конструкция, подобная **Gen<T>**, называется в **C# открыто сконструированным типом**, поскольку в ней указывается параметр типа **T**, но не такой конкретный тип, как **int**.

Пример 2

```
class TwoGen<T, V> {
        T ob1;        V ob2;
        public TwoGen(T o1, V o2) {
            ob1 = o1;            ob2 = o2;
        }
        public void showTypes() {   // Показать типы T и V
            Console.WriteLine("К типу T относится " +
    typeof(T));
            Console.WriteLine("К типу V относится " +
    typeof(V));
        }
        public T GetOb1() { return ob1; }
        public V GetObj2() { return ob2; }
}
```

Пример 2

```
class SimpGen {
   static void Main() {
     TwoGen<int, string> tgObj =  new TwoGen<int,
   string>(119, "Альфа Бета Гамма");
     tgObj.showTypes();       // Показать типы
     int v = tgObj.GetOb1(); // Получить и
                                // вывести значения
     Console.WriteLine("Значение: " + v);
     string str = tgObj.GetObj2();
     Console.WriteLine("Значение: " + str);
   }
}
```

Ограничение типов

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 : ограничение1,
   ограничение2,... {
 // ...
}
```

где ограничения указываются списком через запятую.

Ограничение типов

Ряд ограничений на типы данных:

- Ограничение на базовый класс, требующее наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является неприкрытое ограничение типа, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.
- Ограничение на интерфейс, требующее реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.

Ограничение типов

Ряд ограничений на типы данных (продолжение):

- Ограничение на конструктор, требующее предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора **new**().
- Ограничение ссылочного типа, требующее указывать аргумент ссылочного типа с помощью оператора **class**.
- Ограничение типа значения, требующее указывать аргумент типа значения с помощью оператора **struct**.

Ограничение типов

Определять ограничения можно как для одного любого, так и для всех требуемых обобщенным классом ограничений с использованием нескольких операторов **where**:

class MyGenericClass<Tl, T2> **where** Tl : ограничение1 **where** T2 : ограничение2 { }

Любое из применяемых ограничений должно обязательно идти после указателей наследования:

class MyGenericClass<Tl, T2> : MyBaseClass, IMyInterface **where** T1 : ограничение1 **where** T2 : ограничение2 { }

Ограничение типов

В случае применения ограничения **new** (), оно должно обязательно указываться для типа последним.

Можно применять параметр одного типа в качестве ограничения для другого, как показано ниже:

class MyGenericClass<T1, T2> **where** T2 : T1 { }

Здесь **T2** должен обязательно представлять собой тот же тип, что и **T1**, или наследоваться от **T1**. Такое ограничение называется простым ограничением типа и означает, что параметр одного обобщенного типа применяется в качестве ограничения для другого.

Циклические ограничения типов **запрещены**:

class MyGenericClass<T1, T2>
where T2 : T1 **where** T1 : T2 { }

Этот код компилироваться не будет.

Ограничение на базовый класс

Ограничение на базовый класс позволяет указывать базовый класс, который должен наследоваться аргументом типа.

where T : имя_базового_класса

где T обозначает имя параметра типа, а имя_базового_класса — конкретное имя ограничиваемого базового класса.

Одновременно в этой форме ограничения может быть указан только один базовый класс.

class Test<T> **where** T : A {…

Оператор **where** в этом объявлении накладывает следующее ограничение: любой аргумент, указываемый для типа **T**, должен иметь класс **A** в качестве базового.

Пример 3

```
class A {
    public void Hello () {
        Console.WriteLine("Hello");
    }
}
class B : A { } // Класс B наследует класс A
class C { } // Класс C не наследует класс A
class Test<T> where T : A {
        T obj;
        public Test(T o) {
            obj = o;
        }
```

Пример 3

```
        public void SayHello() { obj. Hello(); }
}
class Demo {
    static void Main() {
        A a = new A();
        B b = new B();
        C c = new C();
     // Следующий код вполне допустим,
     // поскольку класс A указан как базовый
        Test<A> t1 = new Test<A>(a);
         t1.SayHello();
```

Пример 3

```
    // Следующий код допустим,
    // поскольку класс B наследует от класса A
        Test<B> t2 = new Test<B>(b);
        t2.SayHello();
    // Следующий код недопустим,
    // поскольку класс C не наследует от класса A
        // Test<C> t3 = new Test<C>(c); // Ошибка!
        // t3.SayHello(); // Ошибка!
    }
}
```

Ограничение на базовый класс

Два последствия наложения ограничения на базовый класс:

во-первых, это ограничение разрешает доступ к членам базового класса из обобщенного класса;

во-вторых, оно гарантирует допустимость только тех аргументов типа, которые удовлетворяют данному ограничению, обеспечивая тем самым типовую безопасность.

Пример 4

Программа управляет отдельными списками для друзей, поставщиков, клиентов и т.д.

```
using System;
class NotFoundException : Exception {
   public NotFoundException() : base() { }
   public NotFoundException(string str) : base(str){}
   public NotFoundException(string str, Exception
    inner) : base(str, inner) { }
   protected NotFoundException
    (System.Runtime.Serialization.SerializationInfo si,
System.Runtime.Serialization.StreamingContext sc) :
   base (si, sc) { }
}
```

Пример 4

```
// Базовый класс, в котором хранятся
// имя абонента и номер его телефона
class PhoneNumber {
    public PhoneNumber(string n, string num) {
        Name = n;
        Number = num;
    }
     // Автоматически реализуемые свойства, в которых
     // хранятся имя абонента и номер его телефона
    public string Number { get; set; }
    public string Name { get; set; }
}
```

Пример 4

```
class Friend : PhoneNumber {// Класс для тел. номеров друзей
    public Friend(string n, string num, bool wk) : base (n,
   num) {   IsWorkNumber = wk;  }
    public bool IsWorkNumber { get; private set; }
        // ...
}
class Supplier : PhoneNumber {// Класс для тел. поставщиков
    public Supplier(string n, string num) : base (n, num) { }
         // ...
}
// Этот класс не наследует от класса PhoneNumber
class EmailFriend {
     // ...
}
```

Пример 4

```
// Класс PhoneList способен управлять любым видом
// списка телефонных номеров, при условии,
// что он является производным от класса PhoneNumber
class PhoneList<T> where T : PhoneNumber {
    T[] phList;      int end;
    public PhoneList() {
        phList = new T[10];          end = 0;
    }
  public bool Add(T newEntry) {// Добавить элемент в список
        if (end == 10) return false;
        phList[end] = newEntry;
        end++;
        return true;
    }
```

Пример 4

```
// Найти и возвратить сведения о телефоне
// по заданному имени
 public T FindByName(string name) {
    for (int i=0; i<end; i++) {
     // Имя может использоваться, т. к. относится
     // к базовым членам класса по ограничению
         if (phList[i].Name == name)
             return phList[i];
    }
    // Имя отсутствует в списке
    throw new NotFoundException();
}
```

Пример 4

```
        // Найти и возвратить сведения о телефоне
        // по заданному номеру
       public T FindByNumber(string number) {
          for(int i=0; i<end; i++) {
 // Номер телефона также может использоваться, поскольку
// его свойство Number относится к членам класса PhoneNumber,
 // который является базовым по накладываемому ограничению
              if (phList[i].Number == number)
                  return phList[i];
          }
          throw new NotFoundException(); // Нет номера
      }
      // ...
}
```

Пример 4

```
class UseBaseClassConstraint {
    static void Main() {
        // Код допустим, т. к. класс Friend наследует от класса PhoneNumber
        PhoneList<Friend> plist = new PhoneList<Friend>();
        plist.Add(new Friend("Том", "555-1234", true));
        plist.Add(new Friend("Гари", "555-6756", true));
        plist.Add(new Friend("Матт", "555-9254", false));
        try {
    // Найти номер телефона по заданному имени друга
            Friend frnd = plist.FindByName("Гари");
            Console.Write(frnd.Name + ": " + frnd.Number);
            if (frnd.IsWorkNumber) Console.WriteLine(" (рабочий)");
            else   Console.WriteLine();
        } catch(NotFoundException) {
             Console.WriteLine("Не найдено");
        }
```

Пример 4

```
Console.WriteLine();
// Следующий код допустим, поскольку класс
// Supplier наследует от класса PhoneNumber
 PhoneList<Supplier> plist2 =
      new PhoneList<Supplier>();
 plist2.Add(new Supplier("Фирма Global Hardware",
"555-8834"));
 plist2.Add(new Supplier("Агентство Computer
Warehouse", "555-9256"));
 plist2.Add(new Supplier("Компания NetworkCity",
"555-2564"));
```

Пример 4

```
        try {
// Найти поставщика по заданному номеру телефона
            Supplier sp = plist2.FindByNumber("555-2564");
            Console.WriteLine(sp.Name + ": " + sp.Number);
        } catch(NotFoundException) {
            Console.WriteLine("Не найдено");
        }
 // Следующее объявление недопустимо, поскольку
 // класс EmailFriend НЕ наследует от класса PhoneNumber
    // PhoneList<EmailFriend> plist3 =
            new PhoneList<EmailFriend>(); // Ошибка!
    }
}
```

Ограничение на интерфейс

Ограничение на интерфейс позволяет указывать интерфейс, который должен быть реализован аргументом типа. Это ограничение служит тем же основным целям, что и ограничение на базовый класс. Во-первых, оно позволяет использовать члены интерфейса в обобщенном классе. И во-вторых, оно гарантирует использование только тех аргументов типа, которые реализуют указанный интерфейс.

where T : имя_интерфейса

В этой форме ограничения может быть указан список интерфейсов через запятую. Если ограничение накладывается одновременно на базовый класс и интерфейс, то первым в списке должен быть указан базовый класс.

Пример 5

Программа управляет отдельными списками для друзей, поставщиков, клиентов и т.д.

```
using System;
class NotFoundException : Exception {
   public NotFoundException() : base() { }
   public NotFoundException(string str) : base(str){}
   public NotFoundException(string str, Exception
    inner) : base(str, inner) { }
   protected NotFoundException
    (System.Runtime.Serialization.SerializationInfo si,
System.Runtime.Serialization.StreamingContext sc) :
   base (si, sc) { }
}
```

Пример 5

```
public interface IPhoneNumber { // Интерфейс
    string Number { get;  set; } // Имя и номер телефона
    string Name { get;  set; } // Номер телефона
}
class Friend : IPhoneNumber { // Класс для тел. друзей
        public Friend(string n, string num, bool wk) {
           Name = n;  Number = num;   IsWorkNumber = wk;
        }
        public bool IsWorkNumber { get; private set; }
        // Реализовать интерфейс IPhoneNumber
         public string Number { get; set; }
         public string Name { get; set; }
        // ...
}
```

Пример 5

```
// Класс для телефонных номеров поставщиков
class Supplier : IPhoneNumber {
    public Supplier(string n, string num) {
        Name = n;          Number = num;
    }
    // Реализовать интерфейс IPhoneNumber
    public string Number { get; set; }
    public string Name { get; set; }
     // ...
}
// В этом классе интерфейс IPhoneNumber не реализуется
class EmailFriend {
    // ...
}
```

Пример 5

```
// Класс PhoneList способен управлять любым видом списка телефонных
 // номеров, при условии, что он реализует интерфейс PhoneNumber
class PhoneList<T> where T : IPhoneNumber {
    T[] phList;      int end;
    public PhoneList() {
       phList = new T[10];
       end = 0;
    }
    public bool Add(T newEntry) {
        if (end == 10) return false;
        phList[end] = newEntry;
        end++;
        return true;
    }
```

Пример 5

```
// Найти и возвратить сведения
// о телефоне по заданному имени
public T FindByName(string name) {
  for (int i=0; i<end; i++) {
// Имя может использоваться, потому что его свойство
// Name относится кчленам интерфейса IPhoneNumber,
// на который накладывается ограничение
      if (phList[i].Name == name)   return phList[i];
  }
   // Имя отсутствует в списке
  throw new NotFoundException();
}
```

Пример 5

```
// Найти и возвратить сведения о телефоне по заданному номеру
    public T FindByNumber(string number) {
        for (int i=0; i<end; i++) {
 // Номер телефона может использоваться, поскольку его
 // свойство Number относится к членам интерфейса
 // IPhoneNumber, на который накладывается ограничение
            if (phList[i].Number == number)
                return phList[i];
        }
        // Номер телефона отсутствует в списке
        throw new NotFoundException();
    }
    //...
}
```

Пример 5

```
class UseInterfaceConstraint {
    static void Main() {
      // Следующий код допустим, поскольку
      // в классе Friend реализуется интерфейс
   IPhoneNumber
      PhoneList<Friend> plist = new PhoneList<Friend>();
      plist.Add(new Friend("Том", 55-1234", true));
      plist.Add(new Friend("Гари", 55-6756", true));
      plist.Add(new Friend("Матт", 55-9254", false));
     // Найти номер телефона по заданному имени друга
      try {
         Friend frnd = plist.FindByName("Гари");
         Console.Write(frnd.Name + ": " + frnd.Number);
```

Пример 5

```
        if (frnd.IsWorkNumber)
                Console.WriteLine(" (рабочий)");
        else  Console.WriteLine();
    } catch(NotFoundException) {
        Console.WriteLine("Не найдено");
    }
     Console.WriteLine();
// Следующий код также допустим, поскольку в классе
// Supplier также реализуется интерфейс IPhoneNumber
    PhoneList<Supplier> plist2 = new PhoneList<Supplier>();
    plist2.Add(new Supplier("Фирма Global Hardware",
  "555-8834"));
    plist2.Add(new Supplier("Агентство Computer Warehouse",
  "555-9256"));
```

Пример 5

```
        plist2.Add(new Supplier ("Компания NetworkCity",
    "555-2564"));
        try {  // Найти поставщика по заданному номеру телефона
            Supplier sp = plist2.FindByNumber("555-2564");
            Console.WriteLine(sp.Name + ": " + sp.Number);
        } catch(NotFoundException) {
            Console.WriteLine("Не найдено");
        }
// Следующее объявление недопустимо, поскольку
// в классе EmailFriend НЕ реализуется интерфейс IPhoneNumber
        // PhoneList<EmailFriend> plist3 = new
                PhoneList<EmailFriend>(); // Ошибка!
    }
}
```

Ограничение **new** ()

Ограничение **new** () **на конструктор** позволяет получать экземпляр объекта обобщенного типа.

Как правило, создать экземпляр параметра обобщенного типа не удается. Но это положение изменяет ограничение **new** (), поскольку оно требует, чтобы аргумент типа предоставил **конструктор без параметров**. Им может быть конструктор, вызываемый по умолчанию и предоставляемый автоматически, или же конструктор без параметров явно определяемый пользователем (если в классе определены другие конструкторы). Накладывая ограничение **new** (), можно вызывать конструктор без параметров для создания объекта.

Пример 6

```
class MyClass {
    public MyClass() {  ...  }      //...
}
class Test<T> where T : new() {     // Ограничение
        T obj;
        public Test() { // Конструктор без параметров
            obj = new T();          //  Создать объект типа T
       }
       // ...
}
class ConsConstraintDemo {
    static void Main() {
            Test<MyClass> x = new Test<MyClass>();
    }
}
```

Ограничение **new** ()

При применении ограничения **new** () следует обратить внимание на **три важных момента**.

Во-первых, его можно использовать вместе с другими ограничениями, но последним по порядку.

Во-вторых, ограничение **new** () позволяет конструировать объект, используя только конструктор без параметров, — даже если доступны другие конструкторы.

И **в-третьих**, ограничение **new** () нельзя использовать одновременно с ограничением типа значения.

Ограничения ссылочного типа и типа значения

Ограничения ссылочного типа и типа значения позволяют указать на то, что аргумент, обозначающий тип, должен быть либо ссылочного типа, либо типа значения.

where T : **class**

Ключевое слово **class** указывает на то, что аргумент **T** должен быть ссылочного типа.

where T : **struct**

Ключевое слово **struct** указывает на то, что аргумент **T** должен быть типа значения (структуры относятся к типам значений).

Если имеются дополнительные ограничения, то в любом случае **class** или **struct** должно быть первым по порядку накладываемым ограничением.

Пример 7

```
using System;
class MyClass {
    //...
}
// Наложить ограничение ссылочного типа
class Test<T> where T : class {
     T obj;
     public Test() {
// Следующий оператор допустим только потому,
// что аргумент T относится к ссылочному типу
          obj = null;
    }
    // ...
}
```

Пример 7

```
class ClassConstraintDemo {
    static void Main() {
        // Следующий код вполне допустим,
        // поскольку MyClass является классом
        Test<MyClass> x = new Test<MyClass>();
        // Следующая строка кода содержит ошибку,
        // поскольку int относится к типу значения
        // Test<int> y = new Test<int>(); // Ошибка
    }
}
```

Пример 8

```
using System;
struct MyStruct {
      //...
}
class MyClass {
      // ...
}
class Test<T> where T : struct {
       T obj;
       public Test(T x) {
             obj = x;
       }
       // ...
}
```

Пример 8

```
class ValueConstraintDemo {
        static void Main() {
          // Оба следующих объявления вполне допустимы
            Test<MyStruct> x =
                    new Test<MyStruct>(new MyStruct());
             Test<int> y = new Test<int> (10);
           // А следующее объявление недопустимо!
           // Test<MyClass> z =
                    new Test<MyClass>(new MyClass());
        }
}
```

Неприкрытое ограничение типа

Существует разновидность ограничения на базовый класс, позволяющая установить связь между двумя параметрами типа.

Например, в объявлении обобщенного класса

class Gen<T, V> **where** V : T {

оператор **where** уведомляет компилятор о том, что аргумент типа, привязанный к параметру типа **V**, должен быть таким же, как и аргумент типа, привязанный к параметру типа **T**, или же наследовать от него. Если подобная связь отсутствует при объявлении объекта типа **Gen**, то во время компиляции возникнет ошибка. Такое ограничение на параметр типа называется **неприкрытым ограничением типа**.

Пример 9

```
using System;
class A {
        //...
}
class B : A {
      // ...
}

// Здесь параметр типа V должен наследовать
// от параметра типа T
class Gen<T, V> where V : T {
         // ...
}
```

Пример 9

```
class NakedConstraintDemo {
    static void Main() {
        // Это объявление вполне допустимо, поскольку
        // класс B наследует от класса A
            Gen<A, B> x = new Gen<A, B> ();
        // A это объявление недопустимо, поскольку
        // класс A не наследует от класса B
        // Gen<B, A> y = new Gen<B, A>(); // Ошибка
    }
}
```

Ограничения типов

С параметром типа может быть связано **несколько ограничений**. В этом случае ограничения указываются списком через запятую. В этом списке первым должно быть указано ограничение **class** либо **struct**, если оно присутствует, или же ограничение на базовый класс, если оно накладывается. Указывать ограничения **class** или **struct** одновременно с ограничением на базовый класс **не разрешается**. Далее по списку должно следовать ограничение на интерфейс, а последним по порядку — ограничение **new** ().

Ограничения типов

Например, следующее объявление считается допустимым:

```
class Gen<T> where T : MyClass, IMyInterface, new() {
    // ...
```

В данном случае параметр типа **T** должен быть заменен аргументом типа, наследующим от класса **MyClass**, реализующим интерфейс **IMyInterface** и использующим конструктор без параметра.

Если же в обобщении используются два или более параметра типа, то ограничения на каждый из них накладываются с помощью отдельного оператора **where**, которые отделяются друг от друга только пробелом.

Пример 10

```
using System;
// На два параметра типа класса Gen накладываются
// ограничения с помощью отдельных операторов where
class Gen<T, V> where T : class where V : struct {
      T ob1;
      V ob2;
      public Gen(T t, V v) {
            ob1 = t;
            ob2 = v;
      }
}
```

Пример 10

```
class MultipleConstraintDemo {
    static void Main() {
    // Эта строка кода допустима, поскольку
    // string — это ссылочный тип, а int — тип значения
        Gen<string, int> obj =
            new Gen<string, int> ("тест", 11);
  // А следующая строка кода недопустима, поскольку
  // bool не относится к ссылочному типу
     // Gen<bool, int> obj = new Gen<bool, int>(true, 11);
    }
}
```

Значение по умолчанию

Например, если в следующем объявлении класса Test:

```
class Test<T> {
    T obj;
  // ...
```

переменной **obj** требуется присвоить значение по умолчанию, то какой из двух вариантов следует выбрать?

```
obj = null; // подходит только для ссылочных типов
```

или

```
obj =0; // подходит только для числовых типов и
// перечислений, но не для структур
```

default (тип)

Может использоваться для всех аргументов типа, будь то типы значений или ссылочные типы.

Пример 11

```
using System;
class MyClass {
        //...
}
class Test<T> {
      public T obj;
      public Test () {
     // obj = null; // не годится (только для ссылочных типов)
     // obj = 0; // не годится (только для типов значений)
          obj = default(T); // Годится! Для любах типов
      }
       // ...
}
```

Пример 11

```
class DefaultDemo {
    static void Main() {
// Сконструировать объект класса Test, используя ссылочный тип
        Test<MyClass> x = new Test<MyClass>();
        if (x.obj == null)
            Console.WriteLine("Переменная x.obj имеет
   пустое значение <null>.");
// Сконструировать объект класса Test, используя тип значения
        Test<int> y = new Test<int>();
        if (y.obj == 0)
            Console.WriteLine("Переменная y.obj имеет
   значение 0.");
    }
}
```

Пример 12

```
struct XY<T> { // Эта структура является обобщенной
    T x;      T y;
    public XY(T a, T b) {
        x = a;          У = b;
    }
    public T X {
        get { return x; }
        set { x = value; }
    }
    public T Y {
        get { return y; }
        set { y = value; }
    }
}
```

Пример 12

```
class StructTest {
    static void Main() {
        XY<int> xy = new XY<int>(10, 20);
        XY<double> xy2 = new XY<double>(88.0, 99.0);
        Console.WriteLine(xy.X + ", " + xy.Y);        // 10, 20
        Console.WriteLine(xy2.X + ", " + xy2.Y);    // 88, 99
    }
}
```

Как и на обобщенные классы, на обобщенные структуры могут накладываться ограничения.

```
struct XY<T> where T : struct { // ограничение типа значения
// ...
```

Обобщенные методы

В методах, объявляемых в обобщенных классах, может использоваться параметр типа из данного класса, а следовательно, такие методы автоматически становятся обобщенными по отношению к параметру типа.

Но можно **создать обобщенный метод, заключенный в необобщенном классе**.

возвращаемый_тип имя_метода
<список_параметров_типа> (список_параметров) {
// ...
}

В объявлении обобщенного метода список параметров типа следует после имени метода.

Пример 13

```
using System;
class ArrayUtils { // Класс обработки массивов
public static bool CopyInsert<T> (T e, uint idx, T[] src, T[]
    target) {// Этот метод является обобщенным
       if (target.Length < src.Length+1)  return false;
       for (int i=0, j=0; i < src.Length; i++, j++) {
              if (i == idx) {
                   target[j] = e;       j++;
              }
              target[j] = src[i];
         }
         return true;
     }
}
```

Пример 13

```
class GenMethDemo {
    static void Main() {
        int[] nums = { 1, 2, 3 };
        int[] nums2 = new int[4];
       Console.Write("Содержимое массива nums: ");
        foreach (int x in nums)   Console.Write(x + " ");
       Console.WriteLine();
       // Обработать массив типа int
       ArrayUtils.CopyInsert (99, 2, nums, nums2);
      Console.Write("Содержимое массива nums2: ");
       foreach (int x in nums2)   Console.Write(x + " ");
       Console.WriteLine();
```

Пример 13

```
        //Обработать массив строк, используя метод CopyInsert
        string[] strs = {"Обобщения", "весьма", "эффективны."};
         string[] strs2 = new string[4];
         Console.Write("Содержимое массива strs: ");
         foreach (string s in strs)   Console.Write(s + " ");
          Console.WriteLine();
          // Ввести элемент в массив строк
          ArrayUtils.CopyInsert ("в C#", 1, strs, strs2);
          Console.Write("Содержимое массива strs2: ");
          foreach (string s in strs2)   Console.Write(s + " ");
          Console.WriteLine();
// ArrayUtils.CopyInsert (0.01, 2, nums, nums2);  // Ошибка
    }
}
```

Обобщенные методы

Аргументы типа могут быть указаны **явным образом**.

ArrayUtils.CopyInsert<**string**>("В C#", 1, strs, strs2);

Тип передаваемых аргументов **необходимо указывать явно в том случае**, если компилятор не сможет вывести тип параметра **T** или если требуется отменить выводимость типов.

На аргументы обобщенного метода можно наложить ограничения, указав их после списка параметров.

public static bool CopyInsert<T>(T e, uint idx, T[] src, T[] target) **where** T : **class** { // данные только ссылочных типов

ArrayUtils.CopyInsert (99, 2, nurns, nums2); /* вызов метода **CopyInsert**() не будет скомпилирован, поскольку **int** является типом значения, а не ссылочным типом */

Обобщенные делегаты

Как и методы, делегаты также могут быть обобщенными.

delegate возвращаемый_тип имя_делегата <список_параметров_типа> (список_аргументов);

Список_параметров_типа следует непосредственно после имени делегата. Преимущество обобщенных делегатов заключается в том, что их допускается определять в типизированной обобщенной форме, которую можно затем согласовать с любым совместимым методом.

Пример 14

```
delegate T SomeOp<T>(T v); // Обобщенный делегат
class GenDelegateDemo {
static int Sum(int v) { // Суммирование аргумента
        int result = 0;
        for (int i=v; i>0; i--)
            result += i;
            return result;
        }
// Возвратить строку, содержащую обратное значение аргумента
        static string Reflect(string str) {
            string result = "";
            foreach (char ch in str)  result = ch + result;
            return result;
        }
```

Пример 14

```
    static void Main() {
        // Сконструировать делегат типа int
         SomeOp<int> intDel = Sum;
         Console.WriteLine(intDel(3));
        // Сконструировать делегат типа string
         SomeOp<string> strDel = Reflect;
         Console.WriteLine(strDel("Привет") ) ;
        // SomeOp<int> intDel = Reflect; // Ошибка!
    }
}
```

6

тевирП

Пример 15

```
using System;
public interface ISeries<T> {
   T GetNext(); // Возвратить следующее по порядку число
   void Reset(); // Генерировать ряд
    // последовательных чисел с самого начала
   void SetStart (T v); // Задать начальное значение
}
class ByTwos<T> : ISeries<T> {
   T start;    T val;
// Следующий делегат определяет форму метода,
// вызываемого для генерирования очередного
// элемента в ряду последовательных значений
   public delegate T IncByTwo(T v);
```

Пример 15

```
// Этой ссылке на делегат будет присвоен метод,
// передаваемый конструктору класса ByTwos
   IncByTwo incr;
   public ByTwos(IncByTwo incrMeth) {
      start = default(T);
      val = default(T);
      incr = incrMeth;
   }
   public T GetNext() {
      val = incr(val);
      return val;
   }
```

Пример 15

```
   public void Reset() {  val = start; }
   public void SetStart(T v) {
      start = v;       val = start;
   }
}
class ThreeD {
   public int x, y, z;
   public ThreeD(int a, int b, int c) {
      x = a;       y = b;       z = c;
   }
}
```

Пример 15

```
class GenIntfDemo {
  static int IntPlusTwo(int v) {
    return v + 2;
  }
  static double DoublePlusTwo(double v) {
    return v + 2.0;
  }
  static ThreeD ThreeDPlusTwo(ThreeD v) {
    if (v==null) return new ThreeD(0, 0, 0);
    else return new ThreeD(v.x + 2, v.y + 2, v.z +
  2);
  }
```

Пример 15

```
static void Main() {
  ByTwos<int> intBT =
        new   ByTwos<int>(IntPlusTwo);
   for (int i=0; i < 5; i++)
        Console.Write (intBT.GetNext() + "  ");
   Console.WriteLine();
   ByTwos<double> dblBT =
        new ByTwos<double>(DoublePlusTwo);
   dblBT.SetStart(11.4);
   for (int i=0; i < 5; i++)
        Console.Write (dblBT.GetNext() + "  ");
   Console.WriteLine();
```

Пример 15

```
    ByTwos<ThreeD> ThrDBT =
          new ByTwos<ThreeD>(ThreeDPlusTwo);
    ThreeD coord;
    for (int i=0; i < 5; i++) {
        coord = ThrDBT.GetNext();
        Console.Write(coord.x + "," + coord.y + "," +
  coord.z + "   ");
    }
    Console.WriteLine();
  }
}
```

Обобщенные интерфейсы

На параметр типа в обобщенном интерфейсе могут накладываться ограничения таким же образом, как и в обобщенном классе.

public interface ISeries<T> **where** T : **class** {

Если реализуется такой вариант интерфейса **ISeries**, в реализующем его классе следует указать то же самое ограничение на параметр типа **T**.

class ByTwos<T> : ISeries<T> **where** T : **class** {

В силу ограничения ссылочного типа этот вариант интерфейса **ISeries** нельзя применять к типам значений.

Сравнение экземпляров параметра типа

Для сравнения двух объектов параметра обобщенного типа следует испльзовать интерфейс **IComparable** или **IComparable<T>** и/или интерфейс **IEquatable<T>**.

В обоих вариантах интерфейса **IComparable** для этой цели определен метод **CompareTo**(), а в интерфейсе **IEquatable<T>** — метод **Equals**().

Разновидности интерфейса **IComparable** предназначены для применения в тех случаях, когда требуется определить относительный порядок следования двух объектов.

А интерфейс **IEquatable** служит для определения равенства двух объектов.

Сравнение экземпляров параметра типа

public interface IEquatable<T>

Сравниваемый тип данных передается ему в качестве аргумента типа **T**.

В этом интерфейсе определяется метод:

bool Equals(T other)

В методе **Equals**() сравниваются вызывающий объект и другой объект, определяемый параметром **other**. В итоге возвращается логическое значение **true**, если оба объекта равны, а иначе — логическое значение **false**.

В ходе реализации интерфейса **IEquatable<T>** обычно требуется также переопределять методы **GetHashCode**() и **Equals**(Object), определенные в классе **Object**, чтобы они оказались совместимыми с конкретной реализацией метода **Equals**().

Пример 16

```
public static bool IsIn<T>(T what, T[] obs) {
   foreach(T v in obs)
     if (v == what) // Ошибка!
        return true;
   return false;
}
```

```
public static bool IsIn<T>(T what, T[] obs) where T :
    IEquatable<T> {
   foreach (T v in obs)
      if (v.Equals(what)) // Применяется метод Equals()
        return true;
   return false;
}
```

Сравнение экземпляров параметра типа

Для определения относительного порядка следования двух элементов применяется интерфейс **IComparable**.

public interface IComparable<T>

Сравниваемый тип передается ему в качестве аргумента типа **T**.

В этом интерфейсе определяется метод:

int CompareTo(T other)

В методе **CompareTo**() сравниваются вызывающий объект и другой объект, определяемый параметром **other**. В итоге возвращается нуль, если вызывающий объект оказывается больше, чем объект **other**; и отрицательное значение, если вызывающий объект оказывается меньше, чем объект **other**.

Для того чтобы воспользоваться методом **CompareTo**(), необходимо указать ограничение, которое требуется наложить на аргумент типа для реализации обобщенного интерфейса **IComparable**<T>. Затем достаточно вызвать метод **CompareTo**(), чтобы сравнить два экземпляра параметра типа.

Пример 17

```
using System;
class MyClass : IComparable<MyClass>,
    IEquatable<MyClass> {
   public int Val;
   public MyClass(int x) { Val = x; }
   // Реализовать обобщенный интерфейс IComparable<T>
   public int CompareTo(MyClass other) {
      return Val - other.Val;
   }
   // Реализовать обобщенный интерфейс IEquatable<T>
   public bool Equals(MyClass other) {
      return Val == other.Val;
   }
```

Пример 17

```
  // Переопределить метод Equals(Object)
  public override bool Equals(Object obj) {
    if (obj is MyClass)
        return Equals((MyClass) obj);
    return false;
  }
  // Переопределить метод GetHashCode()
  public override int GetHashCode() {
    return Val.GetHashCode();
  }
}
```

Пример 17

```
class CompareDemo {
  public static bool IsIn<T>(T what, T[] obs) where T :
   IEquatable<T> {
    foreach (T v in obs)
      if (v.Equals(what))  return true;
    return false;
  }
  public static bool InRange<T>(T what, T[] obs) where T
   : IComparable<T> {
    if (what.CompareTo(obs[0]) < 0 || what.CompareTo (obs
   [obs.Length-1] ) > 0) return false;
    return true;
  }
```

Пример 17

```
static void Main() {
   int[] nums = { 1, 2, 3, 4, 5 };
   if (IsIn(2, nums)) // Применить метод IsIn() к данным типа int
        Console.WriteLine("Найдено значение 2.");
   if (!IsIn(99, nums))
        Console.WriteLine("Не подлежит выводу.");
   MyClass[] mcs = { new MyClass(1), new MyClass(2),
 new MyClass(3), new MyClass(4) };
   if (IsIn (new MyClass(3), mcs)) // к объектам класса MyClass
        Console.WriteLine("Найден объект MyClass(3).");
   if (IsIn (new MyClass(99), mcs))
       Console.WriteLine("Не подлежит выводу.");
```

Пример 17

```
  // Применить метод InRange() к данным типа int
  if (InRange(2, nums)) Console.WriteLine("Значение 2 
находится в границах массива nums.");
  if (!InRange(0, nums)) Console.WriteLine("Значение 0 
НЕ находится в границах массива nums.");
 // Применить метод InRange() к объектам класса MyClass
  if (InRange(new MyClass(1), mcs))
     Console.WriteLine("Объект MyClass(1) находится в 
границах массива nums.");
  if (!InRange(new MyClass(5), mcs))
    Console.WriteLine("Объект MyClass(5) НЕ " +
"находится в границах массива nums.");
  }
}
```

Иерархии 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 первом случае аргументы типа, необходимые обобщенному базовому классу, должны передаваться всеми производными классами вверх по иерархии аналогично передаче аргументов конструктора.

Разумеется, в производный класс можно свободно добавлять его собственные параметры типа, если в этом есть потребность.

Пример 18

```
class Gen<T> { // Обобщенный базовый класс
   T ob;
   public Gen(T o) { ob = o; }
   public T GetOb() {  return ob; }
}
class Gen2<T> : Gen<T> { // Производный класс
   public Gen2(T o) : base(o) { ... }
}
class GenHierDemo {
   static void Main() {
      Gen2<string> g2 = new Gen2<string>("Привет");
      Console.WriteLine(g2.GetOb());
   }
}
```

Пример 19

```
using System;
class Gen<T> { // Обобщенный базовый класс
   T ob; // Объявить переменную типа T
   public Gen(T o) { ob = o; } // Передать конструктору
                               // ссылку типа T
   public T GetOb() {  return ob;  }
}
class Gen2<T, V> : Gen<T> { // Производный класс
   V ob2;
   public Gen2(T o, V o2) : base(o) {
      ob2 = o2;
   }
   public V GetObj2() { return ob2;  }
}
```

Пример 19

```
class GenHierDemo2 {
   static void Main() {
      // Создать объект класса Gen2
      // с параметрами типа string и int
      Gen2<string, int> x =
            new Gen2<string, int>("Значение равно: ", 99);
      Console.Write(x.GetOb());
      Console.WriteLine(x.GetObj2());
   }
}
```

Пример 20

```
using System;
class NonGen { // Необобщенный базовый класс
   int num;
   public NonGen(int i) { num = i; }
   public int GetNum() { return num; }
}

class Gen<T> : NonGen { // Обобщенный производный класс
   T ob;
   public Gen(T o, int i) : base (i) { ob = o; }
   public T GetOb() { return ob; }
}
```

Пример 20

```
class HierDemo3 {
   static void Main() {
    // Создать объект класса Gen с параметром типа string
      Gen<String> w = new Gen<String>("Привет", 47);
      Console.Write(w.GetOb() + " ");
      Console.WriteLine(w.GetNum());
   }
}
```

Пример 21

```
class Gen<T> { // Обобщенный базовый класс
    protected T ob;
    public Gen(T o) { ob = o; }
   public virtual T GetOb() {
     Console.Write("Метод GetOb() из класса Gen" + " возвращает результат: ");
       return ob;
   }
}
class Gen2<T> : Gen<T> { // Производный класс
    public Gen2 (T o) : base(o) { }
  public override T GetOb(){ //Переопределить метод GetOb()
     Console.Write("Метод GetOb() из класса Gen2" + " возвращает результат: ");
       return ob;
   }
}
```

Пример 21

```
class OverrideDemo {
  static void Main() {
    // Создать объект класса Gen с параметром типа int
    Gen<int> iOb = new Gen<int> (88);
    // Вызвать вариант метода GetOb() из класса Gen
    Console.WriteLine(iOb.GetOb());
    // Создать объект класса Gen2 и присвоить
    // ссылку на него переменной iOb типа Gen<int>
    iOb = new Gen2<int>(99);
    // Вызывать вариант-метода GetOb() из класса Gen2
    Console.WriteLine (iOb.GetOb());
  }
}
```

Перегрузка методов

Методы, параметры которых объявляются с помощью параметров типа, могут быть перегружены. Но правила их перегрузки упрощаются по сравнению с методами без параметров типа. Варианты перегружаемого метода должны отличаться по типу или количеству их параметров. Но типовые различия должны определяться не по параметру обобщенного типа, а исходя из аргумента типа, подставляемого вместо параметра типа при конструировании объекта этого типа. Следовательно, метод с параметрами типа может быть перегружен таким образом, что он окажется пригодным не для всех возможных случаев, хотя и будет выглядеть верно.

Пример 22

```
using System;
// Обобщенный класс, содержащий метод Set(),
// перегрузка которого может привести к неоднозначности
class Gen<T, V> {
   T obi;    V ob2;
   // ...
// В некоторых случаях эти два метода не будут
// отличаться своими параметрами типа
   public void Set(T o) {
      obi = o;
   }
   public void Set(V o) {
      ob2 = o;
   }
}
```

Пример 22

```
class AmbiguityDemo {
   static void Main() {
      Gen<int, double> ok = new Gen<int, double>();
      Gen<int, int> notOK = new Gen<int, int>();
      ok.Set(10); // верно, т. к. аргументы типа отличаются
      notOK.Set(10); // неоднозначно, поскольку
                     // аргументы ничем не отличаются!
   }
}
```

Ковариантность и контравариантность

Понятия **ковариантности** и **контравариантности** связаны с возможностью использовать в приложении вместо некоторого типа другой тип, который находится ниже или выше в иерархии наследования.

Ковариантность: позволяет использовать более конкретный тип, чем заданный изначально

Контравариантность: позволяет использовать более универсальный тип, чем заданный изначально

Инвариантность: позволяет использовать только заданный тип (по умолчанию)

Обобщенные интерфейсы могут быть **ковариантными**, если к универсальному параметру применяется ключевое слово **out**. Для создания **контравариантного** интерфейса надо использовать ключевое слово **in**.

Пример 23

```
class Account{
    static Random rnd = new Random();
    public void DoTransfer() {
        int sum = rnd.Next(10, 120);
        Console.WriteLine("Клиент положил на счет
    {0} долларов", sum);
    }
}
class DepositAccount : Account { }
interface IBank<out T>  where T : Account{
    T DoOperation();
}
```

Пример 23

```
class Bank: IBank<DepositAccount>{
    public DepositAccount DoOperation()    {
        DepositAccount acc = new DepositAccount();
        acc.DoTransfer();
        return acc;
    }
}
static void Main(string[] args){
    IBank<DepositAccount> depositBank = new Bank();
    depositBank.DoOperation();
    IBank<Account> ordinaryBank = depositBank;
    ordinaryBank.DoOperation();
    Console.ReadLine();
}
```

Пример 24

```
// Добавлено
interface IBank<in T>  where T : Account{
    void DoOperation(T account);
}
class Bank<T>: IBank<T> where T : Account{
    public void DoOperation(T account)    {
        account.DoTransfer();
    }
}
// Классы Account и DepositAccount
// остаются без изменений
```

Пример 24

```
static void Main(string[] args){
    Account account = new Account();
    IBank<Account> ordinaryBank =
        new Bank<Account>();
    ordinaryBank.DoOperation(account);
    DepositAccount depositAcc =
        new DepositAccount();
    IBank<DepositAccount> depositBank =
   ordinaryBank;
    depositBank.DoOperation(depositAcc);
    Console.ReadLine();
}
```

Контрольные вопросы

1. Для чего нужны обобщенные методы?
2. Какие действия обычно выполняют обобщенные классы и методы?