№ 8 Обобщения

**Курносенко Софья**

**Задание**

1. Создайте **обобщенный интерфейс** с операциями добавить, удалить, просмотреть.

2. Возьмите за основу лабораторную № 4 «Перегрузка операций» и сделайте из нее **обобщенный тип (класс) CollectionType<T>**, в который вложите обобщённую коллекцию. Наследуйте в обобщенном классе интерфейс из п.1. Реализуйте необходимые методы. Добавьте **обработку исключений** c finally. Наложите какое-либо **ограничение** на обобщение.

3. Проверьте использование обобщения для стандартных типов данных (в качестве стандартных типов использовать целые, вещественные и т.д.). Определить пользовательский класс, который будет использоваться в качестве параметра обобщения. Для пользовательского типа взять класс из лабораторной №5 «Наследование».

4. Добавьте методы сохранения объектов обобщённого типа CollectionType<T> в файл и чтения из него (на выбор: текстовый | xml | json).

**► Обобщения**

# System.Collections.Generic содержит интерфейсы и классы, определяющие универсальные коллекции, которые позволяют пользователям создавать строго типизированные коллекции, обеспечивающие повышенную производительность и безопасность типов по сравнению с неуниверсальными строго типизированными коллекциями.

Кроме обычных типов фреймворк .NET также поддерживает обобщенные типы (generics), а также создание обобщенных методов. Чтобы разобраться в особенности данного явления, сначала посмотрим на проблему, которая могла возникнуть до появления обобщенных типов. Посмотрим на примере. Допустим, мы определяем класс для представления банковского счета. К примеру, он мог бы выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Account  {      public int Id { get; set; }      public int Sum { get; set; }  } |

Класс Account определяет два свойства: Id - уникальный идентификатор и Sum - сумму на счете.

Здесь идентификатор задан как числовое значение, то есть банковские счета будут иметь значения 1, 2, 3, 4 и так далее. Однако также нередко для идентификатора используются и строковые значения. И у числовых, и у строковых значений есть свои плюсы и минусы. И на момент написания класса мы можем точно не знать, что лучше выбрать для хранения идентификатора - строки или числа. Либо, возможно, этот класс будет использоваться другими разработчиками, которые могут иметь свое мнение по данной проблеме.

И на первый взгляд, чтобы выйти из подобной ситуации, мы можем определить свойство Id как свойство типа object. Так как тип object является универсальным типом, от которого наследуется все типы, соответственно в свойствах подобного типа мы можем сохранить и строки, и числа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Account  {      public object Id { get; set; }      public int Sum { get; set; }  } |

Затем этот класс можно было использовать для создания банковских счетов в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Account account1 = new Account { Sum = 5000 };  Account account2 = new Account { Sum = 4000 };  account1.Id = 2;  account2.Id = "4356";  int id1 = (int)account1.Id;  string id2 = (string)account2.Id;  Console.WriteLine(id1);  Console.WriteLine(id2); |

Все вроде замечательно работает, но такое решение является не очень оптимальным. Дело в том, что в данном случае мы сталкиваемся с такими явлениями как **упаковка (boxing)** и **распаковка (unboxing)**.

Так, при присвоении свойству Id значения типа int, происходит упаковка этого значения в тип Object:

|  |  |
| --- | --- |
| 1 | account1.Id = 2;        // упаковка в значения int в тип Object |

Чтобы обратно получить данные в переменную типов int, необходимо выполнить распаковку:

|  |  |
| --- | --- |
| 1 | int id1 = (int)account1.Id;     // Распаковка в тип int |

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа **System.Object** и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

Кроме того, существует другая проблема - проблема безопасности типов. Так, мы получим ошибку во время выполнения программы, если напишем следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | Account account2 = new Account { Sum = 4000 };  account2.Id = "4356";  int id2 = (int)account2.Id;     // Исключение InvalidCastException |

Мы можем не знать, какой именно объект представляет Id, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException.

Эти проблемы были призваны устранить **обобщенные типы** (также часто называют универсальными типами). Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Поэтому определим класс Account как обощенный:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Account<T>  {      public T Id { get; set; }      public int Sum { get; set; }  } |

Угловые скобки в описании class Account<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр **T** в угловых скобках еще называется **универсальным параметром**, так как вместо него можно подставить любой тип.

Например, вместо параметра T можно использовать объект int, то есть число, представляющее номер счета. Это также может быть объект string либо любой другой класс или структура:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Account<int> account1 = new Account<int> { Sum = 5000 };  Account<string> account2 = new Account<string> { Sum = 4000 };  account1.Id = 2;        // упаковка не нужна  account2.Id = "4356";  int id1 = account1.Id;  // распаковка не нужна  string id2 = account2.Id;  Console.WriteLine(id1);  Console.WriteLine(id2); |

Поскольку класс Account является обобщенным, то при определении переменной после названия типа в угловых скобках необходимо указать тот тип, который будет использоваться вместо универсального параметра T. В данном случае объекты Account типизируются типами int и string:

|  |  |
| --- | --- |
| 1  2 | Account<int> account1 = new Account<int> { Sum = 5000 };  Account<string> account2 = new Account<string> { Sum = 4000 }; |

Поэтому у первого объекта account1 свойство Id будет иметь тип int, а у объекта account2 - тип string.

При попытке присвоить значение свойства Id переменной другого типа мы получим ошибку компиляции:

|  |  |
| --- | --- |
| 1  2  3 | Account<string> account2 = new Account<string> { Sum = 4000 };  account2.Id = "4356";  int id1 = account2.Id;  // ошибка компиляции |

Тем самым мы избежим проблем с типобезопасностью. Таким образом, используя обобщенный вариант класса, мы снижаем время на выполнение и количество потенциальных ошибок.

### 

### Значения по умолчанию

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение, в том числе и null. Но напрямую мы его присвоить не можем:

|  |  |
| --- | --- |
| 1 | T id = null; |

В этом случае нам надо использовать оператор **default(T)**. Он присваивает ссылочным типам в качестве значения null, а типам значений - значение 0:

|  |  |
| --- | --- |
| 1  2  3  4 | class Account<T>  {      T id = default(T);  } |

### Статические поля обобщенных классов

При типизации обобщенного класса определенным типом будет создаваться свой набор статических членов. Например, в классе Account определено следующее статическое поле:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Account<T>  {      public static T session;        public T Id { get; set; }      public int Sum { get; set; }  } |

Теперь типизируем класс двумя типами int и string:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Account<int> account1 = new Account<int> { Sum = 5000 };  Account<int>.session = 5436;    Account<string> account2 = new Account<string> { Sum = 4000 };  Account<string>.session = "45245";    Console.WriteLine(Account<int>.session);      // 5436  Console.WriteLine(Account<string>.session);   // 45245 |

В итоге для Account<string> и для Account<int> будет создана своя статическая переменная session.

### Использование нескольких универсальных параметров

Обобщения могут использовать несколько универсальных параметров одновременно, которые могут представлять различные типы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Transaction<U, V>  {      public U FromAccount { get; set; }  // с какого счета перевод      public U ToAccount { get; set; }    // на какой счет перевод      public V Code { get; set; }         // код операции      public int Sum { get; set; }        // сумма перевода  } |

Здесь класс Transaction использует два универсальных параметра. Применим данный класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Account<int> acc1 = new Account<int> { Id = 1857, Sum = 4500 };  Account<int> acc2 = new Account<int> { Id = 3453, Sum = 5000 };    **Transaction<Account<int>, string> transaction1 = new Transaction<Account<int>, string>**  {      FromAccount = acc1,      ToAccount = acc2,      Code = "45478758",      Sum = 900  }; |

Здесь объект Transaction типизируется типами Account<int> и string. То есть в качестве универсального параметра U используется класс Account<int>, а для параметра V - тип string. При этом, как можно заметить, класс, которым типизируется Transaction, сам является обобщенным.

### Обобщенные методы

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Program  {      private static void Main(string[] args)      {          int x = 7;          int y = 25;          Swap<int>(ref x, ref y); // или так Swap(ref x, ref y);          Console.WriteLine($"x={x}    y={y}");   // x=25   y=7            string s1 = "hello";          string s2 = "bye";          Swap<string>(ref s1, ref s2); // или так Swap(ref s1, ref s2);          Console.WriteLine($"s1={s1}    s2={s2}"); // s1=bye   s2=hello            Console.Read();      }      public static void Swap<T> (ref T x, ref T y)      {          T temp = x;          x = y;          y = temp;      }  } |

Здесь определен обобщенный метод Swap, который принимает параметры по ссылке и меняет их значения. При этом в данном случае не важно, какой тип представляют эти параметры.

В методе Main вызываем метод Swap, типизируем его определенным типом и передаем ему некоторые значения.

**Свойства**

1)Универсальный тип может содержать другой универсальный тип



2) Универсальные типы перегружаются на основе количества параметров



3) Универсальными могут быть классы, структуры, интерфейсы, делегаты, методы

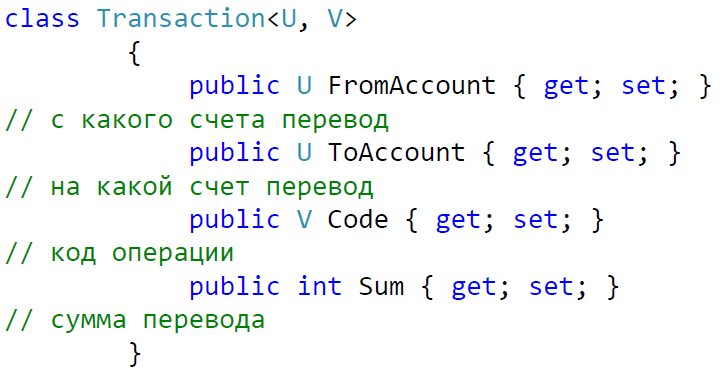


4 ) Могут содержать статические типы

5) Доступность конструируемых типов определяется на основе пересечения доступности универсального типа и типа в списке аргументов

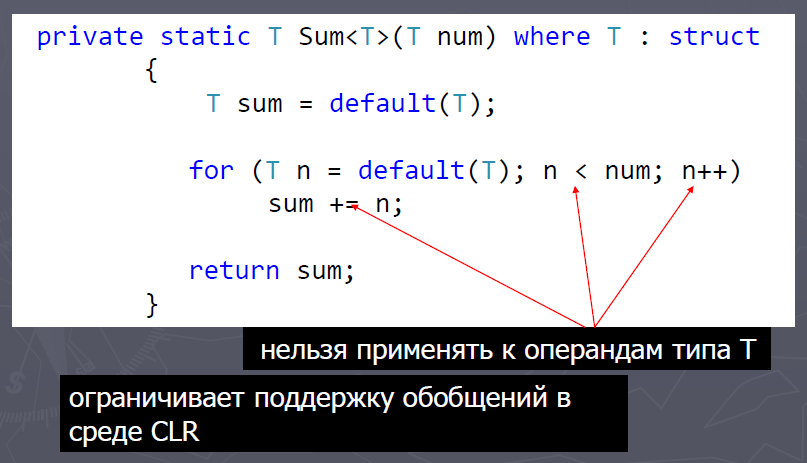


6) могут использовать несколько универсальных параметров одновременно



7) поддерживает механизм ограничений

**Использование переменных универсального типа в качестве операндов**



## ► Ограничения обобщений

### Ограничения универсальных типов

С помощью универсальных параметров мы можем типизировать обобщенные классы любым типом. Однако иногда возникает необходимость конкретизировать тип. Например, у нас есть следующий класс Account, который представляет банковский счет:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Account  {      public int Id { get; private set;} // номер счета      public int Sum { get; set; }      public Account(int \_id)      {          Id = \_id;      }  } |

Для перевода средств с одного счета на другой мы можем определить класс Transaction, который для выполнения всех операций будет использовать объекты класса Account.

Но у класса Account может быть много наследников: DepositAccount (депозитный счет), DemandAccount (счет до востребования) и т.д. И мы не можем знать, какие именно типы счетов будут использоваться в классе Transaction. Возможно, транзакции будут проводиться только между счетами до востребования. И в этом случае в качестве универсального параметра можно установить тип Account:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class Transaction<T> where T: Account  {      public T FromAccount { get; set; }  // с какого счета перевод      public T ToAccount { get; set; }    // на какой счет перевод      public int Sum { get; set; }        // сумма перевода        public void Execute()      {          if (FromAccount.Sum > Sum)          {              FromAccount.Sum -= Sum;              ToAccount.Sum += Sum;              Console.WriteLine($"Счет {FromAccount.Id}: {FromAccount.Sum}$ \nСчет {ToAccount.Id}: {ToAccount.Sum}$");          }          else          {              Console.WriteLine($"Недостаточно денег на счете {FromAccount.Id}");          }      }  } |

С помощью выражения where T : Account мы указываем, что используемый тип T обязательно должен быть классом Account или его наследником. Благодаря подобному ограничению мы можем использовать внутри класса Transaction все объекты типа T именно как объекты Account и соответственно обращаться к их свойствам и методам.

Теперь применим класс Transaction в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class Program  {      static void Main(string[] args)      {          Account acc1 = new Account(1857) { Sum = 4500 };          Account acc2 = new Account(3453) { Sum = 5000 };          Transaction<Account> transaction1 = new Transaction<Account>          {              FromAccount = acc1,              ToAccount = acc2,              Sum = 6900          };          transaction1.Execute();            Console.ReadLine();      }  } |

Следует учитывать, что **только один класс может использоваться в качестве ограничения**.

В качестве ограничения также может выступать и обобщенный класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | class Program  {      private static void Main(string[] args)      {          Account<int> acc1 = new Account<int>(1857) { Sum = 4500 };          Account<int> acc2 = new Account<int>(3453) { Sum = 5000 };            Transaction<Account<int>> transaction1 = new Transaction<Account<int>>          {              FromAccount = acc1,              ToAccount = acc2,              Sum = 6900          };          transaction1.Execute();            Console.Read();      }  }  class Account<T>  {      public T Id { get; private set; } // номер счета      public int Sum { get; set; }      public Account(T \_id)      {          Id = \_id;      }  }  class Transaction<T> where T: Account<int>  {      public T FromAccount { get; set; }  // с какого счета перевод      public T ToAccount { get; set; }    // на какой счет перевод      public int Sum { get; set; }        // сумма перевода        public void Execute()      {          if (FromAccount.Sum > Sum)          {              FromAccount.Sum -= Sum;              ToAccount.Sum += Sum;              Console.WriteLine($"Счет {FromAccount.Id}: {FromAccount.Sum}$ \nСчет {ToAccount.Id}: {ToAccount.Sum}$");          }          else          {              Console.WriteLine($"Недостаточно денег на счете {FromAccount.Id}");          }      }  } |

В данном случае класс Transaction типизирован классом Account<int>. Класс Account же может быть типизирован абсолютно любым типом. Однако класс Transaction может использовать только объекты класса Account<int> или его наследников. То есть следующий код ошибочен и работать не будет:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Account<string> acc1 = new Account<string>("34") { Sum = 4500 };  Account<string> acc2 = new Account<string>("45") { Sum = 5000 };    // так нельзя написать, так как Bank должен быть типизирован классом Account<int> или его наследником  Transaction<Account<string>> transaction1 = new Transaction<Account<string>>  {      FromAccount = acc1,      ToAccount = acc2,      Sum = 900  }; |

В качестве ограничений мы можем использовать следующие типы:

* Классы
* Интерфейсы
* class => универсальный параметр должен представлять класс
* struct => универсальный параметр должен представлять структуру
* new() => универсальный параметр должен представлять тип, который имеет общедоступный (public) конструктор без параметров

### Стандартные ограничения

Есть ряд стандартных ограничений, которые мы можем использовать. В частности, можно указать ограничение, чтобы использовались только структуры или другие типы значений:

|  |  |
| --- | --- |
| 1  2 | class Account<T> where T : struct  {} |

При этом **использовать в качестве ограничения конкретные структуры** в отличие от классов **нельзя**.

Также можно задать в качестве ограничения ссылочные типы:

|  |  |
| --- | --- |
| 1  2 | class Transaction<T> where T : class  {} |

А также можно задать с помощью слова **new** в качестве ограничения класс или структуру, которые имеют общедоступный конструктор без параметров:

|  |  |
| --- | --- |
| 1  2 | class Transaction<T> where T : new()  {} |

Если для универсального параметра задано несколько ограничений, то они должны идти в определенном порядке:

1. Название класса, class, struct. Причем мы можем одновременно определить только одно из этих ограничений
2. Название интерфейса
3. new()

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | interface IAccount  {      int CurrentSum { get; set;}  }  class Person  {      public string Name { get; set; }  }    class Transaction<T> where T: Person, IAccount, new()  {    } |

### Использование нескольких универсальных параметров

Если класс использует несколько универсальных параметров, то последовательно можно задать ограничения к каждому из них:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Transaction<U, V>          where U : Account<int>          where V : struct  {    } |

### Ограничения методов

Подобным образом можно использовать и ограничения методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | private static void Main(string[] args)  {      Account<int> acc1 = new Account<int>(1857) { Sum = 4500 };      Account<int> acc2 = new Account<int>(3453) { Sum = 5000 };        Transact<Account<int>>(acc1, acc2, 900);        Console.Read();  }    public static void Transact<T>(T acc1, T acc2, int sum) where T : Account<int>  {      if (acc1.Sum > sum)      {          acc1.Sum -= sum;          acc2.Sum += sum;      }      Console.WriteLine($"acc1: {acc1.Sum}   acc2: {acc2.Sum}");  } |

Метод Transact в качестве ограничения принимает тип Account<int>.

## ► Ковариантность и контравариантность обобщенных интерфейсов

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

Имеется три возможных варианта поведения:

* **Ковариантность**: позволяет использовать более конкретный тип, чем заданный изначально
* **Контравариантность**: позволяет использовать более универсальный тип, чем заданный изначально
* **Инвариантность**: позволяет использовать только заданный тип

C# позволяет создавать ковариантные и контравариантные обобщенные интерфейсы. Эта функциональность повышает гибкость при использовании обобщенных интерфейсов в программе. По умолчанию все обобщенные интерфейсы, например, IAccout<T> являются инвариантными.

## ► Наследование обобщенных типов

Один обобщенный класс может быть унаследован от другого обобщенного. При этом можно использовать различные варианты наследования.

Допустим, у нас есть следующий базовый класс Account:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Account<T>  {      public T Id { get; private set; }      public Account(T \_id)      {          Id = \_id;      }  } |

**Первый вариант заключается в создание класса-наследника, который типизирован тем же типом, что и базовый**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class UniversalAccount<T> : Account<T>  {      public UniversalAccount(T id) : base(id)      {        }  } |

Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Account<string> acc1 = new Account<string>("34");  Account<int> acc3 = new UniversalAccount<int>(45);  UniversalAccount<int> acc2 = new UniversalAccount<int>(33);  Console.WriteLine(acc1.Id);  Console.WriteLine(acc2.Id);  Console.WriteLine(acc3.Id); |

**Второй вариант представляет создание обычного необобщенного класса-наследника**. В этом случае у базового класса надо явным образом определить используемый тип:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class StringAccount : Account<string>  {      public StringAccount(string id) : base(id)      {      }  } |

Теперь в производном классе в качестве типа будет использоваться тип string. Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4 | StringAccount acc4 = new StringAccount("438767");  Account<string> acc5 = new StringAccount("43875");  // так нельзя написать  //Account<int> acc6 = new StringAccount("45545"); |

**Третий вариант представляет типизацию производного класса параметром совсем другого типа, отличного от универсального параметра в базовом классе.** В этом случае для базового класса также надо указать используемый тип:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class IntAccount<T> : Account<int>  {      public T Code { get; set; }      public IntAccount(int id) : base(id)      {      }  } |

Здесь тип IntAccount типизирован еще одним типом, который может не совпадать с типом, который используется базовым классом. Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4 | IntAccount<string> acc7 = new IntAccount<string>(5) { Code = "r4556" };  Account<int> acc8 = new IntAccount<long>(7) { Code = 4587 };  Console.WriteLine(acc7.Id);  Console.WriteLine(acc8.Id); |

И также в классах-наследниках можно сочетать использование универсального параметра из базового класса с применением своих параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class MixedAccount<T, K> : Account<T>      where K : struct  {      public K Code { get; set; }      public MixedAccount(T id) : base(id)      {        }  } |

Здесь в дополнение к унаследованному от базового класса параметру T добавляется новый параметр K. Также если необходимо при этом задать ограничения, мы их можем указать после названия базового класса. Применение класса:

|  |  |
| --- | --- |
| 1  2  3  4 | MixedAccount<string, int> acc9 = new MixedAccount<string, int>("456") { Code = 356 };  Account<string> acc10 = new MixedAccount<string, int>("9867") { Code = 35678 };  Console.WriteLine(acc9.Id);  Console.WriteLine(acc10.Id); |

При этом стоит учитывать, что если на уровне базового класса для универсального параметра установлено ограничение, то подобное ограничение должно быть определено и в производных классах, которые также используют этот параметр:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Account<T> where T : class  {      public T Id { get; private set; }      public Account(T \_id)      {          Id = \_id;      }  }  class UniversalAccount<T> : Account<T>      where T: class  {      public UniversalAccount(T id) : base(id)      {        }  } |

То есть если в базовом классе в качестве ограничение указано class, то есть любой класс, то в производном классе также надо указать в качестве ограничения class, либо же какой-то конкретный класс.

## ► Интерфейсы в обобщениях

### Интерфейсы как ограничения обобщений

Интерфейсы могут выступать в качестве ограничений обобщений. При этом если в качестве ограничения можно указать только один класс, то интерфейсов можно указать несколько.

Допустим, у нас есть следующие интерфейсы и класс, который их реализует:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | interface IAccount  {      int CurrentSum { get; } // Текущая сумма на счету      void Put(int sum);      // Положить деньги на счет      void Withdraw(int sum); // Взять со счета  }  interface IClient  {      string Name { get; set; }  }  class Client : IAccount, IClient  {      int \_sum; // Переменная для хранения суммы      public Client(string name, int sum)      {          Name = name;          \_sum = sum;      }        public string Name { get; set; }      public int CurrentSum      {          get { return \_sum; }      }      public void Put(int sum)      {          \_sum += sum;      }      public void Withdraw(int sum)      {          if (sum <= \_sum)          {              \_sum -= sum;          }      }  } |

Используем выше перечисленные интерфейсы в качестве ограничений обобщенного класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Transaction<T> where T: IAccount, IClient  {      public void Operate(T acc1, T acc2, int sum)      {          if(acc1.CurrentSum >= sum)          {              acc1.Withdraw(sum);              acc2.Put(sum);              Console.WriteLine($"{acc1.Name} : {acc1.CurrentSum}\n{acc2.Name} : {acc2.CurrentSum}");          }      }  } |

В данном случае параметр T представляет тип, который который реализует сразу два интерфейса IAccount и IClient. Например, выше определен класс Client, который реализует оба интерфейса, поэтому мы можем данным типом типизировать объекты Transaction:

|  |  |
| --- | --- |
| 1  2  3  4 | Client account1 = new Client("Tom", 200);  Client account2 = new Client("Bob", 300);  Transaction<Client> transaction = new Transaction<Client>();  transaction.Operate(account1, account2, 150); |

Также параметр T может представлять интерфейс, который наследуется от обоих интерфейсов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | interface IClientAccount : IAccount, IClient  {    }  class ClientAccount : IClientAccount  {      int \_sum;      public ClientAccount(string name, int sum)      {          \_sum = sum; Name = name;      }      public int CurrentSum { get { return \_sum; } }        public string Name { get; set; }        public void Put(int sum)      {          \_sum += sum;      }      public void Withdraw(int sum)      {          if (\_sum >= sum) \_sum -= sum;      }  } |

В этом случае объекты Transaction мы можем типизировать типом IClientAccount:

|  |  |
| --- | --- |
| 1  2  3  4 | IClientAccount account3 = new ClientAccount("Alice", 400);  IClientAccount account4 = new ClientAccount("Kate", 500);  Transaction<IClientAccount> operation = new Transaction<IClientAccount>();  operation.Operate(account3, account4, 200); |

### Обобщенные интерфейсы

Как и классы, интерфейсы могут быть обобщенными:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | interface IUser<T>  {      T Id { get; }  }  class User<T> : IUser<T>  {      T \_id;      public User(T id)      {          \_id = id;      }      public T Id { get { return \_id; } }  } |

Интерфейс IUser типизирован параметром T, который при реализации интерфейса используется в классе User. В частности, переменная \_id определена как T, что позволяет нам использовать для id различные типы.

Определим две реализации: одна в качестве параметра будет использовать тип int, а другая - тип string:

|  |  |
| --- | --- |
| 1  2  3  4  5 | IUser<int> user1 = new User<int>(6789);  Console.WriteLine(user1.Id);    // 6789    IUser<string> user2 = new User<string>("12345");  Console.WriteLine(user2.Id);    // 12345 |

Также при реализации интерфейса мы можем явным образом указать, какой тип будет использоваться для параметра T:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class IntUser : IUser<int>  {      int \_id;      public IntUser(int id)      {          \_id = id;      }      public int Id { get { return \_id; } }  } |

##### **Пример реализации обобщенного интерфейса для одиночного типа T**

##### **Реализация основных операций над числами: сложение, вычитание**

В примере объявляется обобщенный интерфейс, который получает параметром тип T. В интерфейсе реализованы операции сложения и вычитания чисел.

using System;

namespace ConsoleApp19

{

// Интерфейс, получающий параметром тип T

interface Operations<T>

{

// Объявление методов, которые используют тип T

double Add(T var1, T var2);

double Sub(T var1, T var2);

}

// Класс, реализующий интерфейс MyInterface<T1, T2>

class MyClass<T> : Operations<T>

{

// Реализация метода, который использует тип T

public double Add(T var1, T var2)

{

double res = Convert.ToDouble(var1) + Convert.ToDouble(var2);

return res;

}

public double Sub(T var1, T var2)

{

double res = Convert.ToDouble(var1) + Convert.ToDouble(var2);

return res;

}

}

class Program

{

static void Main(string[] args)

{

// Демонстрация использования обобщенного интерфейса

// 1. Для типа-заполнителя int

MyClass<int> mc1 = new MyClass<int>();

double res1 = mc1.Add(23, 48);

Console.WriteLine("res1 = {0}", res1);

// 2. Для типа-заполнителя float

MyClass<float> mc2 = new MyClass<float>();

double res2 = mc2.Sub(8.804f, 1.704f);

Console.WriteLine("res2 = {0:f}", res2);

Console.ReadKey();

}

}

}

Результат выполнения программы

res1 = 71

res2 = 10.51

##### **Пример реализации в классе обобщенного интерфейса для двух типов T1, T2**

Объявляется интерфейс MyInterface, который содержит методы вывода данных типов T1, T2. В классе MyClass<T1, T2>, реализующего этот интерфейс, продемонстрированы:

* реализация методов интерфейса MyInterface<T1, T2>;
* доступ к данным типов T1, T2.

using System;

namespace ConsoleApp19

{

// Интерфейс, который получает два параметра типа T1, T2

interface MyInterface<T1, T2>

{

// Объявление методов, использующих типы T1, T2

void Print1(T1 var1);

void Print2(T2 var2);

}

// Класс, реализующий интерфейс MyInterface<T1, T2>

class MyClass<T1, T2> : MyInterface<T1, T2>

{

// Реализация метода, использующего тип T1

public void Print1(T1 var1)

{

Console.WriteLine("var1 = {0}", var1);

}

// Реализация метода, использующего тип T2

public void Print2(T2 var2)

{

Console.WriteLine("var2 = {0}", var2);

}

}

class Program

{

static void Main(string[] args)

{

// 1. Объявить экземпляр класса с типами-заполнителями int, string

MyClass<int, string> mc1 = new MyClass<int, string>();

mc1.Print1(233);

mc1.Print2("Hello world!");

// 2. Объявить экземпляр класса с типами-заполнителями double, char

MyClass<double, char> mc2 = new MyClass<double, char>();

mc2.Print1(8.77);

mc2.Print2('z');

// 3. Реализация для типов long, long

MyClass<long, long> mc3 = new MyClass<long, long>();

mc3.Print1(32323L);

mc3.Print2(30000L);

Console.ReadKey();

}

}

}

Результат выполнения программы

var1 = 233

var2 = Hello world!

var1 = 8.77

var2 = z

var1 = 32323

var2 = 30000

## ► Оператор using

## Предоставляет удобный синтаксис, обеспечивающий правильное использование объектов [IDisposable](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable) . Интерфейс IDisposable объявляет один единственный метод Dispose, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов.

## ◦◦

**Неуправляемые ресурсы** - это все, о чем не знает сборщик мусора. Например:

* Открыть файлы
* Открыть сетевые подключения
* Неуправляемая память
* В XNA: буферы вершин, буферы индексов, текстуры и т.д.

Обычно вы хотите освободить эти неуправляемые ресурсы, прежде чем потерять все ссылки, которые у вас есть, на объект, управляющий ими. Вы делаете это, вызывая Dispose на этом объекте или (в С#) с помощью инструкции using, которая будет обрабатывать вызов Dispose для вас.

## Программа в .NET имеет свой менеджер памяти, который занимается автоматическим выделением памяти под объекты и очисткой памяти после объектов (сборка мусора). Это значит что программист не заботится о выделении памяти и ее очистке. Такой тип памяти называется управляемой памятью. И все объекты, которые вы создаете в .NET располагаются в управляемой памяти. Это все встроенные типы: int, string, double, Array, List<> и т.п. Все они - управляемые объекты и располагаются в управляемой памяти. Однако, программа работает не только с управляемыми объектами. Часть объектов связана с операционной и файловой системами, с WinAPI, с драйверами, с видеокартой, с сетью и т.д. Понятно, что все эти внешние ресурсы никак не связаны с менеджером памяти .NET и потому на них не распространяется автоматическое распределение памяти. Такая память и такие объекты называются неуправляемыми. Как определить, какие объекты управляемы а какие нет? В принципе все объекты и типы .NET - являются управляемыми объектами. Но часть из них являются просто оболочками над неуправляемыми ресурсами. Например класс Form внутри содержит ссылку на хендл окна (HWND) из неуправляемого WinAPI. Или например, если вы открыли файл с помощью FileStream, то сам FileStream является управляемым объектом, но внутри содержит хендл файла, который ссылается на неуправляемый файл из WinAPI. И так далее. Обычно, те объекты, которые содержат ссылки на неуправляемые ресурсы, реализуют интерфейс [IDisposable](https://msdn.microsoft.com/ru-ru/library/system.idisposable(v=vs.110).aspx" \o "https://msdn.microsoft.com/ru-ru/library/system.idisposable(v=vs.110).aspx" \t "_blank). По наличию этого интерфейса и можно определить связь объекта с неуправляемыми ресурсами. Этот же интерфейс используется и для явной очистки неуправляемой памяти (поскольку автоматически неуправляемая память очиститься не может).

## ◦◦

Для вызова метода Dispose можно использовать следующую конструкцию try..catch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Person p=null;  try  {      p = new Person();  }  finally  {      if (p != null)      {          p.Dispose();      }  } |

Однако синтаксис C# также предлагает синонимичную конструкцию для автоматического вызова метод Dispose - конструкцию **using**:

|  |  |
| --- | --- |
| 1  2  3 | using (Person p = new Person())  {  } |

Конструкция using оформляет блок кода и создает объект некоторого класса, который реализует интерфейс IDisposable, в частности, его метод Dispose. При завершении блока кода у объекта вызывается метод Dispose.

Важно, что данная конструкция применяется только для классов, которые реализуют интерфейс IDisposable.

Ее использование:

using System;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

Test();

Console.ReadLine();

}

private static void Test()

{

using (Person p = new Person { Name = "Tom" })

{

// переменная p доступна только в блоке using

Console.WriteLine($"Некоторые действия с объектом Person. Получим его имя: {p.Name}");

}

Console.WriteLine("Конец метода Test");

}

}

public class Person : IDisposable

{

public string Name { get; set; }

public void Dispose()

{

Console.WriteLine("Disposed");

}

}

}

Консольный вывод:

Некоторые действия с объектом Person. Получим его имя: Tom

Disposed

Конец метода Test

Здесь мы видим, что по завершении блока using у объекта Person вызывается метод Dispose. Вне блока кода using объект p типа Person не существует.

Начиная с версии C# 8.0 мы можем задать в качестве области действия всю окружающую область видимости, например, метод:

private static void Test()

{

using Person p = new Person { Name = "Tom" };

// переменная p доступна до конца метода Test

Console.WriteLine($"Некоторые действия с объектом Person. Получим его имя: {p.Name}");

Console.WriteLine("Конец метода Test");

}

В данном случае using сообщает компилятору, что объявляемая переменная должна быть удалена в конце области видимости - то есть в конце метода Test. Соответственно мы получим следующий консольный вывод:

Некоторые действия с объектом Person. Получим его имя: Tom

Конец метода Test

Disposed

## ► Чтение и запись текстовых файлов. StreamReader и StreamWriter

### Запись в файл и StreamWriter

Для записи в текстовый файл используется класс **StreamWriter**. Некоторые из его конструкторов, которые могут применяться для создания объекта StreamWriter:

* StreamWriter(string path): через параметр path передается путь к файлу, который будет связан с потоком
* StreamWriter(string path, bool append): параметр append указывает, надо ли добавлять в конец файла данные или же перезаписывать файл. Если равно true, то новые данные добавляются в конец файла. Если равно false, то файл перезаписываетсяя заново
* StreamWriter(string path, bool append, System.Text.Encoding encoding): параметр encoding указывает на кодировку, которая будет применяться при записи

Свою функциональность StreamWriter реализует через следующие методы:

* int Close(): закрывает записываемый файл и освобождает все ресурсы
* void Flush(): записывает в файл оставшиеся в буфере данные и очищает буфер.
* Task FlushAsync(): асинхронная версия метода Flush
* void Write(string value): записывает в файл данные простейших типов, как int, double, char, string и т.д. Соответственно имеет ряд перегруженных версий для записи данных элементарных типов, например, Write(char value), Write(int value), Write(double value) и т.д.
* Task WriteAsync(string value): асинхронная версия метода Write
* void WriteLine(string value): также записывает данные, только после записи добавляет в файл символ окончания строки
* Task WriteLineAsync(string value): асинхронная версия метода WriteLine

Рассмотрим запись в файл на примере:

using System;

using System.IO;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

string writePath = @"C:\SomeDir\hta.txt";

string text = "Привет мир!\nПока мир...";

try

{

using (StreamWriter sw = new StreamWriter(writePath, false, System.Text.Encoding.Default))

{

sw.WriteLine(text);

}

using (StreamWriter sw = new StreamWriter(writePath, true, System.Text.Encoding.Default))

{

sw.WriteLine("Дозапись");

sw.Write(4.5);

}

Console.WriteLine("Запись выполнена");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

}

В данном случае два раза создаем объект StreamWriter. В первом случае если файл существует, то он будет перезаписан. Если не существует, он будет создан. И в нее будет записан текст из переменной text. Во втором случае файл открывается для дозаписи, и будут записаны атомарные данные - строка и число. В обоих случаях будет использоваться кодировка по умолчанию.

По завершении программы в папке C://SomeDir мы сможем найти файл hta.txt, который будет иметь следующие строки:

Привет мир!

Пока мир...

Дозапись

4,5

### Поскольку операции с файлами могут занимать продолжительное время, то в общем случае рекомендуется использовать асинхронную запись.

using System;

using System.IO;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static async Task Main(string[] args)

{

string writePath = @"C:\SomeDir\hta2.txt";

string text = "Привет мир!\nПока мир...";

try

{

using (StreamWriter sw = new StreamWriter(writePath, false, System.Text.Encoding.Default))

{

await sw.WriteLineAsync(text);

}

using (StreamWriter sw = new StreamWriter(writePath, true, System.Text.Encoding.Default))

{

await sw.WriteLineAsync("Дозапись");

await sw.WriteAsync("4,5");

}

Console.WriteLine("Запись выполнена");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

}

### Чтение из файла и StreamReader

Класс StreamReader позволяет нам легко считывать весь текст или отдельные строки из текстового файла.

Некоторые из конструкторов класса StreamReader:

* StreamReader(string path): через параметр path передается путь к считываемому файлу
* StreamReader(string path, System.Text.Encoding encoding): параметр encoding задает кодировку для чтения файла

Среди методов StreamReader можно выделить следующие:

* void Close(): закрывает считываемый файл и освобождает все ресурсы
* int Peek(): возвращает следующий доступный символ, если символов больше нет, то возвращает -1
* int Read(): считывает и возвращает следующий символ в численном представлении. Имеет перегруженную версию: Read(char[] array, int index, int count), где array - массив, куда считываются символы, index - индекс в массиве array, начиная с которого записываются считываемые символы, и count - максимальное количество считываемых символов
* Task<int> ReadAsync(): асинхронная версия метода Read
* string ReadLine(): считывает одну строку в файле
* string ReadLineAsync(): асинхронная версия метода ReadLine
* string ReadToEnd(): считывает весь текст из файла
* string ReadToEndAsync(): асинхронная версия метода ReadToEnd

Сначала считаем текст полностью из ранее записанного файла:

using System;

using System.IO;

using System.Threading.Tasks;

namespace HelloApp

{

class Program

{

static async Task Main(string[] args)

{

string path = @"C:\SomeDir\hta.txt";

try

{

using (StreamReader sr = new StreamReader(path))

{

Console.WriteLine(sr.ReadToEnd());

}

// асинхронное чтение

using (StreamReader sr = new StreamReader(path))

{

Console.WriteLine(await sr.ReadToEndAsync());

}

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

}

Считаем текст из файла построчно:

string path = @"C:\SomeDir\hta.txt";

using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))

{

string line;

while ((line = sr.ReadLine()) != null)

{

Console.WriteLine(line);

}

}

// асинхронное чтение

using (StreamReader sr = new StreamReader(path, System.Text.Encoding.Default))

{

string line;

while ((line = await sr.ReadLineAsync()) != null)

{

Console.WriteLine(line);

}

}

В данном случае считываем построчно через цикл while: while ((line = sr.ReadLine()) != null) - сначала присваиваем переменной line результат функции sr.ReadLine(), а затем проверяем, не равна ли она null. Когда объект sr дойдет до конца файла и больше строк не останется, то метод sr.ReadLine() будет возвращать null.

**Вопросы**

**1. Что такое обобщение (generic)?**

Термин **обобщение**, по существу, означает параметризированный тип. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра. 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, как, например, обобщенный класс или обобщенный метод.

**2. Пусть дан фрагмент листинга. В какой строчке содержится ошибка?**

****

Ошибки нет.

class Gen<T, G>

{

G ob;

T bo;

public Gen(G o) { ob = o; }

public T GetOb() { return bo; }

}

**3. Как можно наложить определенное ограничение на параметр?**

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 { // ...**

где *ограничения* указываются списком через запятую.

В качестве ограничений мы можем использовать следующие типы:

* Классы
* Интерфейсы
* class - универсальный параметр должен представлять класс
* struct - универсальный параметр должен представлять структуру
* new() - универсальный параметр должен представлять тип, который имеет общедоступный (public) конструктор без параметров

**4. Как можно наложить несколько ограничений на параметр?**

Перечислить ограничения через запятую.

Если для универсального параметра задано несколько ограничений, то они должны идти в определенном порядке:

1. Название класса, class, struct. Причем мы можем одновременно определить *только одно* из этих ограничений
2. Название интерфейса
3. new()

Например, следующее объявление считается вполне допустимым.

**class Gen<T> where Т : MyClass, IMylnterface, new() {**  
**// ...**

В данном случае параметр типа Т должен быть заменен аргументом типа, наследующим от классаMyClass, реализующим интерфейсIMylnterfaceи использующим конструктор без параметра.

**5. Перечислите все существующие ограничения на типы данных обобщения?**

(выше в ответах и заданиях--)

**6. Какое ограничение на тип задано в следующем фрагменте листинга?**

class A { }

class В : A { }

class С { }

class Test<T> where T : A { }

Ограничение, суть которого в том, что значением параметра T может быть лишь класс A или его наследники (то есть B).

**7. Какое ограничение на тип задано в следующем фрагменте листинга?**

interface A { }

class Test<T> where T : class { }

Ограничение, суть которого в том, что значением параметра T может быть лишь класс (любой).

**8. Какое ограничение на тип задано в следующем фрагменте листинга?**

interface A { }

class Test<T> where T : struct { }

Ограничение, суть которого в том, что значением параметра T может быть лишь структура (любая).

**9. Приведите примеры, когда обобщенный класс может действовать как базовый или производный класс.**

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.

Ниже представлен пример создания сложной иерархии обобщенных классов:

using System;

using System.Collections.Generic;

using System;

namespace ConsoleApplication1

{

class Class1<T>

{

}

// Унаследованный обобщенный класс

class Class2\_1<T> : Class1<T>

{ }

// Ещё один унаследованный класс с собственными параметрами

class Class2\_2<T, V> : Class1<T>

{ }

class Class3<T, V, E, G> : Class2\_2<T, V>

{ }

// Обычный необобщенный класс

class SomeClass

{ }

// Унаследованный от обычного класса обобщенный класс

class ObClass<T> : SomeClass

{ }

class Program

{

static void Main()

{ }

}

}

**Параметр типа Т указывается в объявлении класса!**

**▪ Обобщенный базовый класс**

**ПРИМЕР:**

// Простая иерархия обобщенных классов,

using System;

// Обобщенный базовый класс,

class Gen<T>

{

T ob;

public Gen(T о)

{

ob = о;

}

// Возвратить значение переменной ob.

public T GetOb()

{

return ob;

}

}

// Класс, производный от класса Gen.

class Gen2<T> : Gen<T>

{

public Gen2(T o) : base(o)

{

// ...

}

}

class GenHierDemo

{

static void Main()

{

Gen2<string> g2 = new Gen2<string>("Привет");

Console.WriteLine(g2.GetOb());

}

}

В этой иерархии класс Gen2 наследует от обобщенного класса Gen. Обратите внимание на объявление класса Gen2 в следующей строке кода.

***class Gen2<T> : Gen<T>***

Параметр типа Т указывается в объявлении класса Gen2 и в то же время передается классу Gen. Это означает, что любой тип, передаваемый классу Gen2, будет передаваться также классу Gen. Например, в следующем объявлении:

***Gen2<string> g2 = new Gen2<string>("Привет");***

параметр типа string передается классу Gen. Поэтому переменная ob в той части класса Gen2, которая относится к классу Gen, будет иметь тип string.

Обратите также внимание на то, что в классе Gen2 параметр типа Т не используется, а только передается вверх по иерархии базовому классу Gen. Это означает, что в производном классе следует непременно указывать параметры типа, требующиеся его обобщенному базовому классу, даже если этот производный класс необязательно должен быть обобщенным.

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

**▪ Обобщенный производный класс**

Необобщенный класс может быть вполне законно базовым для обобщенного производного класса. В качестве примера рассмотрим следующую программу.

// Пример необобщенного класса в качестве базового для

// обобщенного производного класса.

using System;

// Необобщенный базовый класс,

class NonGen

{

int num;

public NonGen(int i)

{

num = i;

}

public int GetNum()

{

return num;

}

}

// Обобщенный производный класс,

class Gen<T> : NonGen

{

T ob;

public Gen(T o, int i) : base(i)

{

ob = o;

}

// Возвратить значение переменной ob.

public T GetOb()

{

return ob;

}

}

// Создать объект класса Gen.

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());

}

}

**10. В каких случаях в обобщениях может использоваться оператор default?**

(где-то в заданиях--)

**11. Поясните, как использовать статические переменные в обобщенных классах.**

(где-то в заданиях--)

**12. Приведите пример обобщенного интерфейса.**

Если в обобщенном классе (структуре) используется статическое поле (static), то в объекте каждого конкретного типа (int, double и т.д.) создается уникальная копия этого поля. То есть, нет единого статического поля для всех объектов различных типов, которые конструируются.

**13. В чем отличие обобщенных классов от обобщенных структур?**

Подобно классам, структуры также могут быть обобщенными. Они очень похожи на обобщенные классы, но нужно учитывать особенности обычных структур, по сравнению с обычными классами.

**14. Какие классы для работы с файлами вы знаете? Приведите пример**

(где-то в заданиях--)