# Обобщения

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.

Следует особо подчеркнуть, что в C# всегда имелась возможность создавать обобщенный код, оперируя ссылками типа object. А поскольку класс object является базовым для всех остальных классов, то по ссылке типа object можно обращаться к объекту любого типа. Таким образом, до появления обобщений для оперирования разнотипными объектами в программах служил обобщенный код, в котором для этой цели использовались ссылки типа object.

Но дело в том, что в таком коде трудно было соблюсти типовую безопасность, поскольку для преобразования типа object в конкретный тип данных требовалось приведение типов. А это служило потенциальным источником ошибок из-за того, что приведение типов могло быть неумышленно выполнено неверно. Это затруднение позволяют преодолеть обобщения, обеспечивая типовую безопасность, которой раньше так недоставало. Кроме того, обобщения упрощают весь процесс, поскольку 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.

Давайте рассмотрим основные преимущества использования обобщений:

**Производительность**

Одним из основных преимуществ обобщений является производительность. Использование типов значений с необобщенными классами коллекций вызывает упаковку (boxing) и распаковку (unboxing) при преобразовании в ссылочный тип и обратно.

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

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

**Безопасность**

Другим свойством обобщений является безопасность типов. Обобщения автоматически обеспечивают типовую безопасность всех операций. В ходе выполнения этих операций обобщения исключают необходимость обращаться к приведению типов и проверять соответствие типов в коде вручную.

**Повторное использование двоичного кода**

Обобщения повышают степень повторного использования двоичного кода. Обобщенный класс может быть определен однажды, и на его основе могут быть созданы экземпляры многих типов.

**"Разбухание" кода**

Насколько много кода генерируется при создании экземпляров конкретных типов из обобщений? Поскольку определение обобщенного класса включается в сборку, создание на его основе конкретных классов специфических типов не приводит к дублированию кода в IL.

C выходом версии 2.0 фреймворк .NET стал поддерживать 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:

|  |
| --- |
| 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:

|  |
| --- |
| class Account  {  public object Id { get; set; }  public int Sum { get; set; }  } |

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

|  |
| --- |
| 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:

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

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

|  |
| --- |
| 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:

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

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

Эти проблемы были призваны устранить обобщенные типы. Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Поэтому определим класс Account как обощенный:

|  |
| --- |
| 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 в 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 или любой другой класс или структура:

|  |
| --- |
| 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 типизируется типами int и string:

|  |
| --- |
| 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:

|  |
| --- |
| 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.

Когда для класса Acount указывается аргумент типа, например int или string, то создается так называемый в C# ***закрыто сконструированный тип***. В частности, Acount<int> является закрыто сконструированным типом. Ведь, по существу, такой обобщенный тип, как Acount<T>, является абстракцией. И только после того, как будет сконструирован конкретный вариант, например Acount<int>, создается конкретный тип. А конструкция, подобная Acount<T>, называется в C# ***открыто сконструированным типом***, поскольку в ней указывается параметр типа Т, но не такой конкретный тип, как int.

В C# чаще определяются такие понятия, как открытый и закрытый типы. *Открытым типом* считается такой параметр типа или любой обобщенный тип, для которого аргумент типа является параметром типа или же включает его в себя. А любой тип, не относящийся к открытому, считается закрытым. *Сконструированным типом* считается такой обобщенный тип, для которого предоставлены все аргументы типов. Если все эти аргументы относятся к закрытым типам, то такой тип считается закрыто сконструированным. А если один или несколько аргументов типа относятся к открытым типам, то такой тип считается открыто сконструированным.

И еще один момент. В связи с изложенным выше возникает следующий резонный вопрос: если аналогичные функциональные возможности обобщенного класса Acount можно получить и без обобщений, просто указав объект как тип данных и выполнив надлежащее приведение типов, то какая польза от того, что класс Acount делается обобщенным? Ответ на этот вопрос заключается в том, что обобщения автоматически обеспечивают типовую безопасность всех операций, затрагивающих класс Acount. В ходе выполнения этих операций обобщения исключают необходимость обращаться к приведению типов и проверять соответствие типов в коде вручную.

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:

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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 переменная session.

Использование нескольких универсальных параметров

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

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| class Program  {  private static void Main(string[] args)  {  int x = 7;  int y = 25;  Swap<int>(ref x, ref y);  Console.WriteLine($"x={x} y={y}"); // x=25 y=7  string s1 = "hello";  string s2 = "bye";  Swap<string>(ref s1, ref s2);  Console.WriteLine($"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 обощенный 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.

# Ограничения обобщений

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

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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 C# предусмотрен ряд ограничений на типы данных:

**Ограничение на базовый класс**

Требует наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. 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**

Требует реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.

**Ограничение на конструктор**

Требует предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().

**Ограничение ссылочного типа**

Требует указывать аргумент ссылочного типа с помощью оператора class.

**Ограничение типа значения**

Требует указывать аргумент типа значения с помощью оператора struct.

Среди всех этих ограничений чаще всего применяются ограничения на базовый класс и интерфейс, хотя все они важны в равной степени.

* 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

Ограничение на базовый класс

Ограничение на базовый класс позволяет указывать базовый класс, который должен наследоваться аргументом типа. Ограничение на базовый класс служит двум главным целям:

* Оно позволяет использовать в обобщенном классе те члены базового класса, на которые указывает данное ограничение. Это дает, например, возможность вызвать метод или обратиться к свойству базового класса. В отсутствие ограничения на базовый класс компилятору ничего не известно о типе членов, которые может иметь аргумент типа. Накладывая ограничение на базовый класс, вы тем самым даете компилятору знать, что все аргументы типа будут иметь члены, определенные в этом базовом классе.
* Ограничение на базовый класс гарантирует использование только тех аргументов типа, которые поддерживают указанный базовый класс. Это означает, что для любого ограничения, накладываемого на базовый класс, аргумент типа должен обозначать сам базовый класс или производный от него класс. Если же попытаться использовать аргумент типа, не соответствующий указанному базовому классу или не наследующий его, то в результате возникнет ошибка во время компиляции.

Ниже приведена общая форма наложения ограничения на базовый класс, в которой используется оператор where:

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.

Пример такого ограничения мы рассмотрели выше мы рассмотрели выше.

Применение ограничения 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. Это означает, что для любого ограничения, накладываемого на интерфейс, аргумент типа должен обозначать сам интерфейс или же тип, реализующий этот интерфейс.

Ниже приведена общая форма наложения ограничения на интерфейс, в которой используется оператор where:

where T : имя\_интерфейса

где T — это имя параметра типа, а имя\_интерфейса — конкретное имя ограничиваемого 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.

Применение ограничения new() на конструктор

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, если явно определяемый конструктор отсутствует или же конструктор без параметров явно объявлен пользователем. Накладывая ограничение new(), можно вызывать конструктор без параметров для создания объекта.

|  |
| --- |
| class Transaction<T> where T : 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. Иными словами, передавать аргументы конструктору параметра типа не разрешается.
* 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. Эти ограничения оказываются полезными в тех случаях, когда для обобщенного кода важно провести различие между ссылочным типом и типом значения. Ниже приведена общая форма ограничения ссылочного типа:

where T : class

В этой форме с оператором where ключевое слово class указывает на то, что аргумент T должен быть ссылочного типа. Следовательно, всякая попытка использовать тип значения, например int или bool, вместо T приведет к ошибке во время компиляции. Ниже приведена общая форма ограничения типа значения:

where T : struct

В этой форме ключевое слово struct указывает на то, что аргумент T должен быть типа значения. (Напомним, что структуры относятся к типам значений.) Следовательно, всякая попытка использовать ссылочный тип, например string, вместо T приведет к ошибке во время компиляции. Но если имеются дополнительные ограничения, то в любом случае class или struct должно быть первым по порядку накладываемым ограничением.

Как правило, пустое значение нельзя присвоить переменной типа значения. (Исключением из этого правила является обнуляемый тип, который представляет собой специальный тип структуры, инкапсулирующий тип значения и допускающий пустое значение (null).) Следовательно, в отсутствие ограничения такое присваивание было бы недопустимым, и код не подлежал бы компиляции. Это один из тех случаев, когда для обобщенного кода может оказаться очень важным различие между типами значений и ссылочными типами.

Ограничение типа значения является дополнением ограничения ссылочного типа. Оно просто гарантирует, что любой аргумент, обозначающий тип, должен быть типа значения, в том числе struct и enum.

Давайте рассмотрим пример:

|  |
| --- |
| struct UI  {  public UI(string Name, int Age)  {  this.Name = Name;  this.Age = Age;  }  public string Name;  public int Age;  }  // Обобщенный класс, использующий ограничение на тип значения  class UserInfo<T> where T : struct  {  T obj;  public UserInfo(T ob)  {  obj = ob;  }  }  class Program  {  static void Main()  {  UI user1 = new UI(Name: "Alexandr", Age: 26);  UserInfo<UI> user = new UserInfo<UI>(user1);  }  } |

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

* Название класса, class, struct. Причем мы можем одновременно определить только одно из этих ограничений
* Название интерфейса

new()

Использование нескольких универсальных параметров

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

|  |
| --- |
| class Transaction<U, V>  where U : Account<int>  where V : struct  {  } |

Ограничения методов

Подобным образом можно использовать и ограничения методов:

|  |
| --- |
| 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>.

Обобщёнными, помимо классов и методов, также могут быть:

* Структуры
* Делегаты
* интерфейсы

# Наследование обобщённых 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:

|  |
| --- |
| 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:

|  |
| --- |
| class UniversalAccount<T> : **Account<T>**  {  public UniversalAccount(T id) : base(id)  {  }  } |

Применение класса:

|  |
| --- |
| 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 при наследовании у базового класса надо явным образом определить используемый тип:

|  |
| --- |
| class StringAccount : Account<string>  {  public StringAccount(string id) : base(id)  {  }  } |

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

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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.

# Коллекции

В C# коллекция представляет собой совокупность объектов. В среде .NET Framework имеется немало интерфейсов и классов, в которых определяются и реализуются различные типы коллекций. Коллекции упрощают решение многих задач программирования благодаря тому, что предлагают готовые решения для создания целого ряда типичных, но порой трудоемких для разработки структур данных. Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц. Коллекции являются современным технологическим средством, заслуживающим пристального внимания всех, кто программирует на C#.

Первоначально существовали только классы необобщенных коллекций. Но с внедрением обобщений в версии C# 2.0 среда .NET Framework была дополнена многими новыми обобщенными классами и интерфейсами. Благодаря введению обобщенных коллекций общее количество классов и интерфейсов удвоилось. Вместе с библиотекой распараллеливания задач (TPL) в версии 4.0 среды .NET Framework появился ряд новых классов коллекций, предназначенных для применения в тех случаях, когда доступ к коллекции осуществляется из нескольких потоков. Нетрудно догадаться, что прикладной интерфейс Collections API составляет значительную часть среды .NET Framework.

Краткий обзор коллекций

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе ArrayList, Hashtable, Stack и Queue, могут применяться в исходном виде и без каких-либо изменений. Имеется также возможность реализовать собственную коллекцию, хотя потребность в этом возникает крайне редко.

В среде .NET Framework поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные.

**Необобщенные коллекции**

Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". В отношении необобщенных коллекций важно иметь в виду следующее: они оперируют данными типа object. Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных. Очевидно, что такие коллекции не типизированы, поскольку в них хранятся ссылки на данные типа object. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен System.Collections.

**Специальные коллекции**

Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен System.Collections.Specialized.

**Поразрядная коллекция**

В прикладном интерфейсе Collections API определена одна коллекция с поразрядной организацией — это BitArray. Коллекция типа BitArray поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например И, ИЛИ, исключающее ИЛИ, а следовательно, она существенно отличается своими возможностями от остальных типов коллекций. Коллекция типа BitArray объявляется в пространстве имен System.Collections.

**Обобщенные коллекции**

Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен System.Collections.Generic.

**Параллельные коллекции**

Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен System.Collections.Concurrent.

В пространстве имен System.Collections.ObjectModel находится также ряд классов, поддерживающих создание пользователями собственных обобщенных коллекций.

Основополагающим для всех коллекций является понятие перечислителя, который поддерживается в необобщенных интерфейсах IEnumerator и IEnumerable, а также в обобщенных интерфейсах IEnumerator<T> и IEnumerable<T>. Перечислитель обеспечивает стандартный способ поочередного доступа к элементам коллекции. Следовательно, он перечисляет содержимое коллекции. В каждой коллекции должна быть реализована обобщенная или необобщенная форма интерфейса IEnumerable, поэтому элементы любого класса коллекции должны быть доступны посредством методов, определенных в интерфейсе IEnumerator или IEnumerator<T>. Это означает, что, внеся минимальные изменения в код циклического обращения к коллекции одного типа, его можно использовать для аналогичного обращения к коллекции другого типа. Любопытно, что для поочередного обращения к содержимому коллекции в цикле foreach используется перечислитель.

С перечислителем непосредственно связано другое средство, называемое итератором. Это средство упрощает процесс создания классов коллекций, например специальных, поочередное обращение к которым организуется в цикле foreach.

Рассмотрим создание и применение двух коллекций:

|  |
| --- |
| using System;  using System.Collections;  using System.Collections.Generic;  namespace Collections  {  class Program  {  static void Main(string[] args)  {              // необобщенная коллекция ArrayList              ArrayList objectList = new ArrayList() { 1, 2, "string", 'c', 2.0f };  object obj = 45.8;  objectList.Add(obj);  objectList.Add("string2");  objectList.RemoveAt(0); // удаление первого элемента              foreach (object o in objectList)  {  Console.WriteLine(o);  }  Console.WriteLine("Общее число элементов коллекции: {0}", objectList.Count);              // обобщенная коллекция List              List<string> countries = new List<string>() { "Россия", "США", "Великобритания", "Китай" };  countries.Add("Франция");  countries.RemoveAt(1); // удаление второго элемента              foreach (string s in countries)  {  Console.WriteLine(s);  }  Console.ReadLine();  }  }  } |

Здесь используются две коллекции: необобщенная - ArrayList и обобщенная - List. Большинство коллекций поддерживают добавление элементов. Например, в данном случае добавление производится методом Add, но для других коллекций название метода может отличаться. Также большинство коллекций реализуют удаление (в данном примере производится с помощью метода RemoveAt).

С помощью свойства Count у коллекций можно посмотреть количество элементов.

И так как коллекции реализуют интерфейс IEnumerable/IEnumerable<T>, то все они поддерживают перебор в цикле foreach.

Конкретные методы и способы использования могут различаться от одного класса коллекции к другому, но общие принципы будут одни и те же для всех классов коллекций.

Необобщённые коллекции

Необобщенные или простые коллекции определены в пространстве имен System.Collections. Их особенность состоит в том, что их функциональность, функциональные возможности описываются в интерфейсах, которые также находятся в этом пространстве имен.

Рассмотрим основные интерфейсы:

* **IEnumerable**: определяет метод GetEnumerator. Данный метод возвращает перечислитель - то есть некоторый объект, реализующий интерфейс IEnumerator.
* **IEnumerator**: реализация данного интерфейса позволяет перебирать элементы коллекции с помощью цикла foreach
* **ICollection**: является основой для всех необобщенных коллекций, определяет основные методы и свойства для всех необобщенных коллекций (например, метод CopyTo и свойство Count). Данный интерфейс унаследован от интерфейса IEnumerable, благодаря чему базовый интерфейс также реализуется всеми классами необобщенных коллекций
* **IList**: позволяет получать элементы коллекции по порядку. Также определяет ряд методов для манипуляции элементами: Add(добавление элементов), Remove/RemoveAt (удаление элемента) и ряд других.
* **IComparer**: определяет метод int Compare(object x, object y) для сравнения двух объектов
* **IDictionary**: определяет поведение коллекции, при котором она должна хранить объекты в виде пар ключ-значение: для каждого объекта определяется уникальный ключ, и этому ключу соответствует определенное значение
* **IDictionaryEnumerator**: определяет методы и свойства для перечислителя словаря
* **IEqualityComparer**: определяет два метода Equals и GetHashCode, с помощью которых два объекта сравниваются на предмет равенства
* **IStructuralComparer**: определяет метод Compare для структурного сравнения двух объектов: при таком сравнении сравниваются не ссылки на объекты, а непосредственное содержимое объектов
* **IStructuralEquatable**: позволяет провести структурное равенство двух объектов. Как и в случае с интерфейсом IStructuralComparer сравнивается содержимое двух объектов

Эти интерфейсы реализуются следующими классами коллекций в пространстве имен System.Collections:

* **ArrayList**: класс простой коллекции объектов. Реализует интерфейсы IList, ICollection, IEnumerable
* **BitArray**: класс коллекции, содержащей массив битовых значений. Реализует интерфейсы ICollection, IEnumerable
* **Hashtable**: класс коллекции, представляющей хэш-таблицу и храняющий набор пар "ключ-значение"
* **Queue**: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable
* **SortedList**: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection, IDictionary, IEnumerable
* **Stack**: класс стека

Обобщённые коллекции

В теме Обобщенные типы мы уже говорили про обобщения: как создавать и использовать обобщенные классы, роль универсального параметра T. Обобщенные коллекции - это те же самые обобщенные классы. И опять же их использование перед необобщенными коллекциями имеет те же преимущества: повышение производительности (не надо тратить время на упаковку и распаковку объекта) и повышенная типобезопасность.

Классы обобщенных коллекций находятся в пространстве имен **System.Collections.Generic**. Функционал коллекций также по большей части описывается в обобщенных интерфейсах.

Только интерфейсы обобщенных коллекций отличаются от необобщеных двойников не только наличием универсального параметра T, но и самой функциональностью. Рассмотрим основные интерфейсы обобщенных коллекций:

* **IEnumerable<T>:** определяет метод GetEnumerator, с помощью которого можно получать элементы любой коллекции

. Реализация данного интерфейса позволяет перебирать элементы коллекции с помощью цикла foreach

* **IEnumerator<T>:** определяет методы, с помощью которых потом можно получить содержимое коллекции по очереди
* **ICollection<T>:** представляет ряд общих свойств и методов для всех обобщенных коллекций (например, методы CopyTo, Add, Remove, Contains, свойство Count)
* **IList<T>:** предоставляет функционал для создания последовательных списков
* **IComparer<T>:** определяет метод Compare для сравнения двух однотипных объектов
* **IDictionary<TKey, TValue>:** определяет поведение коллекции, при котором она должна хранить объекты в виде пар ключ-значение: для каждого объекта определяется уникальный ключ типа, указанного в параметре TKey, и этому ключу соответствует определенное значение, имеющее тип, указанный в параметре TValue
* **IEqualityComparer<T>:** определяет методы, с помощью которых два однотипных объекта сравниваются на предмет равенства

Эти интерфейсы реализуются следующими классами коллекций в пространстве имен System.Collections.Generic:

* **List<T>:** класс, представляющий последовательный список. Реализует интерфейсы IList<T>, ICollection<T>, IEnumerable<T>
* **Dictionary<TKey, TValue>:** класс коллекции, хранящей наборы пар "ключ-значение". Реализует интерфейсы ICollection<T>, IEnumerable<T>, IDictionary<TKey, TValue>
* **LinkedList<T>:** класс двухсвязанного списка. Реализует интерфейсы ICollection<T> и IEnumerable<T>
* **Queue<T>:** класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable<T>
* **SortedSet<T>:** класс отсортированной коллекции однотипных объектов. Реализует интерфейсы ICollection<T>, ISet<T>, IEnumerable<T>
* **SortedList<TKey, TValue>:** класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection<T>, IEnumerable<T>, IDictionary<TKey, TValue>
* **SortedDictionary<TKey, TValue>:** класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. В общем похож на класс SortedList<TKey, TValue>, основные отличия состоят лишь в использовании памяти и в скорости вставки и удаления
* **Stack<T>:** класс стека однотипных объектов. Реализует интерфейсы ICollection<T> и IEnumerable<T>

Большинство обобщенных классов коллекций дублируют необобщенные классы коллекций. Но если вам не надо хранить объекты разных типов, то предпочтительнее использовать обобщенные коллекции.

Класс List<T> представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

* **void Add(T item):** добавление нового элемента в список
* **void AddRange(ICollection collection):** добавление в список коллекции или массива
* **int BinarySearch(T item):** бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
* **int IndexOf(T item):** возвращает индекс первого вхождения элемента в списке
* **void Insert(int index, T item):** вставляет элемент item в списке на позицию index
* **bool Remove(T item):** удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true
* **void RemoveAt(int index):** удаление элемента по указанному индексу index
* **void Sort():** сортировка списка

### Список List<T>

Посмотрим реализацию списка на примере:

|  |
| --- |
| using System;  using System.Collections.Generic;    namespace Collections  {  class Program  {  static void Main(string[] args)  {  List<int> numbers = new List<int>() { 1, 2, 3, 45 };  numbers.Add(6); // добавление элемента  numbers.AddRange(new int[] { 7, 8, 9 });  numbers.Insert(0, 666); // вставляем на первое место в списке число 666  numbers.RemoveAt(1); // удаляем второй элемент  foreach (int i in numbers)  {  Console.WriteLine(i);  }  List<Person> people = new List<Person>(3);  people.Add(new Person() { Name = "Том" });  people.Add(new Person() { Name = "Билл" });  foreach (Person p in people)  {  Console.WriteLine(p.Name);  }  Console.ReadLine();  }  }  class Person  {  public string Name { get; set; }  }  } |

Здесь у нас создаются два списка: один для объектов типа int, а другой - для объектов Person. В первом случае мы выполняем начальную инициализацию списка: List<int> numbers = new List<int>() { 1, 2, 3, 45 };

Во втором случае мы используем другой конструктор, в который передаем начальную емкость списка: List<Person> people = new List<Person>(3);. Указание начальной емкости списка (capacity) позволяет в будущем увеличить производительность и уменьшить издержки на выделение памяти при добавлении элементов. Также начальную емкость можно установить с помощью свойства Capacity, которое имеется у класса List.

### Двухсвязный список LinkedList<T>

Класс LinkedList<T> представляет двухсвязный список, в котором каждый элемент хранит ссылку одновременно на следующий и на предыдущий элемент.

Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел представляет объект класса LinkedListNode<T>. Этот класс имеет следующие свойства:

* **Value**: само значение узла, представленное типом T
* **Next**: ссылка на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null
* **Previous**: ссылка на предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null

Используя методы класса LinkedList<T>, можно обращаться к различным элементам, как в конце, так и в начале списка:

* **AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode):** вставляет узел newNode в список после узла node.
* **AddAfter(LinkedListNode<T> node, T value):** вставляет в список новый узел со значением value после узла node.
* **AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode):** вставляет в список узел newNode перед узлом node.
* **AddBefore(LinkedListNode<T> node, T value):** вставляет в список новый узел со значением value перед узлом node.
* **AddFirst(LinkedListNode<T> node):** вставляет новый узел в начало списка
* **AddFirst(T value):** вставляет новый узел со значением value в начало списка
* **AddLast(LinkedListNode<T> node):** вставляет новый узел в конец списка
* **AddLast(T value):** вставляет новый узел со значением value в конец списка
* **RemoveFirst():** удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным
* **RemoveLast():** удаляет последний узел из списка

Посмотрим на двухсвязный список в действии:

|  |
| --- |
| using System;  using System.Collections.Generic;    namespace Collections  {  class Program  {  static void Main(string[] args)  {  LinkedList<int> numbers = new LinkedList<int>();  numbers.AddLast(1); // вставляем узел со значением 1 на последнее место  // так как в списке нет узлов, то последнее будет также и первым  numbers.AddFirst(2); // вставляем узел со значением 2 на первое место  numbers.AddAfter(numbers.Last, 3); // вставляем после последнего узла новый узел со значением 3  // теперь у нас список имеет следующую последовательность: 2, 1, 3  foreach (int i in numbers)  {  Console.WriteLine(i);  }  LinkedList<Person> persons = new LinkedList<Person>();  // добавляем persona в список и получим объект LinkedListNode<Person>, в котором хранится имя Tom  LinkedListNode<Person> tom = persons.AddLast(new Person() { Name = "Tom" });  persons.AddLast(new Person() { Name = "John" });  persons.AddFirst(new Person() { Name = "Bill" });  Console.WriteLine(tom.Previous.Value.Name); // получаем узел перед томом и его значение  Console.WriteLine(tom.Next.Value.Name); // получаем узел после тома и его значение  Console.ReadLine();  }  }  class Person  {  public string Name { get; set; }  }  } |

Здесь создаются и используются два списка: для чисел и для объектов класса Person. С числами, наверное, все более менее понятно. Разберем работу с классом Person.

Методы вставки (AddLast, AddFirst) при добавлении в список возвращают ссылку на добавленный элемент LinkedListNode<T> (в нашем случае LinkedListNode<Person>). Затем управляя свойствами Previous и Next, мы можем получить ссылки на предыдущий и следующий узлы в списке.

### Очередь Queue<T>

Класс Queue<T> представляет обычную очередь, работающую по алгоритму FIFO ("первый вошел - первый вышел").

У класса Queue<T> можно отметить следующие методы:

* **Dequeue**: извлекает и возвращает первый элемент очереди
* **Enqueue**: добавляет элемент в конец очереди
* **Peek**: просто возвращает первый элемент из начала очереди без его удаления

Посмотрим применение очереди на практике:

|  |
| --- |
| using System;  using System.Collections.Generic;    namespace Collections  {  class Program  {  static void Main(string[] args)  {  Queue<int> numbers = new Queue<int>();  numbers.Enqueue(3); // очередь 3  numbers.Enqueue(5); // очередь 3, 5  numbers.Enqueue(8); // очередь 3, 5, 8  // получаем первый элемент очереди  int queueElement = numbers.Dequeue(); //теперь очередь 5, 8  Console.WriteLine(queueElement);  Queue<Person> persons = new Queue<Person>();  persons.Enqueue(new Person() { Name = "Tom" });  persons.Enqueue(new Person() { Name = "Bill" });  persons.Enqueue(new Person() { Name = "John" });  // получаем первый элемент без его извлечения  Person pp = persons.Peek();  Console.WriteLine(pp.Name);  Console.WriteLine("Сейчас в очереди {0} человек", persons.Count);  // теперь в очереди Tom, Bill, John  foreach (Person p in persons)  {  Console.WriteLine(p.Name);  }  // Извлекаем первый элемент в очереди - Tom  Person person = persons.Dequeue(); // теперь в очереди Bill, John  Console.WriteLine(person.Name);  Console.ReadLine();  }  }  class Person  {  public string Name { get; set; }  }  } |

### Коллекция Stack<T>

Класс Stack<T> представляет коллекцию, которая использует алгоритм LIFO ("последний вошел - первый вышел"). При такой организации каждый следующий добавленный элемент помещается поверх предыдущего. Извлечение из коллекции происходит в обратном порядке - извлекается тот элемент, который находится выше всех в стеке.

В классе Stack можно выделить два основных метода, которые позволяют управлять элементами:

* **Push**: добавляет элемент в стек на первое место
* **Pop**: извлекает и возвращает первый элемент из стека
* **Peek**: просто возвращает первый элемент из стека без его удаления

Посмотрим на примере:

|  |
| --- |
| using System;  using System.Collections.Generic;    namespace Collections  {  class Program  {  static void Main(string[] args)  {  Stack<int> numbers = new Stack<int>();  numbers.Push(3); // в стеке 3  numbers.Push(5); // в стеке 5, 3  numbers.Push(8); // в стеке 8, 5, 3  // так как вверху стека будет находиться число 8, то оно и извлекается  int stackElement = numbers.Pop(); // в стеке 5, 3  Console.WriteLine(stackElement);  Stack<Person> persons = new Stack<Person>();  persons.Push(new Person() { Name = "Tom" });  persons.Push(new Person() { Name = "Bill" });  persons.Push(new Person() { Name = "John" });  foreach (Person p in persons)  {  Console.WriteLine(p.Name);  }  // Первый элемент в стеке  Person person = persons.Pop(); // теперь в стеке Bill, Tom  Console.WriteLine(person.Name);  Console.ReadLine();  }  }  class Person  {  public string Name { get; set; }  }  } |

### Коллекция Dictionary<T, V>

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Каждый такой объект является объектом структуры **KeyValuePair<TKey, TValue>.** Благодаря свойствам Key и Value, которые есть у данной структуры, мы можем получить ключ и значение элемента в словаре.

Рассмотрим на примере использование словарей:

|  |
| --- |
| Dictionary<int, string> countries = new Dictionary<int, string>(5);  countries.Add(1, "Russia");  countries.Add(3, "Great Britain");  countries.Add(2, "USA");  countries.Add(4, "France");  countries.Add(5, "China");  foreach (KeyValuePair<int, string> keyValue in countries)  {  Console.WriteLine(keyValue.Key + " - " + keyValue.Value);  }  // получение элемента по ключу  string country = countries[4];  // изменение объекта  countries[4] = "Spain";  // удаление по ключу  countries.Remove(2); |

Класс словарей также, как и другие коллекции, предоставляет методы Add и Remove для добавления и удаления элементов. Только в случае словарей в метод Add передаются два параметра: ключ и значение. А метод Remove удаляет не по индексу, а по ключу.

Так как в нашем примере ключами является объекты типа int, а значениями - объекты типа string, то словарь в нашем случае будет хранить объекты KeyValuePair<int, string>. В цикле foreach мы их можем получить и извлечь их них ключ и значение.

Кроме того, мы можем получить отдельно коллекции ключей и значений словаря:

|  |
| --- |
| Dictionary<char, Person> people = new Dictionary<char, Person>();  people.Add('b', new Person() { Name = "Bill" });  people.Add('t', new Person() { Name = "Tom" });  people.Add('j', new Person() { Name = "John" });  foreach (KeyValuePair<char, Person> keyValue in people)  {  // keyValue.Value представляет класс Person  Console.WriteLine(keyValue.Key + " - " + keyValue.Value.Name);  }  // перебор ключей  foreach (char c in people.Keys)  {  Console.WriteLine(c);  }  // перебор по значениям  foreach (Person p in people.Values)  {  Console.WriteLine(p.Name);  } |

Здесь в качестве ключей выступают объекты типа char, а значениями - объекты Person. Используя свойство Keys, мы можем получить ключи словаря, а свойство Values соответственно хранит все значения в словаре.

Для добавления необязательно применять метод Add(), можно использовать сокращенный вариант:

|  |
| --- |
| Dictionary<char, Person> people = new Dictionary<char, Person>();  people.Add('b', new Person() { Name = "Bill" });  people['a'] = new Person() { Name = "Alice" }; |

Несмотря на то, что изначально в словаре нет ключа 'a' и соответствующего ему элемента, то он все равно будет установлен. Если же он есть, то элемент по ключу 'a' будет заменен на новый объект new Person() { Name = "Alice" }

**Инициализация словарей**

В C# 5.0 мы могли инициализировать словари следующим образом:

|  |
| --- |
| Dictionary<string, string> countries = new Dictionary<string, string>  {  {"Франция", "Париж"},  {"Германия", "Берлин"},  {"Великобритания", "Лондон"}  };  foreach (var pair in countries)  Console.WriteLine("{0} - {1}", pair.Key, pair.Value); |

То начиная с C# 6.0 (Visual Studio 2015) доступен также еще один способ инициализации:

|  |
| --- |
| Dictionary<string, string> countries = new Dictionary<string, string>  {  ["Франция"] = "Париж",  ["Германия"] = "Берлин",  ["Великобритания"] = "Лондон"  }; |