Generic

Обобщение (Универсальные шаблоны) –элемент кода, способный адаптироваться для выполнения общих (сходных) действий над различными типами данных

> Универсальные шаблоны были добавлены в язык C# версии 2.0 и среду **CLR**. Эта возможность **CTS (CommonTypeSystem**—общая система типов), названа обобщениями (**generics)**

Преимущества обобщений

Обобщения позволяют создавать открытые **(open-ended)** типы, которые преобразуются в закрытые во время выполнения

Идентификатор **<T>**–это указатель места заполнения, вместо которого подставляется любой тип

```
class Types<T>
{
    T[] mass = new T[5];
}
```

Создание открытого типа

```
static void Main()
{
    Types<int> type = new Types<int>();
}
```

Закрытый тип

Каждый закрытый тип получает свою собственную копию набора статических полей

Значения по умолчанию

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение, в том числе и null. Но наприямую 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

Ограничения универсальных типов

При определении универсального типа можно ограничить виды типов, которые могут использоваться клиентским кодом в качестве аргументов типа при инициализации соответствующего класса. При попытке клиентского кода создать экземпляр класса с помощью типа, который не допускается ограничением, в результате возникает ошибка компиляции. Это называется ограничениями

Where

Предложение `where` используется в определении универсального типа для указания ограничений типов, которые могут использоваться в качестве аргументов параметра типа, определенного в универсальном объявлении

```
class MyClass<T> where T : struct
{
    //…
}
```

```
static void Main()
{
    MyClass<int> instance = new MyClass<int>();
    //MyClass<string> instance2 = new MyClass2<string>();
}
```

Ограничения определяются с помощью контекстно-зависимого ключевого слова `where`

Допустим, мы хотим, чтобы класс `Bank` хранил набор счетов, представленных объектами некоторого класса Account

```
class Account
{
    public int Id { get; private set;} // номер счета
    public Account(int _id)
    {
        Id = _id;
    }
}
```

Но у этого класса может быть много наследников: **DepositAccount** (депозитный счет), **DemandAccount** (счет до востребования) и т.д. Однако мы не можем знать, какой вид счета в банке в данном случае будет использоваться. Возможно, банк будет принимать только депозитные счета. И в этом случае в качестве универсального параметра можно установить тип Account:

```
class Bank<T> where T : Account
{
    T[] accounts;

    public Bank(T[] accs)
    {
        this.accounts = accs;
    }
    // вывод информации обо всех аккаунтах
    public void AccountsInfo()
    {
        foreach(Account acc in accounts)
        {
            Console.WriteLine(acc.Id);
        }
    }
}
```

С помощью выражения **`where T : Account`** мы указываем, что используемый тип T обязательно должен быть классом Account или его наследником

При этом мы можем задать множество ограничений через запятую

```
class Bank<T> where T : Client, Account
{}
```

Кроме того, можно указать ограничение, чтобы использовались только структуры

```
class Bank<T> where T : struct
{}
```

или классы:

```
class Bank<T> where T : class
{}
```

А также можно задать в качестве ограничения класс или структуру, которые реализуют конструктор по умолчанию с помощью слова **new**:

```
class Bank<T> where T : new()
{}
```

New

Ограничение `new()` указывает, что аргумент любого типа в объявлении общего класса должен иметь открытый конструктор без параметров

```
class MyClass<T> where T : new()
{
    public T instance = new T();
}
```

Использовать ограничение `new()` можно только в том случае, если тип не является абстрактным

Правила использования

`where T: struct`-Аргумент типа должен быть структурного типа, кроме `Nullable`.

`where T: class`-Аргумент типа должен иметь ссылочный тип; это так же распространяется на тип любого класса, интерфейса, делегата или массива.

`where T: <baseclassname>` - Аргумент типа должен являться или быть производным от указанного базового класса

`where T: U`-Аргумент типа, поставляемый для T, должен являться или быть производным от аргумента, поставляемого для U. Это называется неприкрытым ограничением типа

Использование нескольких универсальных параметров

Мы можем также задать сразу несколько универсальных параметров и ограничения к каждому из них:

```
class Operation<T, U>
    where U : class
    where T : Account, new() { }
```

Обобщенные методы

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

Наследование обобщенных типов

Один обобщенный класс может быть унаследован от другого обобщенного

При этом производный класс необязательно должен быть обобщенным

```
class Transaction<T>
{
    T inAccount;
    T outAccount;

    public Transaction(T inAcc, T outAcc)
    {
        inAccount = inAcc;
        outAccount = outAcc;
    }
}
class UniversalTransaction<T> : Transaction<T>
{
    public UniversalTransaction(T inAcc, T outAcc) : base(inAcc, outAcc)
    {

    }
}
```

```
class StringTransaction : Transaction<string>
{
    public StringTransaction(string inAcc, string outAcc)
            : base(inAcc, outAcc)
    {
    }
}
```

Boxing-Unboxing

Обобщения обеспечивают большую производительность, так как не происходит операции "упаковки-распаковки"(**Boxing-Unboxing)**

```
class MyClass<T>
{
    public T field;

    public void Method()
    {
        Console.WriteLine(field.GetType());
    }
}
```

```
static void Main()
{
    MyClass<int> instance1 = new MyClass<int>();
    MyClass<string> instance2 = new MyClass<string>();
    instance1.Method();
    instance2.Method();
}
```

Обобщения обеспечивают безопасность типов, так как могут содержать только типы, которые задаются при объявлении

Upcast

Ковариантность обобщений – **UpCast** параметров типов

Ковариантность обобщений в C# 4.0 ограничена интерфейсами и делегатами

```
class Animal { }
class Cat : Animal { }

delegate T MyDelegate<out T>();

static void Main()
{
    MyDelegate<Cat> delegateCat = () => new Cat();
    MyDelegate<Animal> delegateAnimal = delegateCat;

    Animal animal = delegateAnimal.Invoke();
    Console.WriteLine(animal.GetType().Name);
}
```

DownCast

Контрвариантность обобщений –**DownCast** параметров типов.

Контрвариантность обобщений в C# 4.0 ограничена интерфейсами и делегатами

```
class Animal { }
class Cat : Animal { }
delegate void MyDelegate<in T>(T a);

static void Main()
{
    MyDelegate<Animal> delegateAnimal = (Animal animal) =>
                    Console.WriteLine(animal.GetType().Name);

    MyDelegate<Cat> delegateCat = delegateAnimal;

    delegateAnimal(new Animal());
    delegateCat(new Cat());
}
```

Перегрузка обобщенных типов

Перегрузки обобщенных типов различаются количеством параметров типа, а не их именами

```
class MyClass<T>
{
    T[] mass = new T[5];
}
```

```
class MyClass<T,R>
{
    T[] mass = new T[5];
    R[] mass = new R[5];
}
```

- Правильная перегрузка: `MyClass<T>{ }, MyClass<T,R>{ }`
- Пример открытого типа: `MyClass<T>`
- Пример закрытого типа: `MyClass<int>`

Общие сведения

Общие сведения об универсальных шаблонах:

- Используйте универсальные типы для достижения максимального уровня повторного использования кода, безопасности типа и производительности.
- Наиболее частым случаем использования универсальных шаблонов является создание классов коллекции.
- Можно создавать собственные универсальные интерфейсы, классы, методы, события и делегаты.
- Доступ универсальных классов к методам можно ограничить определенными типами данных

Тип Nullable

Тип Nullable<T>представляет типы значений с пустыми (нулевыми) значениями.

```
int? a = null;
int? b = a + 4;
```

b = null

Фактически запись `?` является упрощенной формой использования структуры **System.Nullable<T>**. Параметр `T` в угловых скобках представляет универсальный параметр, вместо которого в конкретной задача уже подставляется конкретный тип данных

Также надо учитывать, что структура Nullable применяется только для типов значений, поскольку ссылочные типы итак могут иметь значение null

Чтобы получить доступ к значению Nullable, надо использовать свойство **Value**: `state.Value`. Чтобы проверить, имеется ли у структуры какое-либо значение, нужно применит свойство **HasValue**:

```
if(state.HasValue)
```

Структура Nullable с помощью метода `GetValueOrDefault()` позволяет использовать значение по умолчанию (для числовых типов это 0), если не значение не задано

При сравнении операндов один из которых `null`-результатом сравнения всегда будет –`false`

```
int? a = null;
int? b = -5;

if (a >= b)  // - false
{
    Console.WriteLine("a >= b");
}
else
{
    Console.WriteLine("a < b");
}
```

Операция поглощения

Оператор ?? называется оператором **null-объединения**. Он применяется для установки значений по умолчанию для типов значений и ссылочных типов, которые допускают значение null. Оператор ?? возвращает левый операнд, если этот операнд не равен `null`. Иначе возвращается правый операнд. При этом левый операнд должен принимать null

```
static void Main()
{
    int? a = null;
    int? b;

    b = a ?? 10; // b = 10

    a = 3;
    b = a ?? 10; // b = 3
}
```

Оператор **??** возвращает левый операнд, если он не `null` и правый операнд, еслилевый `null`

Если переменная не является nullable-типом и не может принимать значение null, то в качестве левого операнда в операции ?? она использоваться не может

Коллекции

ArrayList

`ArrayList`-коллекция с динамическим увеличением размера до нужного значения

```
static void Main()
{
    ArrayList arrayList = new ArrayList();

    arrayList.Add(1);
    arrayList.Add((object)2);
}
```

При добавлении элементов в коллекцию `ArrayList` ее емкость автоматически увеличивается нужным образом за счет перераспределения внутреннего массива

> Коллекция `ArrayList`-использует **boxing/unboxing,** поэтому не рекомендуется ее использовать в больших коллекциях

List<T>

`List<T>`(класс)-представляет строготипизированный список объектов, доступных по индексу

```
static void Main()
{
    List<int> list = new List<int>();

    list.Add(3);
    list.Add(7);
}
```

> Для работы с коллекциями Generic должно быть подключено пространство имен **System.Collections.Generic**

Dictionary<TKey, TValue>

Dictionary<TKey, TValue> -класс представляет коллекцию, которая работает по принципу -ключей и значений

```
static void Main()
{
    Dictionary<int, string> dictionary = new Dictionary<int, string>();

    dictionary.Add(0, "Ноль");
    dictionary.Add(1, "Один");
    dictionary.Add(2, "Два");
    dictionary.Add(3, "Три");
}
```