



# Web-разработка на C# и платформе Microsoft .NET

## Обобщения (Generics)

# Зачем нужны обобщения

---

Обобщения используются в ситуациях, когда поведение алгоритма не зависит от типа данных, с которым осуществляется работа.



# Зачем нужны обобщения

---

```
// Работает только с целочисленными массивами
static void WriteArr(int[] arr)
{
    foreach (var item in arr)
        Console.WriteLine(item);
}
```

```
// Работает с любыми массивами
static void WriteTArr<T>(T[] arr)
{
    foreach (var item in arr)
        Console.WriteLine(item);
}
```

# Использование обобщений

---

Пример:

```
int[] iarr = { 1, 2, 4, 5, 7 };  
WriteTArr(iarr);
```

```
double[] darr = { 1, 2, 4, 5, 7 };  
WriteTArr(darr);
```

```
void Program.WriteTArr<double>(double[] arr)
```

Определение конкретного типа происходит на этапе компиляции на основании типа переданных аргументов или явного указания типа

```
WriteTArr<int>(iarr);
```

# Виды обобщений

---

- Обобщённые методы

```
void WriteTArr<T>(T[] arr) { }
```

- Обобщённые типы (классы и структуры)

```
class MyList<T>
```

```
struct Pair<TKey, TValue>
```

- Обобщённые интерфейсы

```
interface IDrawable<T>
```

- Обобщённые делегаты

```
delegate void DoSomething<T>(T x, T y);
```

# Ограничения обобщений

Действия выполнимы только если компилятор о них знает:

```
T Max<T>(T x, T y)
{
    return x > y ? x : y;
}
```

(parameter) T x

Error:

Operator '>' cannot be applied to operands of type 'T' and 'T'

Правильная реализация:

```
T Max<T>(T x, T y) where T : IComparable<T>
{
    return x.CompareTo(y) > 0 ? x : y;
}
```

# Ограничение обобщения

Синтаксис:

where шаблон: список\_ограничений

| Ограничение             | Действие   |
|-------------------------|--|
| where T: struct         | Структура (value-type)                           |
| where T: class          | Класс (reference-type)                           |
| where T: new()          | Тип должен иметь конструктор по умолчанию        |
| where T: имя класса     | Соответствует указанному классу или его потомкам |
| where T: имя интерфейса | Реализует указанный интерфейс                    |
| where TChild: TBase     | TChild — класс, являющийся потомком класса TBase |

# Примеры

- Генератор массивов:

```
static T[] Generate<T>(int n) where T : new()
{
    T[] arr = new T[n];
    for (int i = 0; i < n; i++)
    {
        // Требуется конструктор по умолчанию
        arr[i] = new T();
    }
    return arr;
}
```

Использование:

```
int[] ints = Generate<int>(6);
MemoryStream[] streams = Generate<MemoryStream>(6);
```

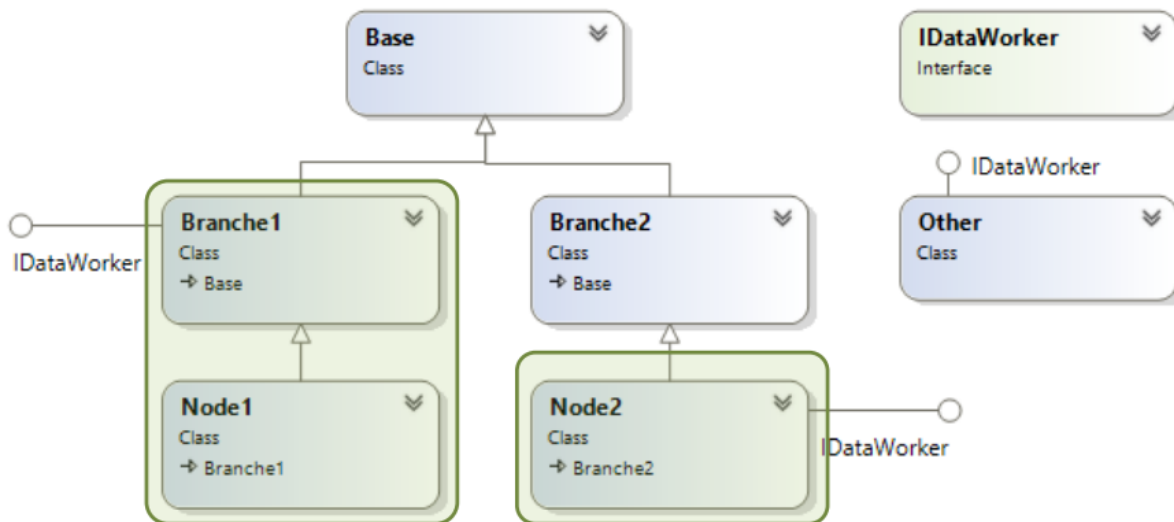
Почему оно будет работать?





# Примеры

## Задание конкретного класса



```
static T[] Generate<T>(int n)
    where T : Base, IDataWorker, new()
```

# Примеры

- Ограничение наследованием

```
static TBase[] ConvertToBase<TBase, TChild>(TChild[] array)
    where TChild : class, TBase
{
    TBase[] result = new TBase[array.Length];
    for (int i = 0; i < array.Length; i++)
    {
        // Требуется ссылочная совместимость (наследование)
        result[i] = array[i];
    }
    return result;
}
```

- Использование

```
Circle[] circles = { new Circle(2), new Circle(5) };
Figure[] figures = ConvertToBase<Figure, Circle>(circles);
```

# Получение значения по умолчанию

- Оператор default

```
static T[] Generate<T>(int n)
{
    T[] arr = new T[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = default(T);
    }
    return arr;
}
```



|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|

```
int[] ints = Generate<int>(6);
MemoryStream[] streams = Generate<MemoryStream>(6);
```



|      |      |      |      |      |      |
|------|------|------|------|------|------|
| null | null | null | null | null | null |
|------|------|------|------|------|------|

# Поиск полиморфизма там, где его нет

```
class Storage<T>
{
    public T Data { get; set; }
}
```

```
// Это ссылочная совместимость
Parent p = new Child();
```

```
// А это непонимание работы обобщённых типов
Storage<Parent> st = new Storage<Child>();
```



Storage<Child>.Storage()

Cannot implicitly convert type 'Demo.Storage<Demo.Child>' to 'Demo.Storage<Demo.Parent>'

[Show potential fixes \(Ctrl+.\)](#)

**Storage<Parent>** и **Storage<Child>** — два независимых типа данных

# Ковариация и контравариация

---

- Ковариация обобщённого класса позволяет возвращать вместо него классы-потомки, но запрещает их передавать.
- Контравариация обобщённого класса позволяет передавать вместо него классы-предки, но запрещает их возвращать.
- Ковариация и контравариация применимы только в обобщённых интерфейсах и делегатах.

# Пример ковариации

```
List<string> strings = new List<string>();  
strings.Add("test");  
IEnumerable<object> objects = strings;
```



Не работает в C# ниже 4.0

```
List<object> objects = new List<object>();  
objects.Add("test");  
IEnumerable<string> strings = objects;
```



Не может работать в принципе

# Пример ковариации

---

```
// out – ковариация
interface IGenerator<out T>
{
    T Create();
}
```

```
class ChildGenerator : IGenerator<Child>
{
    public Child Create()
}
```

```
IGenerator<Parent> gen = new ChildGenerator();
```

# Пример контрвариации

---

```
// in – контравариация
interface IProjector<in T>
{
    void Show(T obj);
}
```

```
class ParentProjector : IProjector<Parent>
{
    public void Show(Parent obj)...
```

```
IProjector<Child> prj = new ParentProjector();
```



СПАСИБО ЗА ВНИМАНИЕ