Обобщения

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:

```
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);
```

Обобщения

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 в 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, 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:

```
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.

Обобщения

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.

Обобщения

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); // 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 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.