



ПАРАМЕТРИЧНИЙ ПОЛІМОРФІЗМ. УЗАГАЛЬНЕНІ ТИПИ ДАНИХ

Питання 5.5.

Абстрагування типів даних



Абстрагування – це «процес визначення спільних паттернів, які мають систематичні варіації; абстракція представляє спільний паттерн та забезпечує засоби уточнення того, яку варіацію використовувати»

Richard Gabriel

- Абстрактні класи дозволяють спеціалізацію (уточнення) в похідних класах, проте поліморфізм не вимагає абстракції.
 - **Обговорення:** чи є поліморфізм способом досягнення абстракції?
 - Як методологія програмування, абстрагування передбачає приховування несуттєвих подробиць програмних кодів від користувача та надання йому тільки необхідних інструментів для взаємодії.
- **Параметричний поліморфізм** дозволяє створювати *універсальні* базові типи.
 - *Універсальними шаблонами (дженериками)* є класи, структури, інтерфейси й методи, що мають прототипи (заповнювачі – placeholders, параметри типів – type parameters) для одного або кількох типів, які вони зберігають або використовують.
 - Узагальнений клас не може використовуватись «as-is», оскільки він є лише шаблоном (blueprint) типу.

Універсальні шаблони (generics) у .NET

- Наприклад, клас універсальної колекції може використовувати параметр типу в якості заповнювача для типу об'єктів, які в ньому зберігаються.
 - Параметри типу відображаються як типи його полів і типи параметрів його методів.
 - Універсальний метод може використовувати параметр типу в якості типу вихідного значення або як тип одного зі своїх формальних параметрів.
 - При створенні екземпляра універсального класу необхідно вказати фактичні типи (actual types) для заміни параметрів типу.
 - При цьому створюється новий універсальний клас – *сконструйований універсальний клас (constructed generic type, constructed type)*, з вибраними типами, які замінюють всі параметри типу.
 - Результатом є типобезпечний клас, відповідний вашому вибору типів

```
public class Generic<T> { public T Field; }
```

```
public static void Main() {  
    Generic<string> g = new Generic<string>();  
    g.Field = "A string";  
    //...  
    Console.WriteLine("Generic.Field          = \"{0}\"", g.Field);  
    Console.WriteLine("Generic.Field.GetType() = {0}", g.Field.GetType().FullName);  
}
```

Терміни в контексті універсальних шаблонів у .NET

- **Визначення універсального типу (*generic type definition*)** – це оголошення класу, структури або інтерфейсу, яке працює як шаблон з прототипами для типів, які він може містити або використовувати.
 - Наприклад, клас `System.Collections.Generic.Dictionary <TKey, TValue>` може містити два типи: ключі і значення. Оскільки визначення універсального типу – це тільки шаблон, створювати екземпляри класу, структури або інтерфейсу, що є визначенням універсального типу, не можна.
- **Параметри універсального типу або параметри типу (*type parameters*)** є прототипами у визначенні універсального типу або методу.
 - Універсальний тип `System.Collections.Generic.Dictionary <TKey, TValue>` має два параметри типу `TKey` і `TValue`, які представляють типи його ключів і значень.
- **Сконструйований універсальний тип або сконструйований тип (*constructed type*)** є результатом вказівки типів для параметрів універсального типу у визначенні універсального типу.
- **Аргумент універсального типу (*generic type argument*)** є будь-яким типом, замінним на параметр універсального типу.

Дженерики в С#

Specifying the Type Parameter using angular brackets

`public static bool AreEqual<T>(T value1, T value2)`

Using that Type Parameter as the data type of the method parameter

Specifying the data type as integer

`bool IsEqual = ClsCalculator.AreEqual<int>(10, 20);`

Specifying the Data Type the AreaEqual method to work as integer

`bool IsEqual = ClsCalculator.AreEqual<int>(10, 20);`

T is replaced with int

`public static bool AreEqual<T>(T value1, T value2)`

int

int

```
namespace GenericsDemo
```

```
{
```

```
    public class ClsMain
```

```
    {
```

```
        private static void Main()
```

```
        {
```

```
            //bool IsEqual = ClsCalculator.AreEqual<int>(10, 20);
```

```
            //bool IsEqual = ClsCalculator.AreEqual<string>("ABC", "ABC");
```

```
            bool IsEqual = ClsCalculator.AreEqual<double>(10.5, 20.5);
```

```
            if (IsEqual)
```

```
            {
```

```
                Console.WriteLine("Both are Equal");
```

```
            }
```

```
            else
```

```
            {
```

```
                Console.WriteLine("Both are Not Equal");
```

```
            }
```

```
            Console.ReadKey();
```

```
        }
```

```
    }
```

```
    public class ClsCalculator
```

```
    {
```

```
        public static bool AreEqual<T>(T value1, T value2)
```

```
        {
```

```
            return value1.Equals(value2);
```

```
        }
```

```
    }
```

```
}
```

Терміни в контексті універсальних шаблонів у .NET

- Загальний термін *універсальний тип* включає визначення як сконструйованих типів, так і універсальних типів.
- **Коваріація і контраваріантність** параметрів універсального типу дозволяють використовувати сконструйовані універсальні типи, аргументи типів яких знаходяться на більш високому (в разі коваріації) або низькому (в разі контраваріантності) рівні ієрархії успадкування, ніж у цільового сконструйованого типу.
 - Разом коваріантність і контраваріантність називають **варіацією**.
- **Обмеження** – це границі, накладені на параметри універсального типу.
 - Наприклад, можна обмежити параметр типу типами, що реалізують універсальний інтерфейс `System.Collections.Generic.IComparer<T>`, щоб забезпечити упорядкування екземплярів типу.
 - Можна також обмежити параметри типу типами, які мають деякий базовий клас, що містить конструктор без параметрів, або типами, які є посилальними типами чи значимими типами.
 - Користувачі універсального типу не можуть підставити аргументи типу, які не задовольняють обмеженням.

Терміни в контексті універсальних шаблонів у .NET

```
T Generic<T>(T arg)
{
    T temp = arg;
    //...
    return temp;
}
```

```
class A {
    T G<T>(T arg) {
        T temp = arg;
        //... return temp;
    }
}
```

```
class Generic<T> {
    T M(T arg) {
        T temp = arg;
        //... return temp;
    }
}
```

- **Визначення універсального методу (*generic method definition*)** – це метод з двома списками параметрів: списком параметрів універсальних типів і списком формальних параметрів.
 - Параметри типу можуть з'являтися як вихідний тип або в якості типів формальних параметрів.
- Універсальні методи можуть бути присутніми в універсальних і неуніверсальних типах.
 - Метод не є універсальним тільки тому, що він належить універсальному типу або навіть якщо він має формальні параметри, типи яких є універсальними параметрами для типу, що їх включає.
 - Метод є універсальним тільки в тому випадку, якщо він має свій власний список параметрів типу.
 - У коді тільки метод G є універсальним.

Переваги універсальних шаблонів

- **Типобезпечність.** Універсальні шаблони дозволяють передати компілятору обов'язки забезпечення безпеки типів. Немає необхідності написання коду для перевірки правильності типу даних, так як перевірка відбувається під час компіляції.
 - Зменшується потреба в зведенні типів та ймовірність помилок під час виконання.
- **Обсяг коду зменшений і підтримує багаторазову реалізацію.**
 - Немає необхідності успадкування базового типу і переважаючих членів. Наприклад, `LinkedList <T>` готовий до негайного використання.
 - Наприклад, можна створити пов'язаний список рядків з наступним оголошенням змінної:
`LinkedList<string> llist = new LinkedList<string>();`
- **Підвищена продуктивність.** Універсальні типи колекцій мають більш високу продуктивність при зберіганні й управлінні значимими типами, оскільки відсутня необхідність їх упаковки.
- **Універсальні делегати підтримують типобезпечні зворотні виклики** без необхідності створення декількох класів делегатів.
 - Наприклад, універсальний делегат `Predicate <T>` дозволяє створити метод, який реалізує власні умови пошуку для певного типу, і використовувати ваш метод з методами типу `Array`, такими як `Find`, `FindLast` і `FindAll`.
- **Універсальні шаблони спрощують динамічно створюваний код.**
 - Універсальні делегати можна також використовувати в динамічно створюваному коді без необхідності створення типу делегата.
 - Це збільшує кількість ситуацій, в яких можна використовувати полегшені динамічні методи замість створення цілих збірок.

Недоліки (limitations) дженериків

- Узагальнені типи можна отримати з найбільш базових класів, зокрема `MarshalByRefObject`.
 - Проте .NET Framework не підтримує context-bound узагальнені типи.
 - Узагальнений тип може породжуватись від `ContextBoundObject`, проте спроба створення екземпляру цього типу викликає `TypeLoadException`.
- Перелічення не можуть мати узагальнені параметри типів `generic type parameters`.
 - Перелічення може бути узагальненим тільки опосередковано (наприклад, через свою вкладеність в узагальнений тип, визначений мовами Visual Basic, C# або C++).
- Легковагові динамічні методи не можуть бути узагальненими.
- У Visual Basic, C# та C++ вкладений тип, охоплений узагальненим типом, не може інстанціюватись, поки не визначено всі параметри типу для охоплюючих типів.
 - Іншими словами, вкладений тип, що визначено з використанням цих мов, включає параметри типу усіх охоплюючих типів.
 - Це дозволяє параметрам типів охоплюючого типу застосовуватись у визначеннях членів вкладеного типу.

Обмеження узагальнених параметрів у мові C#

- **Обмеження (*constraints*)** – конкретні правила, які інформують компілятор щодо можливостей, які повинен мати аргумент типу.
 - Без них аргумент типу може бути будь-яким (System.Object).
 - Коли клієнтський код інстанціює клас з недозволеним типом, отримуємо помилку часу компіляції.
- Для визначення обмеження використовується ключове слово **where**. Існує 8 типів обмежень:
 - **where T : struct**: аргумент повинен бути значимого типу, за винятком Nullable<T>.
 - **where T : class**: аргумент повинен бути посилального типу та застосовується до класів, інтерфейсів, делегатів та масивів.
 - **where T : notnull**: аргумент повинен бути ненулабельним.
 - **where T : unmanaged**: аргумент повинен бути unmanaged-типу, тобто вказівником чи з unsafe-коду.
 - **where T : base-class**: аргумент повинен породжуватись від базового класу типу.
 - **where T : new()**: аргумент повинен мати безаргументний публічний конструктор.
 - **where T : interface**: аргумент повинен бути чи реалізувати заданий інтерфейс.
 - **where T : U**: аргумент повинен бути чи породжуватись від аргументу, який постачається для U.

Обмеження узагальнених параметрів у мові C#

- Накладання обмежень на параметри типів дозволяє підвищити кількість дозволених операцій та викликів методів до такої, що підтримується обмежуючим типом (constraining type) та типами з його ієрархії наслідування.

```
namespace cnstraints {
    class GenericClass<T> where T : class {
        private readonly T _field;
        public GenericClass(T value) { this._field = value; }
        public T genericMethod(T parameter) {
            Console.WriteLine($"The type of parameter we got is: {typeof(T)} and value is: {parameter}");
            Console.WriteLine($"The return type of parameter is: {typeof(T)} and value is: {this._field}");
            return this._field;
        }
    }
}

class Program {
    static void Main(string[] args) {
        GenericClass<string> myGeneric = new GenericClass<string>("Hello World");
        myGeneric.genericMethod("string");
        Console.ReadKey();
    }
}
```

До яких програмних сутностей застосовуються дженерики?

- Дженерики можуть застосовуватись до:
 - Інтерфейсу
 - Абстрактного класу
 - Класу
 - Методу
 - Статичного методу
 - Властивості
 - Події
 - Делегата
 - Оператора.
- Узагальнений інтерфейс, який має коваріантні або контраваріантні параметри узагальнених типів, називають **варіативним (variant)**.
 - Варіативні узагальнені інтерфейси оголошуються за допомогою ключових слів `in` та `out` для узагальнених типів параметрів.
 - `ref`, `in` та `out` параметри в C# не можуть бути варіативними. Значимі типи також не підтримують варіативність.

Узагальнені делегати .NET

- Делегат може визначати власні параметри типу.

```
public delegate void Del<T>(T item);  
public static void Notify(int i) { }
```

```
Del<int> m1 = new Del<int>(Notify);
```

- У C# доступна функція групового перетворення (conversion) методів, яка застосовується як до конкретних, так і до узагальнених типів делегатів. Код спроститься:

```
Del<int> m2 = Notify;
```

- Делегати, визначені в універсальному класі, можуть використовувати параметри класу універсального типу таким же чином, як це роблять методи класу.

```
class Stack<T> {  
    T[] items;  
    int index;  
  
    public delegate void StackDelegate(T[] items);  
}
```

```
private static void DoWork(float[] items) { }  
  
public static void TestStack() {  
    Stack<float> s = new Stack<float>();  
    Stack<float>.StackDelegate d = DoWork;  
}
```

Узагальнені делегати .NET

- Універсальні делегати особливо корисні при визначенні подій, заснованих на типових шаблонах розробки, оскільки аргумент відправника може бути строго типізований і більше не вимагає зведення до `Object` та з нього.

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T> {
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

    protected virtual void OnStackChanged(StackEventArgs a) { stackEvent(this, a); }
}

class SampleClass {
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test() {
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}
```

Варіативність в узагальнених інтерфейсах

- У .NET Framework 4 представлена підтримка варіативності для кількох існуючих узагальнених інтерфейсів:
 - IEnumerable<T> (T – коваріант)
 - IEnumerator<T> (T – коваріант)
 - IQueryable<T> (T – коваріант)
 - IGrouping<TKey,TElement> (TKey та TElement – коваріанти)
 - IComparer<T> (T – коваріант)
 - IEqualityComparer<T> (T – коваріант)
 - IComparable<T> (T – коваріант)
- Починаючи з .NET Framework 4.5, варіативними є інтерфейси:
 - IReadOnlyList<T> (T is covariant)
 - IReadOnlyCollection<T> (T is covariant)
- **Коваріація** дозволяє методу мати *нижчий в ієрархії наслідування вихідний тип*, ніж визначений узагальнений параметр типу в інтерфейсі.
 - **Контраваріантність** дозволяє методам інтерфейсу мати *типи аргументів, які знаходяться вище в ієрархії наслідування*, ніж це задано узагальненими параметрами.

За допомогою ключового слова `out` оголошується коваріантний параметр узагальненого типу

- Вимоги до коваріантного типу:

- Тип використовується тільки в якості типу вихідного значення методу в інтерфейсі та не застосовується в якості типу аргументів методу:

```
interface ICovariant<out R> {  
    R GetSomething();  
    // void SetSomething(R sampleArg); // помилка компіляції  
}
```

- Виняток: якщо параметром є контраваріантний узагальнений делегат, цей тип можна використовувати як параметр узагальненого типу для цього делегата (у прикладі – тип `R`):

```
interface ICovariant<out R> {  
    void DoSomething(Action<R> callback);  
}
```

- Тип не використовується як узагальнене обмеження для методів інтерфейсу:

```
interface ICovariant<out R> {  
    // Сигнатура генерує помилку компіляції, оскільки можна використовувати  
    // тільки контраваріантні або інваріантні типи в узагальнених обмеженнях  
    // void DoSomething<T>() where T : R;  
}
```


Для оголошення контраваріантного параметра узагальненого типу можна використовувати ключове слово `in`

- Контраваріантний тип можна застосувати тільки в якості типу аргументів методу, проте не в якості типу вихідного значення для методів інтерфейсу.
 - Контраваріантний тип можна також використати для узагальнених обмежень.

```
interface IContravariant<in A> {  
    void SetSomething(A sampleArg);  
    void DoSomething<T>() where T : A;  
    // A GetSomething(); // помилка компіляції  
}
```

- Крім того, можна реалізувати підтримку коваріації та контраваріації в одному інтерфейсі, проте для різних параметрів типу:

```
interface IVariant<out R, in A> {  
    R GetSomething();  
    void SetSomething(A sampleArg);  
    R GetSetSomethings(A sampleArg);  
}
```

Реалізація варіативних узагальнених інтерфейсів

- Використовується той же синтаксис, що і для інваріантних інтерфейсів.

```
interface ICovariant<out R> {  
    R GetSomething();  
}  
  
class SampleImplementation<R> : ICovariant<R> {  
    public R GetSomething() {  
        // Деякий код.  
        return default(R);  
    }  
}
```

- Класи, які реалізують варіативні інтерфейси, є інваріантними.

// Інтерфейс коваріатний.

```
ICovariant<Button> ibutton = new SampleImplementation<Button>();
```

```
ICovariant<Object> iobj = ibutton;
```

// Клас інваріатний.

```
SampleImplementation<Button> button = new SampleImplementation<Button>();
```

```
// SampleImplementation<Object> obj = button; // помилка компіляції через інваріантність класу
```

Розширення варіативних універсальних інтерфейсів

- При розширенні варіативних узагальнених інтерфейсів необхідно використовувати ключові слова `in` та `out` для явного вказування того, чи підтримує варіативність похідний інтерфейс.

- Компілятор не передбачає варіативність інтерфейса, який розширяється.

```
interface ICovariant<out T> { }  
interface IInvariant<T> : ICovariant<T> { }  
interface IExtCovariant<out T> : ICovariant<T> { }
```

- В інтерфейсі `IInvariant<T>` параметр узагальненого типу `T` є інваріантним, а в `IExtCovariant<out T>` - коваріантним, хоч обидва інтерфейси розширяють один і той же інтерфейс.
 - Те ж правило застосовується до контраваріантних параметрів узагальненого типу.
 - Можна створити інтерфейс, який розширяє як інтерфейс, у якому параметр узагальненого типу `T` є коваріантним, так і інтерфейс, де він є контраваріантним, якщо в інтерфейсі, що розширяється, параметр узагальненого типу `T` є інваріантним.

```
interface ICovariant<out T> { }  
// The following statement generates a compiler error.  
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Недопущення неоднозначності

```
class Animal { }
class Cat : Animal { }
class Dog : Animal { } // This class introduces ambiguity
                        // because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog> {
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator() {
        Console.WriteLine("Cat"); // Some code. return null;
    }
    IEnumerator IEnumerable.GetEnumerator() {
        // Some code. return null;
    }
    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator() {
        Console.WriteLine("Dog");
        // Some code. return null;
    }
}
class Program {
    public static void Test() {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

- При реалізації варіативних узагальнених інтерфейсів варіативність може призводити до неоднозначності, якої слід уникати.
 - Наприклад, якщо явно реалізується один варіантний універсальний інтерфейс із різними параметрами узагальненого типу в одному класі.
 - Компілятор не повідомляє про помилку в такому випадку, проте і не вказує, яка реалізація інтерфейсу буде обрана під час виконання.
 - Така неоднозначність може призвести до виникнення неявних помилок у коді.
 - У прикладі не вказано, яким чином метод `pets.GetEnumerator` здійснює вибір між `Cat` і `Dog`, що може викликати проблеми.

Варіативність в узагальнених делегатах

- У платформі .NET Framework 4+ можна включити неявне перетворення між делегатами, яке дозволить узагальненим методам-делегатам, що мають різні типи, вказані параметрами узагальненого типу, бути призначеними один одному, якщо типи успадковуються один від одного так, як того вимагає варіативність.
 - Щоб задіяти неявне перетворення, потрібно явно оголосити узагальнені параметри в делегаті як коваріантні або контраваріантні за допомогою ключового слова `in` або `out`.

// Тип T оголошено коваріантним за допомогою ключового слова `out`.

```
public delegate T SampleGenericDelegate <out T>();  
public static void Test() {  
    SampleGenericDelegate <String> dString = () => " ";    // Можна присвоїти делегати один одному,  
                                                           // оскільки тип T оголошено коваріантним.  
    SampleGenericDelegate <Object> dObject = dString;  
}
```

- Якщо підтримка варіативності використовується тільки для зіставлення сигнатур методів з типами делегатів, а ключові слова `in` та `out` не використовуються, можна створити екземпляри делегатів з однаковими лямбда-виразами або методами, проте неможливо призначити один делегат іншому.

Варіативність в узагальнених делегатах

- У наступному прикладі `SampleGenericDelegate<String>` неможливо явно перетворити в `SampleGenericDelegate<Object>`, хоч `String` успадковується від `Object`.
 - Цю проблему можна усунути, відмітивши узагальнений параметр `T` ключовим словом `out`.

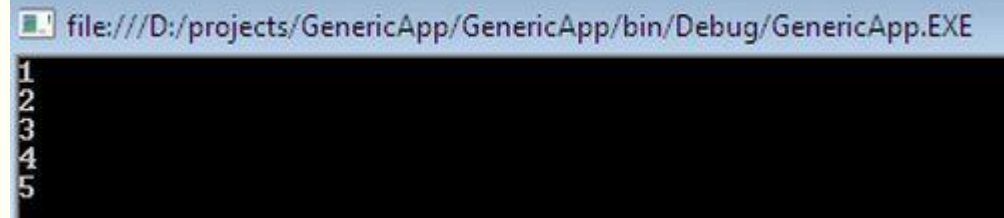
```
public delegate T SampleGenericDelegate<T>();
```

```
public static void Test() {  
    SampleGenericDelegate<String> dString = () => " "; // можна присвоїти делегат dObject  
                                                         // тому ж лямбда-виразу, що і делегат dString,  
                                                         // оскільки є підтримка варіативності для співставних  
                                                         // сигнатур методів з типами delegate.  
  
    SampleGenericDelegate<Object> dObject = () => " ";  
    // Далі помилка компіляції, оскільки узагальнений тип T не позначений коваріантним.  
    // SampleGenericDelegate <Object> dObject = dString;  
}
```

Узагальнені методи-делегати з варіативними параметрами типу

- У платформі .NET Framework 4 з'явилась підтримка варіативності для параметров узагальненого типу в декількох існуючих методах-делегатах.
 - Делегати Action з простору імен System, наприклад Action<T> та Action<T1,T2>
 - Делегати Func з простору імен System, наприклад Func<TResult> та Func<T,TResult>
 - Делегат Predicate<T>
 - Делегат Comparison<T>
 - Делегат Converter<TInput,TOutput>
- Додаткові приклади

Приклад узагальненого класу



```
public class TestClass<T>
{
    // визначення масиву на 5 значень узагальненого типу
    T[] obj = new T[5];
    int count = 0;

    // механізм додавання елементів узагальненого типу
    public void Add(T item)
    {
        //перевірка довжини
        if (count + 1 < 6)
        {
            obj[count] = item;

            count++;
        }
    }
    //індексатор для оператора foreach
    public T this[int index]
    {
        get { return obj[index]; }
        set { obj[index] = value; }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        //інстанціювання дженерика типом Integer
        TestClass<int> intObj = new TestClass<int>();

        //додавання значень у колекцію
        intObj.Add(1);
        intObj.Add(2);
        intObj.Add(3);    //упаковка відсутня
        intObj.Add(4);
        intObj.Add(5);

        //виведення значень
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(intObj[i]); //відсутня розпаковка
        }
        Console.ReadKey();
    }
}
```




ДЯКУЮ ЗА УВАГУ!

Наступна тема: Принципи побудови якісного об'єктно-орієнтованого коду