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

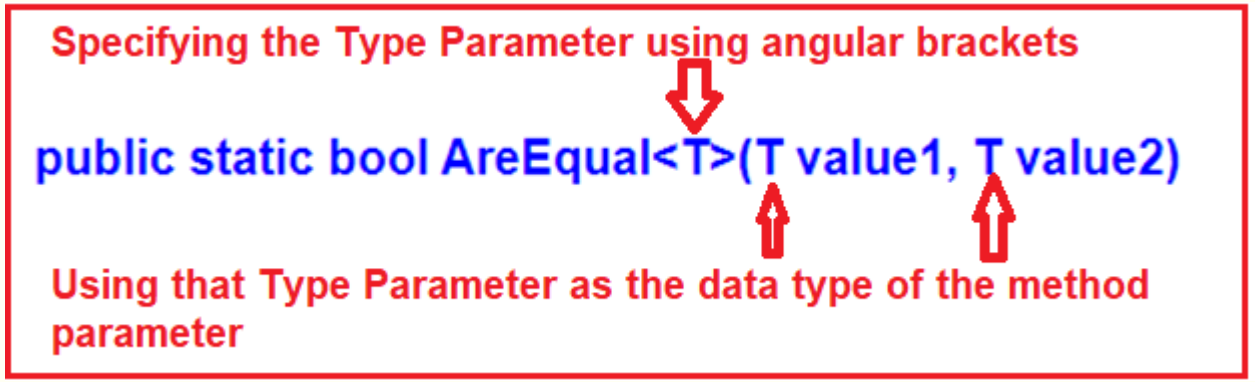
- Наприклад, клас універсальної колекції може використовувати параметр типу в якості заповнювача для типу об'єктів, які в ньому зберігаються.
  - Параметри типу відображаються як типи його полів і типи параметрів його методів.
  - Універсальний метод може використовувати параметр типу в якості типу вихідного 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 заміняють всі параметри типу.
  - Результатом є типобезпечний клас, відповідний вашому вибору типів

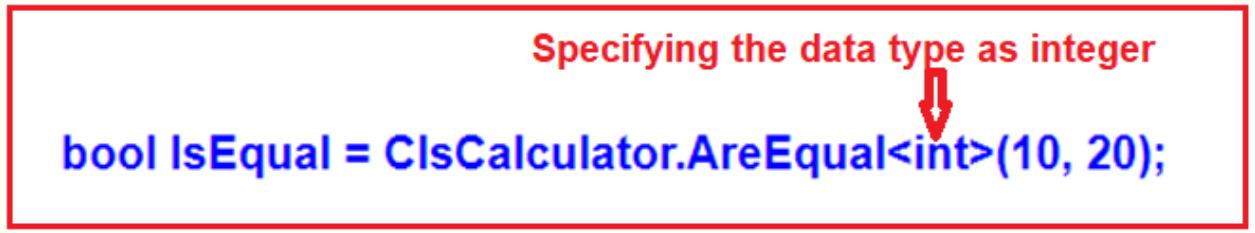
```
public class Generic<T> { public T Field; }

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);
}
```

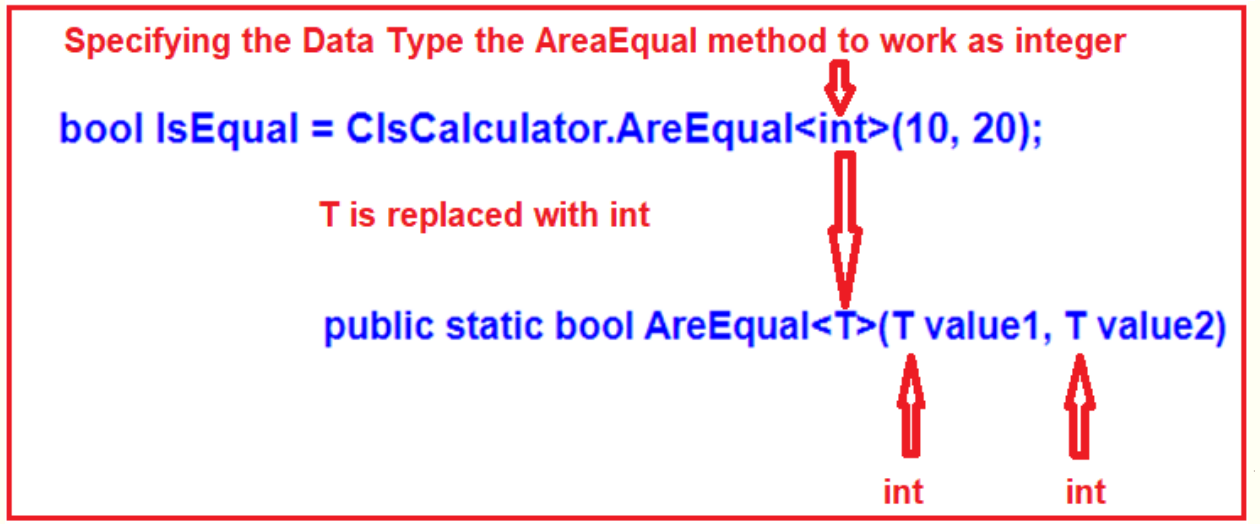
Терміни в контексті універсальних шаблонів у .NET

- ***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 C#








```
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 (and constraints can be used to require that generic type parameters derive from base classes like MarshalByRefObject).
  - However, the .NET Framework does not support context-bound generic types. A generic type can be derived from ContextBoundObject, but trying to create an instance of that type causes a TypeLoadException.
- Перелічення не можуть мати узагальнені параметри типів.generic type parameters.
  - An enumeration can be generic only incidentally (for example, because it is nested in a generic type that is defined using Visual Basic, C#, or C++). For more information, see "Enumerations" in Common Type System.
- 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 за допомогою a nested type that is defined using these languages includes the type parameters of all its enclosing types. This allows the type parameters of enclosing types to be used in the member definitions of a nested type. For more information, see "Nested Types" in MakeGenericType.

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;
```

  - Делегаты, определенные в универсальном классе, могут использовать параметры класса универсального типа таким же образом, как это делают методы класса.

```
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

- Универсальные делегаты особенно полезны при определении событий, основанных на типовых шаблонах разработки, поскольку аргумент отправителя может быть строго типизирован и больше не требует приведения к Object и из него

```
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 можа 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 объявления контравариантного параметра универсального типа можно использовать ключевое слово in

- Контравариантный тип можно использовать только в качестве типа аргументов метода, но не в качестве типа значения, возвращаемого методами интерфейса.
  - Контравариантный тип можно также использовать для универсальных ограничений.

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

  - Кроме того, можно реализовать поддержку ковариации и контравариации в одном интерфейсе, но для разных параметров типа:

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

Реалізація варіативних узагальнених інтерфейсів

- используется тот же синтаксис, что и для инвариантных интерфейсов.

```
interface ICovariant<out R> {
    R GetSomething();
}

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

  - Классы, которые реализуют вариантные интерфейсы, являются инвариантными.

```
// 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 расширении вариантных универсальных интерфейсов необходимо использовать ключевые слова in и out для явного указания того, поддерживает ли вариативность производный интерфейс.
  - Компилятор не подразумевает вариативность интерфейса, который расширяется.

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

  - В интерфейсе IInvariant<T> параметр универсального типа T является инвариантным, тогда как в IExtCovariant<out T> параметр типа является ковариантным, хотя оба интерфейса расширяют один и тот же интерфейс.
  - То же правило применяется к контравариантным параметрам универсального типа.
  - Можно создать интерфейс, который расширяет и интерфейс, в котором параметр универсального типа T является ковариантным, и интерфейс, где он является контравариантным, если в расширяемом интерфейсе параметр универсального типа T является инвариантным.

```
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 реализации вариантных универсальных интерфейсов вариативность может приводить к неоднозначности. Такой неоднозначности следует избегать.
- Например, если вы явно реализуете один вариантный универсальный интерфейс с разными параметрами универсального типа в одном классе, это может создавать неоднозначность. Компилятор не сообщает об ошибке в данном случае, но и не указывает, какая реализация интерфейса будет выбрана во время выполнения. Такая неоднозначность может привести к возникновению неявных ошибок в коде.
- В этом примере не указано, каким образом метод pets.GetEnumerator делает выбор между Cat и Dog. Это может вызвать проблемы в вашем коде.

Варіативність в узагальнених делегатах

- В платформе .NET Framework 4 и более поздних версиях можно включить неявное преобразование между делегатами, которое позволит универсальным методам-делегатам, имеющим разные типы, указанные параметрами универсального типа, быть назначенными друг другу, если типы наследуются друг от друга так, как того требует вариативность.
  - Чтобы включить неявное преобразование, необходимо явно объявить универсальные параметры в делегате как ковариантные или контравариантные с помощью ключевого слова in или out.

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();
public static void Test() {
    SampleGenericDelegate <String> dString = () => " ";   // You can assign delegates to each other,
                                                          // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

  - Если поддержка вариативности используется только для сопоставления сигнатур методов с типами делегатов, а ключевые слова in и out не используются, можно создать экземпляры делегатов с одинаковыми лямбда-выражениями или методами, но нельзя назначить один делегат другому.

Варіативність в узагальнених делегатах

- В следующем примере кода SampleGenericDelegate<String> нельзя явно преобразовать в SampleGenericDelegate<Object>, хотя String наследует Object.
  - Эту проблему можно устранить, пометив универсальный параметр T ключевым словом out.

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

public static void Test() {
    SampleGenericDelegate<String> dString = () => " "; // You can assign the dObject delegate
                                                       // to the same lambda expression as dString delegate
                                                       // because of the variance support for
                                                       // matching method signatures with delegate types.

    SampleGenericDelegate<Object> dObject = () => " ";
    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;
}
```

Узагальнені методи-делегати з варіативними параметрами типу

- В платформе .NET Framework 4 появилась поддержка вариативности для параметров универсального типа в нескольких существующих методах-делегатах.
  - Делегаты Action из пространства имен System, например Action<T> и Action<T1,T2>
  - Делегаты Func из пространства имен System, например Func<TResult> и Func<T,TResult>
  - Делегат Predicate<T>
  - Делегат Comparison<T>
  - Делегат Converter<TInput,TOutput>
- Додаткові приклади

Приклад узагальненого класу


```
public class TestClass<T>
{
    // define an Array of Generic type with length 5
    T[] obj = new T[5];
    int count = 0;

    // adding items mechanism into generic type
    public void  Add(T item)
    {
        //checking length
        if (count + 1 < 6)
        {
            obj[count] = item;

        }
        count++;
    }
    //indexer for foreach statement iteration
    public T this[int index]
    {
        get { return obj[index]; }
        set { obj[index] = value; }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        //instantiate generic with Integer
        TestClass<int> intObj = new TestClass<int>();

        //adding integer values into collection
        intObj.Add(1);
        intObj.Add(2);
        intObj.Add(3);     //No boxing
        intObj.Add(4);
        intObj.Add(5);

        //displaying values
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(intObj[i]);   //No unboxing
        }
        Console.ReadKey();
    }
}
```

ДЯКУЮ ЗА УВАГУ!

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