SVEUČILIŠTE U SPLITU

FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

Algoritmi i strukture podataka

Vježba 6.

.NET Generics

Ivan Banovac, 220

Datum izvođenja vježbe:
22.12.2025

Zadatak 1. – Ordinary versus Generic Types

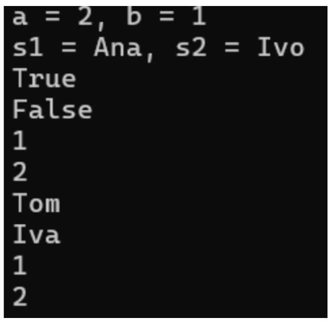
Ovaj dio vježbe demonstrira osnovne generičke metode *Swap<T>* za zamjenu vrijednosti i *Compare<T>* za provjeru jednakosti . Naglasak je na razlici između kolekcija poput *List<object>*, koje zahtijevaju eksplicitno pretvaranje tipova, i generičkih kolekcija poput *List<int>*, koje sprječavaju pogreške već pri samom pisanju koda.

Program.cs:

```
static void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
static bool Compare<T>(T a, T b)
{
    return EqualityComparer<T>.Default.Equals(a, b);
}
int a = 1, b = 2;
string s1 = "Ivo", s2 = "Ana";
Swap<int>(ref a, ref b);
Console.WriteLine($"a = {a}, b = {b}");
Swap(ref s1, ref s2);
Console.WriteLine($"s1 = {s1}, s2 = {s2}");
bool cmp = Compare<int>(a, 2);
Console.WriteLine(cmp);
cmp = Compare(s1, s2);
Console.WriteLine(cmp);
List<object> ordinary = new()
{
1,
2,
"Tom"
};
int i = (int)ordinary[0];
ordinary.Add("Iva");
foreach (var x in ordinary)
    Console.WriteLine(x);
List<int> generic = new()
{
1,
2
// generic.Add("Ana");
};
foreach (var x in generic)
    Console.WriteLine(x);
```

U Main programu metode se testiraju s različitim tipovima podataka. Također se prikazuje razlika između obične liste tipa *List<object>,* koja dopušta pohranu različitih tipova i zahtijeva castanje, te generičke liste *List<int>,* koja omogućuje pohranu elemenata određenog tipa.

Rezultati:



```
a = 2, b = 1
s1 = Ana, s2 = Ivo
True
False
1
2
Tom
Iva
1
2
```

Zadatak 2. – Defining Generic types

U ovom zadatku definiramo vlastite generičke klase kroz implementaciju jednostruko povezane liste . Koriste se klase *Node<T>* za pohranu podataka i *GenericList<T>,* koja implementira sučelje *IEnumerable<T>*. Ovo omogućuje da se kroz vlastitu listu prolazi pomoću standardne *foreach* petlje.

Node.cs:

```
internal class Node<T>
{
    public T Data { get; set; } = default!;
    public Node<T>? Next { get; set; }
    public Node(T data)
    {
        Data = data;
    }
}
```

GenericList.cs

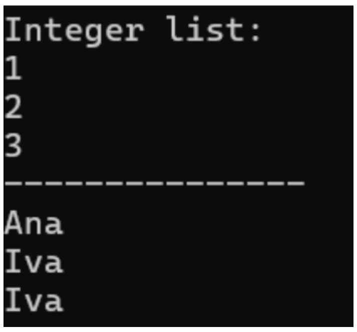
```
using System.Collections;

internal class GenericList<T> : IEnumerable<T>
{
    private Node<T> _head;
    private Node<T> _tail;
    public void AddFront(T data)
    {
        var node = new Node<T>(data);
        if (_head is null)
        {
            _head = _tail = node;
        }
        else
        {
            node.Next = _head;
            _head = node;
        }
    }
 public IEnumerator<T> GetEnumerator()
    {
        var current = _head;
        while (current is not null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Program.cs

```
var integers = new GenericList<int>();
integers.AddFront(3);
integers.AddFront(2);
integers.AddFront(1);
//integers.AddFront("Bob");
Console.WriteLine("Integer list:");
foreach (var i in integers)
Console.WriteLine(i);
Console.WriteLine("---------------");
var strings = new GenericList<string>();
strings.AddFront("Iva");
strings.AddFront("Iva");
strings.AddFront("Ana");
//strings.AddFront(5);
//integers = strings;
foreach (var s in strings)
Console.WriteLine(s);
```

Rezultat:



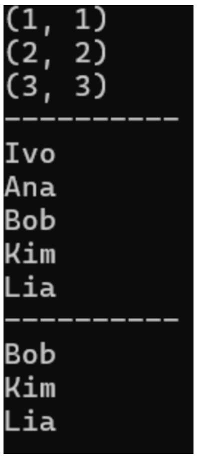
```
Integer list:
1
2
3
---------------
Ana
Iva
Iva
```

Zadatak 3. – Constraining Type Parameters
Ovaj zadatak pokazuje kako pomoću ključne riječi *where* možemo točno odrediti kakve tipove podataka neka generička klasa smije primiti.
Generički stog (*Stack<T>*) koji prihvaća samo vrijednosne tipove (where T: struct), poput koordinata u strukturi Point.
Generički red (*Queue<T>*) koji je ograničen na referentne tipove (*where T: class*), što je demonstrirano na primjeru nizova znakova (*strings*)

Point.cs:
```
namespace Constraints
{
    public struct Point
    {
        public int X { get; }
        public int Y { get; }
        public Point(int x, int y) => (X, Y) = (x, y);
        public override string ToString() => $"({X}, {Y})";
    }
}
```
Stack.cs:
```
using System.Collections;
namespace Constraints
{
    internal class Stack<T> : IEnumerable<T> where T : struct
    {
        private int _top = -1;
        private readonly T[] _array;
        public Stack(int capacity)
        {
            if (capacity <= 0)
                throw new ArgumentOutOfRangeException(nameof(capacity),
               "Capacity must be positive.");
            _array = new T[capacity];
        }
        public void Push(T data)
        {
            if (_top == _array.Length - 1)
                throw new InvalidOperationException("Stack is full.");
            _array[++_top] = data;
        }
        public T Pop()
        {
            if (_top == -1)
                throw new InvalidOperationException("Stack is empty.");
            return _array[_top--];
        }
        public IEnumerator<T> GetEnumerator()
        {
            for (int i = _top; i >= 0; i--)
                yield return _array[i];
        }
        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
    }
}
```

Queue.cs:
```
using System.Collections;
namespace Constraints
{
    internal class Queue<T> : IEnumerable<T> where T : class
    {
        private readonly T[] _array;
        private int _head;
        private int _tail;
        private int _count;
        public Queue(int capacity)
        {
            if (capacity <= 0)
                throw new ArgumentOutOfRangeException(nameof(capacity),
               "Capacity must be positive.");
            _array = new T[capacity];
        }
        public void Enqueue(T data)
        {
            if (_count == _array.Length)
                throw new InvalidOperationException("Queue is full.");
            _array[_tail] = data;
            _tail = (_tail + 1) % _array.Length;
            _count++;
        }
        public T Dequeue()
        {
            if (_count == 0)
                throw new InvalidOperationException("Queue is empty.");
            var data = _array[_head];
            _array[_head] = null!;
            _head = (_head + 1) % _array.Length;
            _count--;
            return data;
        }
        public IEnumerator<T> GetEnumerator()
        {
            for (int i = 0; i < _count; i++)
            {
                yield return _array[(_head + i) % _array.Length];
            }
        }
        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
    }
}
```

Program.cs:
```
using Constraints;
var integers = new Constraints.Stack<int>(10);
//var strings = new Constraints.Stack<string>(10);
var points = new Constraints.Stack<Point>(5);
points.Push(new Point(3, 3));
points.Push(new Point(2, 2));
points.Push(new Point(1, 1));
foreach (var p in points)
    Console.WriteLine(p);
points.Pop();
points.Pop();
points.Pop();
Console.WriteLine("----------");
var strings = new Constraints.Queue<string>(5);
//var integers2 = new Constraints.Queue<int>(5);
strings.Enqueue("Ivo");
strings.Enqueue("Ana");
strings.Enqueue("Bob");
strings.Enqueue("Kim");
strings.Enqueue("Lia");
foreach (var s in strings)
    Console.WriteLine(s);
Console.WriteLine("----------");
strings.Dequeue();
strings.Dequeue();
foreach (var s in strings)
Console.WriteLine(s);
```

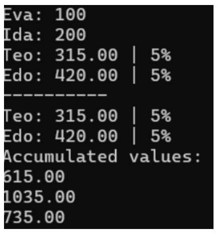
Rezultat:

```
(1, 1)
(2, 2)
(3, 3)
----------
Ivo
Ana
Bob
Kim
Lia
----------
Bob
Kim
Lia
```

Zadatak 4. – Generic Methods
Kroz sučelje *IAccount* i klase *Account* te *SavingsAccount*, prikazano je kako generičke metode poput *Display* i *Accumulate* mogu obrađivati različite tipove računa dok god oni dijele istu baznu klasu ili sučelje.

Account.cs:
```
public class Account : IAccount
{
    public string Name { get; set; } = string.Empty;
    public virtual decimal Balance { get; set; }
    public Account() { }
    public Account(string name, decimal balance)
    {
        Name = name;
        Balance = balance;
    }
    public override string ToString() => $"{Name}: {Balance}";
}
```
Iaccount.cs:
```
public interface IAccount
{
    string Name { get; set; }
    decimal Balance { get; set; }
}
```
SavingsAccount.cs:
```
public class SavingsAccount : Account
{
    public decimal Interest { get; set; }
    public override decimal Balance => base.Balance * (1 + Interest
   / 100m);
    public SavingsAccount(string name, decimal balance, decimal
   interest)
    : base(name, balance)
    {
        Interest = interest;
    }
    public override string ToString() => base.ToString() + $" | {Interest}% ";
 }
```

Program.cs:
```
static void Display<T>(List<T> list) where T : IAccount
{
    //list.Add(new Account("Gea", 500m));
    foreach (var account in list)
        Console.WriteLine(account);
}
static decimal Accumulate<T>(IEnumerable<T> collection) where T : Account
{
    return collection.Sum(account => account.Balance);
}
var a = new Account("Eva", 100m);
var b = new Account("Ida", 200m);
var c = new SavingsAccount("Teo", 300m, 5m);
var d = new SavingsAccount("Edo", 400m, 5m);
Account[] array = { a, b, c };
var accounts = new List<Account> { a, b, c, d };
var savings = new List<SavingsAccount> { c, d };
//savings = accounts;
//accounts = savings;
//Display(array);
Display(accounts);
Console.WriteLine("----------");
Display(savings);
Console.WriteLine("Accumulated values:");
decimal total = Accumulate(array);
Console.WriteLine(total);
total = Accumulate(accounts);
Console.WriteLine(total);
total = Accumulate(savings);
Console.WriteLine(total);
```

Rezultati:


```
Eva: 100
Ida: 200
Teo: 315.00 | 5%
Edo: 420.00 | 5%
----------
Teo: 315.00 | 5%
Edo: 420.00 | 5%
Accumulated values:
615.00
1035.00
735.00
```

Zadatak 5. – Generic Delegates

U ovom se dijelu koristi napredna metoda *Accumulate* koja kao parametre prima delegate *Func* i *Predicate* . To omogućuje korisniku da pri pozivu metode sam definira logiku filtriranja elemenata i način na koji će se podaci zbrajati.

Program.cs:
```
using System.Security.Principal;
 public static U Accumulate<T, U>(
 IEnumerable<T> collection,
 Func<T, U, U> aggregator,
 Predicate<T> match)
{
    U total = default!;
    foreach (var item in collection)
    {
        if (match(item))
        {
            total = aggregator(item, total);
        }
    }
    return total;
}
static void Main(string[] args)
{
    var accounts = new List<Account>
 {
 new Account("Eva", 100m),
 new Account("Ida", 200m),
 new SavingsAccount("Teo", 300m, 5m),
 new SavingsAccount("Edo", 400m, 5m)
 };
    decimal total = Accumulate<Account, decimal>(
    accounts,
    (a, sum) => a.Balance + sum,
    a => a.Balance > 200m
    );
    Console.WriteLine(total);
}
```

Rezultati:
735.00

Zadatak 6. – Generic Delegates with Arrays
U ovom zadatku koristimo .NET metode klase *Array* u kombinaciji s lambda izrazima. Korištenjem klasa *Player* i *Coach*, demonstrirano je kako se nizovi mogu brzo sortirati (*Comparison*), modificirati (*Action*), filtrirati (*Predicate*) ili potpuno transformirati u drugi tip objekta (*Converter*)

Player.cs:
```
using static System.Formats.Asn1.AsnWriter;
class Player
{
    public int Number { get; set; }
    public string Name { get; set; }
    public int Score { get; set; }
    public Player(int number, string name, int score)
    {
        Number = number;
        Name = name;
        Score = score;
    }
    public override string ToString() => $"{Number}.{Name}: { Score}";
}
```
Coach.cs:
```
class Coach
{
    public string Name { get; set; }
    public string? Team { get; set; }
    public Coach(string name)
    {
        Name = name;
    }
    public override string ToString()
    {
        return Name;
    }
}
```

Program.cs:
```
using ArrayDelegates;
using System.Globalization;
using System.Text;
var players = new[]
 {
 new Player(34, "Balić", 5),
 new Player(5, "Duvnjak", 9),
 new Player(27, "Čupić", 8),
 new Player(9, "Vori", 6),
 new Player(1, "Losert", 7)
 };
Console.OutputEncoding = Encoding.UTF8;
Console.WriteLine(CultureInfo.CurrentCulture.Name);
Console.WriteLine("----------");
Array.Sort(players, (a, b) =>
 StringComparer.CurrentCulture.Compare(a.Name, b.Name));
foreach (var p in players)
    Console.WriteLine(p);
Console.WriteLine("----------");
Array.ForEach(players, p => p.Score++);
foreach (var p in players)
    Console.WriteLine(p);
Console.WriteLine("----------");
Player[] filtered = Array.FindAll(players, p => p.Number > 10);
foreach (var p in filtered)
    Console.WriteLine(p);
Console.WriteLine("----------");
Coach[] coaches = Array.ConvertAll(players, p => new Coach(p.Name));
foreach (var c in coaches)
    Console.WriteLine(c);
```