Sprawozdanie z laboratorium

Wyższa Szkoła Ekonomii i Informatyki w Krakowie

Ćw nr:	Temat:
L-1	Implementacja „dobrze uformowanego typu" na przykładzie klasy/struktury Ulamek

Nazwisko i imię:	Nr albumu:	Kierunek:	Rok akademicki:
Kacper Adamczyk	15606	Informatyka Stosowana	2024/2025

Grupa Lab:	Data wykonania:	Data złożenia:	Ocena:
Lab1	18.03.2025	22.03.2025	

1. Opis środowiska pracy

- Komputer nr 1
- System operacyjny: Windows 11
- Środowisko programistyczne: Visual Studio Code
- Platforma .NET: .NET 9.0
- Plik projektowy:

2. Wstęp

Celem tego laboratorium było stworzenie własnego typu danych w C# - klasy `Ulamek`, która reprezentuje ułamek matematyczny. Tworzenie dobrze zaprojektowanego typu to kluczowy element programowania obiektowego, który pozwala na tworzenie zgodnego ze standardami kodu, ułatwiając jego późniejsze wykorzystanie oraz utrzymanie.

Klasa `Ulamek` świetnie nadaje się do ćwiczenia, bo wymaga zaimplementowania różnych mechanizmów C#: konstruktorów, metod, przeciążania operatorów oraz implementacji interfejsów.

3. Cel i zakres pracy

Głównym celem była implementacja klasy `Ulamek`, która pozwala na wykonywanie podstawowych operacji matematycznych na ułamkach zwykłych.

W ramach laboratorium zrealizowałem:

1. Podstawową strukturę klasy z polami na licznik i mianownik
2. Konstruktor zapewniający poprawność danych (brak dzielenia przez zero)
3. Automatyczne upraszczanie ułamków przy pomocy algorytmu Euklidesa
4. Operatory arytmetyczne (+, -, *, /)
5. Operatory porównania (>, <, >=, <=, ==, !=)
6. Interfejsy `IComparable<Ulamek>` i `IEquatable<Ulamek>`
7. Konwersję jawną do typu double
8. Testy sprawdzające poprawność implementacji

4. Metodologia

Pracę nad klasą `Ulamek` podzieliłem na następujące etapy:

1. Stworzenie podstawowej struktury klasy z polami `licznik` i `mianownik`
2. Implementacja konstruktora z walidacją i normalizacją danych
3. Dodanie metody upraszczającej ułamki przy użyciu algorytmu NWD
4. Implementacja podstawowych operatorów arytmetycznych
5. Dodanie operatorów porównania
6. Implementacja interfejsu `IComparable<Ulamek>` do porównywania i sortowania
7. Dodanie interfejsu `IEquatable<Ulamek>` wraz z metodami `Equals` i `GetHashCode`
8. Implementacja konwersji do typu `double`
9. Przygotowanie testów sprawdzających wszystkie funkcjonalności

Podczas pracy korzystałem głównie z dokumentacji Microsoft, szczególnie przy implementacji interfejsów `IComparable` i `IEquatable`, które były najbardziej problematyczne.

5. Opis zadań

5.1. Implementacja klasy Ulamek

Klasa `Ulamek` została zdefiniowana z dwoma podstawowymi polami licznik i ułamek. Chociaż użycie publicznych pól nie jest najlepszą praktyką (narusza hermetyzację), w tym przypadku zastosowałem je dla uproszczenia.

5.2. Konstruktor i upraszczanie ułamków

Zaimplementowałem konstruktor [1], który:

- Sprawdza czy mianownik nie jest zerem
- Normalizuje znak (zawsze dodatni mianownik)
- Automatycznie upraszcza ułamek

Dla upraszczania ułamków napisałem metodę `NajwiekszyWspolnyDzielnik` [2] wykorzystującą algorytm Euklidesa oraz metodę `Uproscic` [3], która dzieli licznik i mianownik przez ich NWD.

5.3. Reprezentacja tekstowa `toString`

Zdefiniowałem prostą reprezentację tekstową ułamka w postaci "licznik/mianownik" [4].

5.4. Operatory arytmetyczne

Zaimplementowałem cztery podstawowe operatory arytmetyczne:

- Mnożenie [5] - mnożenie liczników i mianowników
- Dodawanie [6] - sprowadzenie do wspólnego mianownika
- Odejmowanie [7] - podobnie jak dodawanie
- Dzielenie [8] - mnożenie przez odwrotność drugiego ułamka

Każda operacja zwraca nowy obiekt typu `Ulamek` , co zapewnia niezmienniczość.

5.5. Operatory porównania

Dla porównywania ułamków zaimplementowałem operatory:

- Większy/mniejszy (>, <) [9]
- Większy/mniejszy równy (>=, <=) [10]
- Równość/nierówność (==, !=) [11]

Przy porównywaniu ułamków korzystam z techniki mnożenia krzyżowego: a/b < c/d wtedy i tylko wtedy, gdy a*d* < *c*b.

5.6. Implementacja IComparable i IEquatable

Największym wyzwaniem było zaimplementowanie interfejsów:

- `IComparable<Ulamek>` [12] - do porównywania i sortowania
- `IEquatable<Ulamek>` [13] - do efektywnego porównywania równości

Implementacja interfejsu `IEquatable` była problematyczna, ale dzięki dokumentacji Microsoftu udało się ją poprawnie zaimplementować.

5.7. Konwersje typów

Dodałem jawną konwersję do typu `double` [14], która dzieli licznik przez mianownik. Konwersja jest jawna, bo może wiązać się z utratą precyzji.

6. Analiza wyników

Do sprawdzenia poprawności implementacji klasy `Ulamek` przygotowałem serię testów w metodzie `Main` [15]. Testy obejmowały:

1. Podstawowe operacje arytmetyczne
2. Działanie operatorów porównania
3. Konwersję do typu double
4. Sortowanie tablicy ułamków za pomocą Array.Sort()

Wyniki testów:

```
## Test 1: 1/2 * 1/4 = 1/8
## Test 2: 1/2 + 1/4 = 3/4
## Test 3: 1/2 - 1/4 = 1/4
## Test 4: 1/2 / 1/4 = 2/1
## Test 5: 1/2 > 1/4 ? True
## Test 6: 1/2 < 1/4 ? False
## Test 7: 1/2 >= 1/4 ? True
## Test 8: 1/2 <= 1/4 ? False
## Test 9: (double)1/2 = 0.5
## Test 10: Tablica przed sortowaniem: 1/5, 4/5, 3/5, 2/5, 2/8
## Test 11: Tablica po sortowaniu: 1/5, 1/4, 2/5, 3/5, 4/5
```

Wszystkie testy zakończyły się pomyślnie, co potwierdza poprawność implementacji.

Napotkane problemy:

1. Trudność z implementacją `IComparable` - początkowo nie wiedziałem, jak prawidłowo zaimplementować metodę `CompareTo`. Rozwiązałem problem dzięki dokumentacji Microsoft.
2. Poprawna obsługa znaku ułamka - musiałem zadbać o to, by znak był zawsze przechowywany w liczniku, a mianownik był dodatni.
3. Upraszczanie ułamków - implementacja algorytmu Euklidesa wymagała dokładnego zrozumienia jego działania.

7. Wnioski

Laboratorium pozwoliło mi praktycznie zastosować i lepiej zrozumieć kilka ważnych koncepcji w C#:

1. **Tworzenie "dobrze uformowanego typu"** - nauczyłem się, jakie elementy powinien zawierać poprawnie zaprojektowany typ danych.
2. **Implementacja interfejsów** - poznałem zasady implementacji `IComparable` i `IEquatable`.

Uważam, że zaimplementowany przeze mnie typ `Ulamek` spełnia podstawowe wymagania stawiane "dobrze uformowanym typom" w C#. Klasa:

- Zapewnia poprawne działanie operacji arytmetycznych
- Umożliwia porównywanie i sortowanie
- Jest niemodyfikowalna po utworzeniu
- Automatycznie upraszcza ułamki
- Ma sensowną reprezentację tekstową

8. Odnośniki

[1] Konstruktor klasy Ulamek:

```
public Ulamek(int inLicznik, int inMianownik)
{
    if (inMianownik == 0)
    {
        throw new ArgumentException("nie moze byc zero");
    }

    licznik = inLicznik;
    mianownik = inMianownik;

    if (mianownik < 0)
    {
        licznik = -licznik;
        mianownik = -mianownik;
    }
    Uproscic();
}
```

[2] Metoda NajwiekszyWspolnyDzielnik:

```
public static int NajwiekszyWspolnyDzielnik(int a, int b)
{
    int i = 0;
    while (b != 0)
    {
        ++i;
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

[3] Metoda Uproscic:

```
private void Uproscic()
{
    int nwd = NajwiekszyWspolnyDzielnik(Math.Abs(licznik), Math.Abs(mianownik));

    if (nwd > 1)
    {
        licznik /= nwd;
        mianownik /= nwd;
    }
}
```

[4] Metoda ToString:

```
public override string ToString() => $"{licznik}/{mianownik}";
```

[5] Operator mnożenia:

```
public static Ulamek operator *(Ulamek a, Ulamek b)
{
    int licznik = a.licznik * b.licznik;
    int mianownik = a.mianownik * b.mianownik;
    return new Ulamek(licznik, mianownik);
}
```

[6] Operator dodawania:

```
public static Ulamek operator +(Ulamek a, Ulamek b)
{
    int nowy_licznik = a.licznik * b.mianownik + b.licznik * a.mianownik;
    int nowy_mianownik = a.mianownik * b.mianownik;
    return new Ulamek(nowy_licznik, nowy_mianownik);
}
```

[7] Operator odejmowania:

```
public static Ulamek operator -(Ulamek a, Ulamek b)
{
    int nowy_licznik = a.licznik * b.mianownik - b.licznik * a.mianownik;
    int nowy_mianownik = a.mianownik * b.mianownik;
    return new Ulamek(nowy_licznik, nowy_mianownik);
}
```

[8] Operator dzielenia:

```
public static Ulamek operator /(Ulamek a, Ulamek b)
{
    int licznik = a.licznik * b.mianownik;
    int mianownik = a.mianownik * b.licznik;

    if (mianownik == 0)
        throw new DivideByZeroException("Nie można dzielić przez zero");

    return new Ulamek(licznik, mianownik);
}
```

[9] Operatory > i <:

```
public static bool operator >(Ulamek a, Ulamek b) => a.licznik * b.mianownik > b.licznik * a.mia
public static bool operator <(Ulamek a, Ulamek b) => a.licznik * b.mianownik < b.licznik * a.mia
```

[10] Operatory >= i <=:

```
public static bool operator >=(Ulamek a, Ulamek b) => a.licznik * b.mianownik >= b.licznik * a.r
public static bool operator <=(Ulamek a, Ulamek b) => a.licznik * b.mianownik <= b.licznik * a.r
```

[11] Operatory == i !=:

```
public static bool operator ==(Ulamek? a, Ulamek? b)
{
    if (a is null && b is null) return true;
    if (a is null || b is null) return false;

    return a.licznik == b.licznik && a.mianownik == b.mianownik;
}

public static bool operator !=(Ulamek? a, Ulamek? b) => !(a == b);
```

[12] Implementacja CompareTo:

```
public int CompareTo(Ulamek? other)
{
    if (other == null) return 1;

    // Mnożymy krzyżowo aby porównywać ułamki
    // a/b < c/d jeśli a*d < c*b
    return (licznik * other.mianownik).CompareTo(other.licznik * mianownik);
}
```

[13] Implementacja Equals i GetHashCode:

```
public bool Equals(Ulamek? other)
{
    if (other is null) return false;
    return licznik == other.licznik && mianownik == other.mianownik;
}

public override bool Equals(object? obj)
{
    if (obj is Ulamek other) return Equals(other);
    return false;
}

public override int GetHashCode()
{
    return HashCode.Combine(licznik, mianownik);
}
```

[14] Konwersja na double:

```
public static explicit operator double(Ulamek a) => a.licznik / (double)a.mianownik;
```

[15] Metoda Main z testami:

```
static void Main(string[] args)
{
    Ulamek polowa = new Ulamek(1, 2);
    Ulamek cwierc = new Ulamek(1, 4);

    Console.WriteLine($"Test 1: {polowa} * {cwierc} = {polowa * cwierc}");
    Console.WriteLine($"Test 2: {polowa} + {cwierc} = {polowa + cwierc}");
    Console.WriteLine($"Test 3: {polowa} - {cwierc} = {polowa - cwierc}");
    Console.WriteLine($"Test 4: {polowa} / {cwierc} = {polowa / cwierc}");

    Console.WriteLine($"Test 5: {polowa} > {cwierc} ? {polowa > cwierc}");
    Console.WriteLine($"Test 6: {polowa} < {cwierc} ? {polowa < cwierc}");
    Console.WriteLine($"Test 7: {polowa} >= {cwierc} ? {polowa >= cwierc}");
    Console.WriteLine($"Test 8: {polowa} <= {cwierc} ? {polowa <= cwierc}");

    Console.WriteLine($"Test 9: (double){polowa} = {(double)polowa}");

    // Test tablicy przed i po sortowaniu
    Ulamek[] tablica = { new Ulamek(1, 5), new Ulamek(4, 5), new Ulamek(3, 5),
                         new Ulamek(2, 5), new Ulamek(2, 8) };

    Console.WriteLine("Test 10: Tablica przed sortowaniem:");
    Console.WriteLine(string.Join(", ", tablica.Select(u => u.ToString())));

    Array.Sort(tablica);

    Console.WriteLine("Test 11: Tablica po sortowaniu:");
    Console.WriteLine(string.Join(", ", tablica.Select(u => u.ToString())));
}
```