

WARTOŚCI NIEMODYFIKOWALNE W C#

Dlaczego mutowalne wartości nie zawsze są dobre?

```
class Osoba {  
    public string Imie { get; set; }  
    public string Nazwisko { get; set; }  
    public DateTime DataUrodzenia { get; set; }  
  
    public Osoba(string imie, string nazwisko, DateTime dataUrodzenia)  
    {  
        Imie = imie;  
        Nazwisko = nazwisko;  
        DataUrodzenia = dataUrodzenia;  
    }  
}
```

Założmy, że DateTime jest mutowalny
(w rzeczywistości jest niemutowalny)

Dlaczego mutowalne wartości nie zawsze są dobre?

```
class Program {  
  
    static void Main(string[] args) {  
        var osoba = new Osoba("Ala", "Kot", new DateTime(1998, 01, 12));  
  
        var nowaData = osoba.DataUrodzenia.AddDays(10);  
  
        Console.WriteLine($"Data urodzenia: {osoba.DataUrodzenia}");  
    }  
}
```

Jaka data urodzenia nam się wyświetli (przy założeniu z poprzedniego slajdu)?

Jaka data urodzenia wyświetli się w rzeczywistości?

Dlaczego mutowalne wartości nie zawsze są dobre?

```
class Osoba {  
    public string Imie { get; set; }  
    public string Nazwisko { get; set; }  
    public DateTime DataUrodzenia { get; set; }  
  
    public Osoba(string imie, string nazwisko, DateTime dataUrodzenia) {  
        Imie = imie;  
        Nazwisko = nazwisko;  
        DataUrodzenia = dataUrodzenia;  
    }  
  
    public override int GetHashCode() {  
        return Nazwisko.GetHashCode();  
    }  
}
```

Dlaczego mutowalne wartości nie zawsze są dobre?

```
class Program {
```

```
    static void Main(string[] args) {  
        HashSet<Osoba> hs = new HashSet<Osoba>();  
        var osoba = new Osoba("Ala", "Kot", DateTime.Now);  
        hs.Add(osoba);  
        Console.WriteLine(hs.Contains(osoba));  
        osoba.Nazwisko = "Nowak";  
        Console.WriteLine(hs.Contains(osoba));  
    }  
}
```



true



false

Wartości modyfikowalne

```
class WartoscWZakresie {  
    public int Min { get; set; }  
    public int Max { get; set; }  
    public int Wartosc { get; set; }  
  
    public WartoscWZakresie(int min, int max, int wartosc) {  
        Min=min;  
        Max=max;  
        Wartosc = wartosc;  
    }  
  
    public void ZwiekszWartosc(int oWartosc) {  
        Wartosc += oWartosc;  
    }  
}
```

Wartości modyfikowalne

```
WartoscWZakresie w  
    = new WartoscWZakresie(min:-10, max:10, wartosc: 5);
```

```
WartoscWZakresie w1  
    = new WartoscWZakresie(10, 0, 5);
```



Obiekt jest całkowicie w błędnym stanie

Jeżeli mam obiekt to jest on poprawny

Wartości modyfikowalne

```
class WartoscWZakresie {  
    public int Min { get; set; }  
    public int Max { get; set; }  
    public int Wartosc { get; set; }  
  
    public WartoscWZakresie(int min, int max, int wartosc) {  
        Min=min;  
        Max=max;  
        Wartosc = wartosc;  
        SprawdzZakres();  
        SprawdzWartosc();  
    }  
    ...  
}
```


Wartości modyfikowalne

```
class WartoscWZakresie {  
    ...  
    private void SprawdzZakres() {  
        if(Min>Max)  
            throw new Exception();  
    }  
  
    private void SprawdzWartosc() {  
        if(wartosc<Min)  
            wartosc = Min;  
        if(wartosc>Max)  
            wartosc = Max;  
    }  
}
```

Wartości modyfikowalne

```
class WartoscWZakresie {  
  
    public int Min { get; private set; }  
    public int Max { get; private set; }  
    public int Wartosc { get; private set; }  
  
    void ZwiekszWartosc(int oWartosc) {  
        Wartosc += oWartosc;  
    }  
  
}
```

Wartości modyfikowalne

```
class WartoscWZakresie {  
  
    void ZwiekszWartosc(int oWartosc) {  
        Wartosc += oWartosc;  
        SprawdzWartosc();  
    }  
  
}
```

Wartości modyfikowalne

```
class WartoscWZakresie {  
  
    private int _min;  
  
    public int Min {  
        get { return _min; }  
        set { _min = value; }  
    }  
  
}
```

Wartości modyfikowalne

```
class WartoscWZakresie {  
  
    private int _min;  
    private int _max;  
  
    public int Min {  
        get { return _min; }  
        set {  
            if(value>_max)  
                throw new Exception();  
            _min = value;  
            SprawdzWartosc();  
        }  
    }  
}
```

Wartości niemodyfikowalne

Wartości niemodyfikowalne są to wartości, których składowe mogą być ustalone tylko w konstruktorze.

Wartości niemodyfikowalne

```
class WartoscWZakresie {  
  
    private readonly int _min;  
    private readonly int _max;  
    private readonly int _wartosc;  
  
    public int Min { get => _min; }  
    public int Max { get => _max; }  
    public int Wartosc { get => _wartosc; }  
  
    public WartoscWZakresie(int min, int max, int wartosc) {  
        _min=min;  
        _max=max;  
        _wartosc = wartosc;  
        SprawdzZakres();  
        SprawdzWartosc();  
    }  
    ...  
}
```

Wartości niemodyfikowalne

```
class WartoscWZakresie {  
  
    ...  
  
    public WartoscWZakresie ZwiekszWartosc(int oWartosc) {  
        return new WartoscWZakresie(_min, _max, _wartosc+oWartosc);  
    }  
  
    public WartoscWZakresie ZmienMin(int min) {  
        return new WartoscWZakresie(min, _max, _wartosc);  
    }  
}
```


Wartości niemodyfikowalne

```
class WartoscWZakresie {  
  
    ...  
  
    public WartoscWZakresie ZwiekszWartosc(int oWartosc) {  
        return new WartoscWZakresie(_min, _max, _wartosc+oWartosc);  
    }  
  
    public WartoscWZakresie ZmienMin(int min) {  
        return new WartoscWZakresie(min, _max, _wartosc);  
    }  
}
```

Wartości niemodyfikowalne

```
class WartoscWZakresie {  
  
    public int Min { get; }  
    public int Max { get; }  
    public int Wartosc { get; }  
  
    public WartoscWZakresie(int min, int max, int wartosc) {  
        Min=min;  
        Max=max;  
        Wartosc = wartosc;  
        SprawdzZakres();  
        SprawdzWartosc();  
    }  
    ...  
}
```

Czy ta klasa jest poprawnie zapisana?

Zasada pojedynczej odpowiedzialności

Nigdy nie powinno być więcej niż jednego powodu do istnienia klasy lub metody

Tom DeMarco, Meilir Page-Jones

Nigdy nie powinno być więcej niż jednego powodu do modyfikacji klasy lub metody

Robert C. Martin

Stwórzmy dwie klasy

```
public class Zakres {
    public int Min { get; }
    public int Max { get; }

    public Zakres(int min, int max) {
        if (min > max) throw new ArgumentException("...");
        Min = min;
        Max = max;
    }

    public Zakres ZmienMin(int min) => new Zakres(min, Max);
    public Zakres ZmienMax(int max) => new Zakres(Min, max);
    public bool WZakresie(int wartosc) => Min <= wartosc && wartosc <= Max;
}
```

Stwórzmy dwie klasy

```
public class WartoscWZakresie {  
  
    public Zakres Zakres { get; }  
    public int Wartosc { get; }  
  
    public WartoscWZakresie(Zakres zakres, int wartosc) {  
  
        Zakres = zakres;  
        if (wartosc < zakres.Min) Wartosc = zakres.Min;  
        else if (zakres.Max < wartosc) Wartosc = zakres.Max;  
        else Wartosc = wartosc;  
    }  
  
}
```

Czy klasa Zakres jest rzeczywiście niemutowalna?

```
public class NiemutowalnyZakres {  
  
    public int Min { get; }  
    public int Max { get; }  
  
    public NiemutowalnyZakres(int min, int max) {  
        if (min > max) throw new ArgumentException("...");  
        Min = min;  
        Max = max;  
    }  
  
    public override string ToString() {  
        return $"[{Min} {Max}]";  
    }  
}
```

Czy klasa Zakres jest rzeczywiście niemutowalna?

```
class Program {  
    static void Main(string[] args) {  
        NiemutowalnyZakres niezmiennik = new NiemutowalnyZakres(-10, 10);  
  
        Console.WriteLine(niezmiennik);  
        ...  
        Console.WriteLine(niezmiennik);  
    }  
}
```

[-10,10]

[-10,10]



Czy klasa Zakres jest rzeczywiście niemutowalna?

```
public class ZakresZNazwa : NiemutowalnyZakres {  
  
    public string Nazwa { get; set; }  
  
    public ZakresZNazwa(int min, int max, string nazwa)  
        : base(min, max)  
    {  
        Nazwa = nazwa;  
    }  
  
    public override string ToString() {  
        return $"{Nazwa}: [{Min} {Max}]";  
    }  
  
}
```


Czy klasa Zakres jest rzeczywiście niemutowalna?

```
class Program {
```

```
    static void Main(string[] args) {
```

```
        var zakres = new ZakresZNazwa(-10, 10, "Jakiś zakres");
```

```
        NiemutowalnyZakres niezmiennik = zakres;
```

```
        Console.WriteLine(niezmiennik);
```

Jakiś zakres [-10,10]

```
        zakres.Nazwa = "Nowa nazwa";
```

```
        Console.WriteLine(niezmiennik);
```

Nowa nazwa [-10,10]



```
    }
```

```
}
```

Jak się przed taką sytuacją obronić?

```
public sealed class NiemutowalnyZakres {  
  
    public int Min { get; }  
    public int Max { get; }  
  
    public NiemutowalnyZakres(int min, int max) {  
        if (min > max) throw new ArgumentException("...");  
        Min = min;  
        Max = max;  
    }  
  
    public override string ToString() {  
        return $"[{Min} {Max}]";  
    }  
}
```

Przykład bardziej praktyczny

Musimy stworzyć klasę Osoba w której:

- Imię musi być podane, ale mieć nie więcej niż 30 znaków
- Nazwisko musi być podane, ale mieć nie więcej niż 30 znaków
- Wiek może być opcjonalny, ale z sensownego zakresu

Klasa Osoba

```
public sealed class Osoba {  
    public string Imie { get; }  
    public string Nazwisko { get; }  
    public int Wiek { get; }  
  
    public Osoba(string imie, string nazwisko, int wiek)  
    {  
        ...  
    }  
}
```

Klasy definiujące wartości

```
class Str30 {  
    public string Wartosc { get; }  
  
    public Str30(string wartosc) {  
        if (string.IsNullOrEmpty(wartosc) || 30 < wartosc.Length)  
            throw new ArgumentException();  
        else  
            Wartosc = wartosc;  
    }  
  
    public static implicit operator string(Str30 str30)  
        => str30.Wartosc;  
    public static implicit operator Str30(string wartosc)  
        => new Str30(wartosc);  
}
```

Klasy definiujące wartości

```
class Wiek {  
    public int Wartosc { get; }  
  
    public Wiek(int wartosc) {  
        if (wartosc < 0 || 140 < wartosc)  
            throw new ArgumentException("...");  
        Wartosc = wartosc;  
    }  
  
    public static implicit operator int(Wiek wiek)  
        => wiek.Wartosc;  
    public static implicit operator Wiek(int wartosc)  
        => new Wiek(wartosc);  
}
```

Klasa Osoba

```
class Osoba {

    public Str30 Imie { get; }
    public Str30 Nazwisko { get; }
    public Wiek Wiek { get; }

    public Osoba(Str30 imie, Str30 nazwisko, Wiek wiek) {
        if (imie == null)
            throw new ArgumentNullException(nameof(imie));
        else
            Imie = imie;
        if (nazwisko == null)
            throw new ArgumentNullException(nameof(nazwisko));
        else
            Nazwisko = nazwisko;
    }
}
```

Klasa Osoba

```
class Osoba {
    public Str30 Imie { get; }
    public Str30 Nazwisko { get; }
    public Wiek Wiek { get; }

    public Osoba(Str30 imie, Str30 nazwisko, Wiek wiek) {
        Imie = imie ?? throw new ArgumentNullException(nameof(imie));
        Nazwisko =
            nazwisko ?? throw new ArgumentNullException(nameof(nazwisko));
        Wiek = wiek;
    }
}
```


Klasa Osoba

```
var osoba1 = new Osoba("Tomek", "Nowak", 23);
```

```
var osoba2 = new Osoba("Ala", "Kot", null);
```

```
var osoba3 = new Osoba("Tomek", "Nowak")  
{  
    Wiek = 23  
};
```

Klasa Osoba

```
class Osoba {  
    public Str30 Imie { get; }  
    public Str30 Nazwisko { get; }  
    public Wiek Wiek { get; set; }  
  
    public Osoba(Str30 imie, Str30 nazwisko) {  
        Imie =  
            imie ?? throw new ArgumentNullException(nameof(imie));  
        Nazwisko =  
            nazwisko ?? throw new ArgumentNullException(nameof(nazwisko));  
    }  
}
```

Właściwości tylko do inicjalizacji

```
class Osoba {  
    public Str30 Imie { get; }  
    public Str30 Nazwisko { get; }  
    public Wiek Wiek { get; init; }  
  
    public Osoba(Str30 imie, Str30 nazwisko) {  
        Imie =  
            imie ?? throw new ArgumentNullException(nameof(imie));  
        Nazwisko =  
            nazwisko ?? throw new ArgumentNullException(nameof(nazwisko));  
    }  
}
```

Co zrobić, aby zrezygnować ze sprawdzania null?

```
struct Str30 {  
    public string Wartosc { get; }  
  
    public Str30(string wartosc) {  
        if (string.IsNullOrEmpty(wartosc) || 30 < wartosc.Length)  
            throw new ArgumentException();  
        else  
            Wartosc = wartosc;  
    }  
  
    public static implicit operator string(Str30 str30)  
        => str30.Wartosc;  
    public static implicit operator Str30(string wartosc)  
        => new Str30(wartosc);  
}
```

Klasa Osoba

```
class Osoba
{
    public Str30 Imie { get; }
    public Str30 Nazwisko { get; }
    public Wiek Wiek { get; }

    public Osoba(Str30 imie, Str30 nazwisko)
    {
        Imie = imie;
        Nazwisko = nazwisko;
    }
}
```

Klasa Osoba

```
class Osoba
{
    public Str30 Imie { get; }
    public Str30 Nazwisko { get; }
    public Wiek? Wiek { get; }

    public Osoba(Str30 imie, Str30 nazwisko)
    {
        Imie = imie;
        Nazwisko = nazwisko;
    }
}
```

Rekordy

C# 9 wprowadza nowy rodzaj typu danych - rekord

```
public record Osoba (int Id, string Imie, string Nazwisko)
```

```
public record Osoba (int Id, string Imie, string Nazwisko)
{
    public int Wiek {get;set;}
}
```

- Właściwości pozycyjne są niemutowalne.
- Rekordy porównywane są strukturalnie

Rekordy

Nieniszcząca mutacja

```
var osoba = new Osoba("Ala", "Kot", 23);  
var nowaOsoba = osoba with {Nazwisko = "Nowak"}
```

```
public record Osoba (int Id, string Imie, string Nazwisko)  
{  
    public int Wiek {get;set;}  
}
```


Dziedziczenie rekordów

Rekordy mogą dziedziczyć tylko po rekordach

Niedopuszczalne jest dziedziczenie rekordów po klasach oraz klas po rekordach

```
public record Osoba(string Imie, string Nazwisko);
```

```
public record Student(string Imie, string Nazwisko, int NrIndeksu)  
    : Osoba(Imie, Nazwisko);
```

Dekonstrukcja rekordów

```
var osoba = new Osoba("Ala", "Kot", 23);  
  
var (imie, nazwisko) = osoba;
```

Domyślna dekonstrukcja dotyczy wyłączenie właściwości pozycyjnych

Dekonstrukcja obiektów

```
class Osoba {  
    public string Imie { get; set; }  
    public string Nazwisko { get; set; }  
  
    public Osoba(string imie, string nazwisko) {  
        Imie = imie;  
        Nazwisko = nazwisko;  
    }  
  
    public void Deconstruct(out string imie, out string nazwisko) {  
        imie = Imie;  
        nazwisko = Nazwisko;  
    }  
}
```

System.Collections.Immutable

Niemutowalne kolekcje dostępne
są w przestrzeni nazw
System.Collections.Immutable

System.Collections.Immutable

ImmutableArray

ImmutableDictionary

ImmutableHashSet

ImmutableList

ImmutableQueue

ImmutableSortedDictionary

ImmutableStack

System.Collections.Immutable

```
var lista = ImmutableList.Create<Osoba>();  
  
var nowaLista = lista.Add(new Osoba("Ala", "Kot", DateTime.Now));  
  
Console.WriteLine(lista.Count);  
Console.WriteLine(nowaLista.Count);
```

Immutable vs Readonly

```
var lista = ImmutableList.Create<Osoba>();  
var nowaLista = lista.Add(new Osoba("Ala", "Kot", DateTime.Now));
```

```
var listaDO = new List<Osoba>  
    {new Osoba("Ala", "Kot", DateTime.Now)}  
    .AsReadOnly();
```

```
listaDO.Add(new Osoba("Ala", "Kot", DateTime.Now));
```

Immutable vs Readonly

```
var lista = new List<Osoba>  
            { new Osoba("Ala", "Kot", DateTime.Now) };  
listaDO = lista.AsReadOnly();
```

```
Console.WriteLine(listaDO.Count);
```

1

```
lista.Add(new Osoba("Ewa", "Nowak", DateTime.Now));
```

```
Console.WriteLine(listaDO.Count);
```

2!!!!!!