Erweiterte Sprachfeatures in C#

Tiefes Eintauchen in moderne C#-Funktionen

Mühlehner & Tavolato GmbH

Agenda

- Spracherweiterungen und neue Syntax-Optionen

  unvollständige Liste
- Discards und Read-only Structs
- Private Protected Modifier
- Default Literals
- Expression-Bodied Members
- Pattern Matching: Konzepte und Anwendungsfälle

Spracherweiterungen

seit C# 12 & .NET 8

```
1   // type alias
2   using Customer = (string name,int id);
3
4   using System.Collections.Generic; using System; using System.Linq;
5
6   // primary constructor
7   class Library(string name) {
8
9     // collection expressions/s (public ist nur zur Demo eines anderen Features)
10    public List<string> books = ["The Pragmatic Programmer","Designing Data-Intense Applications"];
11    // lambda with optional parameter
12    private readonly Func<int,int,int> sum = (int a, int b = 0) ⇒ a + b;
13    // params collections IEnumerable, List<>, usw. fuer params keyword
14    // lambda with params modifier
15    private readonly Func<IEnumerable<Customer>,int> memberLength = (params IEnumerable<Customer> c) ⇒ c.Count();
16
17    // field contextual keyword (statt explizitem Feld)
18    public int Hours {
19        get;
20        set ⇒ field = ( value ⩾ 0) ? value : 0;
21    }
22
23  }
```

Spracherweiterungen

seit C# 12 & .NET 8

```
1   // type alias
2   using Customer = (string name,int id);
3
4   using System.Collections.Generic; using System; using System.Linq;
5
6   // primary constructor
7   class Library(string name) {
8
9     // collection expressions/s (public ist nur zur Demo eines anderen Features)
10    public List<string> books = ["The Pragmatic Programmer","Designing Data-Intense Applications"];
11    // lambda with optional parameter
12    private readonly Func<int,int,int> sum = (int a, int b = 0) => a + b;
13    // params collections IEnumerable, List<>, usw. fuer params keyword
14    // lambda with params modifier
15    private readonly Func<IEnumerable<Customer>,int> memberLength = (params IEnumerable<Customer> c) => c.Count();
16
17    // field contextual keyword (statt explizitem Feld)
18    public int Hours {
19        get;
20        set => field = ( value >= 0) ? value : 0;
21    }
22
23  }
```

Spracherweiterungen

seit C# 12 & .NET 8

```
// type alias
using Customer = (string name,int id);

using System.Collections.Generic; using System; using System.Linq;

// primary constructor
class Library(string name) {

  // collection expressions/s (public ist nur zur Demo eines anderen Features)
  public List<string> books = ["The Pragmatic Programmer","Designing Data-Intense Applications"];
  // lambda with optional parameter
  private readonly Func<int,int,int> sum = (int a, int b = 0) => a + b;
  // params collections IEnumerable, List<>, usw. fuer params keyword
  // lambda with params modifier
  private readonly Func<IEnumerable<Customer>,int> memberLength = (params IEnumerable<Customer> c) => c.Count();

  // field contextual keyword (statt explizitem Feld)
  public int Hours {
      get;
      set => field = ( value >= 0) ? value : 0;
  }

}
```

Spracherweiterungen

seit C# 12 & .NET 8

```
1   // type alias
2   using Customer = (string name,int id);
3
4   using System.Collections.Generic; using System; using System.Linq;
5
6   // primary constructor
7   class Library(string name) {
8
9     // collection expressions/s (public ist nur zur Demo eines anderen Features)
10    public List<string> books = ["The Pragmatic Programmer","Designing Data-Intense Applications"];
11    // lambda with optional parameter
12    private readonly Func<int,int,int> sum = (int a, int b = 0) => a + b;
13    // params collections IEnumerable, List<>, usw. fuer params keyword
14    // lambda with params modifier
15    private readonly Func<IEnumerable<Customer>,int> memberLength = (params IEnumerable<Customer> c) => c.Count();
16
17    // field contextual keyword (statt explizitem Feld)
18    public int Hours {
19        get;
20        set => field = ( value >= 0) ? value : 0;
21    }
22
23  }
```

Spracherweiterungen

seit C# 12 & .NET 8

```
// type alias
using Customer = (string name,int id);

using System.Collections.Generic; using System; using System.Linq;

// primary constructor
class Library(string name) {

  // collection expressions/s (public ist nur zur Demo eines anderen Features)
  public List<string> books = ["The Pragmatic Programmer","Designing Data-Intense Applications"];
  // lambda with optional parameter
  private readonly Func<int,int,int> sum = (int a, int b = 0) => a + b;
  // params collections IEnumerable, List<>, usw. fuer params keyword
  // lambda with params modifier
  private readonly Func<IEnumerable<Customer>,int> memberLength = (params IEnumerable<Customer> c) => c.Count();

  // field contextual keyword (statt explizitem Feld)
  public int Hours {
      get;
      set => field = ( value >= 0) ? value : 0;
  }

}
```

Spracherweiterungen

seit C# 12 & .NET 8

```
1   // type alias
2   using Customer = (string name,int id);
3
4   using System.Collections.Generic; using System; using System.Linq;
5
6   // primary constructor
7   class Library(string name) {
8
9     // collection expressions/s (public ist nur zur Demo eines anderen Features)
10    public List<string> books = ["The Pragmatic Programmer","Designing Data-Intense Applications"];
11    // lambda with optional parameter
12    private readonly Func<int,int,int> sum = (int a, int b = 0) => a + b;
13    // params collections IEnumerable, List<>, usw. fuer params keyword
14    // lambda with params modifier
15    private readonly Func<IEnumerable<Customer>,int> memberLength = (params IEnumerable<Customer> c) => c.Count();
16
17    // field contextual keyword (statt explizitem Feld)
18    public int Hours {
19        get;
20        set => field = ( value >= 0) ? value : 0;
21    }
22
23  }
```

Spracherweiterungen

seit C# 12 & .NET 8

```
// type alias
using Customer = (string name,int id);

using System.Collections.Generic; using System; using System.Linq;

// primary constructor
class Library(string name) {

  // collection expressions/s (public ist nur zur Demo eines anderen Features)
  public List<string> books = ["The Pragmatic Programmer","Designing Data-Intense Applications"];
  // lambda with optional parameter
  private readonly Func<int,int,int> sum = (int a, int b = 0) => a + b;
  // params collections IEnumerable, List<>, usw. fuer params keyword
  // lambda with params modifier
  private readonly Func<IEnumerable<Customer>,int> memberLength = (params IEnumerable<Customer> c) => c.Count();

  // field contextual keyword (statt explizitem Feld)
  public int Hours {
      get;
      set => field = ( value >= 0) ? value : 0;
  }

}
```

Spracherweiterungen

seit C# 12 & .NET 8

```
1   // type alias
2   using Customer = (string name,int id);
3
4   using System.Collections.Generic; using System; using System.Linq;
5
6   // primary constructor
7   class Library(string name) {
8
9     // collection expressions/s (public ist nur zur Demo eines anderen Features)
10    public List<string> books = ["The Pragmatic Programmer","Designing Data-Intense Applications"];
11    // lambda with optional parameter
12    private readonly Func<int,int,int> sum = (int a, int b = 0) ⇒ a + b;
13    // params collections IEnumerable, List<>, usw. fuer params keyword
14    // lambda with params modifier
15    private readonly Func<IEnumerable<Customer>,int> memberLength = (params IEnumerable<Customer> c) ⇒ c.Count();
16
17    // field contextual keyword (statt explizitem Feld)
18    public int Hours {
19        get;
20        set ⇒ field = ( value ⩾ 0) ? value : 0;
21    }
22
23  }
```

Spracherweiterungen

seit C# 12 & .NET 8

```
1   // implicit index access in an object initializer expression
2   var lib = new Library("National Bibliothek") {
3       books = { // Length = 2
4           [^1] = "Don Carlos", // letztes Element aka [1]
5           [^2] = "Eine kurze Geschichte der Zeit" // vorletztes aka [0]
6       }
7   };
8
9   Console.WriteLine(lib.books[0]); // Eine kurze Geschichte der Zeit
10  Console.WriteLine(lib.books[1]); // Don Carlos
```

Discards

```
1   // Beispiel für Discards
2   var (x, _, z) = GetCoordinates();
3   _ = DoSomething(); // Rückgabewert ignorieren
```

Discards

- **Discards (_)**: Platzhalter für ungenutzte Variablen

```
1   // Beispiel für Discards
2   var (x, _, z) = GetCoordinates();
3   _ = DoSomething(); // Rückgabewert ignorieren
```

Read-only Structs

```
1   // Beispiel für Read-only Struct
2   readonly struct Point
3   {
4       public readonly byte Z;
5
6       public double X { get; init; }  // init ist okay
7       public double Y { get; } // setter NICHT okay
8       public double Distance => Math.Sqrt(X * X + Y * Y + Z * Z);
9   }
```

Read-only Structs

- **Read-only Structs**: Unveränderliche Value Types

```
1   // Beispiel für Read-only Struct
2   readonly struct Point
3   {
4       public readonly byte Z;
5
6       public double X { get; init; }  // init ist okay
7       public double Y { get; } // setter NICHT okay
8       public double Distance ⇒ Math.Sqrt(X * X + Y * Y + Z * Z);
9   }
```

Private Protected Modifier

```
public class Base
{
    private protected void InternalMethod()
    {
        // Nur für die Basisklasse und abgeleitete Klassen
        // in derselben Assembly zugänglich
    }
}

public class Derived : Base
{
    public void AccessBase()
    {
        InternalMethod(); // Funktioniert innerhalb derselben Assembly
    }
}
```

Private Protected Modifier

- Kombinierter Zugriffsmodifikator

```
1   public class Base
2   {
3       private protected void InternalMethod()
4       {
5           // Nur für die Basisklasse und abgeleitete Klassen
6           // in derselben Assembly zugänglich
7       }
8   }
9
10  public class Derived : Base
11  {
12      public void AccessBase()
13      {
14          InternalMethod(); // Funktioniert innerhalb derselben Assembly
15      }
16  }
```

Private Protected Modifier

- Kombinierter Zugriffsmodifikator
- Zugriff beschränkt auf:
  - Dieselbe Assembly
  - Abgeleitete Klassen

```
1   public class Base
2   {
3       private protected void InternalMethod()
4       {
5           // Nur für die Basisklasse und abgeleitete Klassen
6           // in derselben Assembly zugänglich
7       }
8   }
9
10  public class Derived : Base
11  {
12      public void AccessBase()
13      {
14          InternalMethod(); // Funktioniert innerhalb derselben Assembly
15      }
16  }
```

Default Literals

```
// Alte Syntax
int oldDefault = default(int);

// Neue Syntax mit default literal
int newDefault = default;
string text = default;
DateTime date = default;

// In Methodenaufrufen
Process(default);
```

Default Literals

- Vereinfachte Syntax für Standardwerte

```
// Alte Syntax
int oldDefault = default(int);

// Neue Syntax mit default literal
int newDefault = default;
string text = default;
DateTime date = default;

// In Methodenaufrufen
Process(default);
```

Default Literals

- Vereinfachte Syntax für Standardwerte
- Typ wird vom Kontext abgeleitet

```
// Alte Syntax
int oldDefault = default(int);

// Neue Syntax mit default literal
int newDefault = default;
string text = default;
DateTime date = default;

// In Methodenaufrufen
Process(default);
```

Default Literals

- Vereinfachte Syntax für Standardwerte
- Typ wird vom Kontext abgeleitet

```
// Alte Syntax
int oldDefault = default(int);

// Neue Syntax mit default literal
int newDefault = default;
string text = default;
DateTime date = default;

// In Methodenaufrufen
Process(default);
```

Default Literals

- Vereinfachte Syntax für Standardwerte
- Typ wird vom Kontext abgeleitet

```
// Alte Syntax
int oldDefault = default(int);

// Neue Syntax mit default literal
int newDefault = default;
string text = default;
DateTime date = default;

// In Methodenaufrufen
Process(default);
```

Default Literals

- Vereinfachte Syntax für Standardwerte
- Typ wird vom Kontext abgeleitet

```
// Alte Syntax
int oldDefault = default(int);

// Neue Syntax mit default literal
int newDefault = default;
string text = default;
DateTime date = default;

// In Methodenaufrufen
Process(default);
```

Default Literals

- Vereinfachte Syntax für Standardwerte
- Typ wird vom Kontext abgeleitet

```
// Alte Syntax
int oldDefault = default(int);

// Neue Syntax mit default literal
int newDefault = default;
string text = default;
DateTime date = default;

// In Methodenaufrufen
Process(default);
```

Expression-Bodied Members

```
1   // Properties
2   public string FullName => $"{FirstName} {LastName}";
3
4   // Methoden
5   public string GetGreeting() => $"Hello, {FullName}!";
6
7   // Konstruktoren
8   public Person(string name) => Name = name;
9
10  // Finalizers
11  ~Person() => Dispose(false);
```

Expression-Bodied Members

- Kurzschreibweise für einfache Member-Definitionen

```
1   // Properties
2   public string FullName ⇒ $"{FirstName} {LastName}";
3
4   // Methoden
5   public string GetGreeting() ⇒ $"Hello, {FullName}!";
6
7   // Konstruktoren
8   public Person(string name) ⇒ Name = name;
9
10  // Finalizers
11  ~Person() ⇒ Dispose(false);
```

Expression-Bodied Members

- Kurzschreibweise für einfache Member-Definitionen
- Verbessert Lesbarkeit und Wartbarkeit

```
// Properties
public string FullName => $"{FirstName} {LastName}";

// Methoden
public string GetGreeting() => $"Hello, {FullName}!";

// Konstruktoren
public Person(string name) => Name = name;

// Finalizers
~Person() => Dispose(false);
```

Expression-Bodied Members

- Kurzschreibweise für einfache Member-Definitionen
- Verbessert Lesbarkeit und Wartbarkeit

```
1   // Properties
2   public string FullName => $"{FirstName} {LastName}";
3
4   // Methoden
5   public string GetGreeting() => $"Hello, {FullName}!";
6
7   // Konstruktoren
8   public Person(string name) => Name = name;
9
10  // Finalizers
11  ~Person() => Dispose(false);
```

Expression-Bodied Members

- Kurzschreibweise für einfache Member-Definitionen
- Verbessert Lesbarkeit und Wartbarkeit

```
1   // Properties
2   public string FullName => $"{FirstName} {LastName}";
3
4   // Methoden
5   public string GetGreeting() => $"Hello, {FullName}!";
6
7   // Konstruktoren
8   public Person(string name) => Name = name;
9
10  // Finalizers
11  ~Person() => Dispose(false);
```

Expression-Bodied Members

- Kurzschreibweise für einfache Member-Definitionen
- Verbessert Lesbarkeit und Wartbarkeit

```
1   // Properties
2   public string FullName => $"{FirstName} {LastName}";
3
4   // Methoden
5   public string GetGreeting() => $"Hello, {FullName}!";
6
7   // Konstruktoren
8   public Person(string name) => Name = name;
9
10  // Finalizers
11  ~Person() => Dispose(false);
```

Expression-Bodied Members

- Kurzschreibweise für einfache Member-Definitionen
- Verbessert Lesbarkeit und Wartbarkeit

```
1   // Properties
2   public string FullName => $"{FirstName} {LastName}";
3
4   // Methoden
5   public string GetGreeting() => $"Hello, {FullName}!";
6
7   // Konstruktoren
8   public Person(string name) => Name = name;
9
10  // Finalizers
11  ~Person() => Dispose(false);
```

Expression-Bodied Members

- Kurzschreibweise für einfache Member-Definitionen
- Verbessert Lesbarkeit und Wartbarkeit

```
1   // Properties
2   public string FullName => $"{FirstName} {LastName}";
3
4   // Methoden
5   public string GetGreeting() => $"Hello, {FullName}!";
6
7   // Konstruktoren
8   public Person(string name) => Name = name;
9
10  // Finalizers
11  ~Person() => Dispose(false);
```

Pattern Matching: Grundlagen

```csharp
// Grundlegendes Pattern Matching
object value = "Hello";

if (value is string message)
{
    // 'message' ist jetzt als string typisiert
    Console.WriteLine(message.Length);
}

// Switch Expression mit Pattern Matching
string GetDescription(object obj) => obj switch
{
    null => "Nothing",
    int i => $"Number: {i}",
    string s => $"Text: {s}",
    _ => "Unknown"
};
```

Pattern Matching: Grundlagen

- Leistungsstarke Technik zur Datenanalyse

```
// Grundlegendes Pattern Matching
object value = "Hello";

if (value is string message)
{
    // 'message' ist jetzt als string typisiert
    Console.WriteLine(message.Length);
}

// Switch Expression mit Pattern Matching
string GetDescription(object obj) => obj switch
{
    null => "Nothing",
    int i => $"Number: {i}",
    string s => $"Text: {s}",
    _ => "Unknown"
};
```

Pattern Matching: Grundlagen

- Leistungsstarke Technik zur Datenanalyse
- Ersetzt komplexe if/else oder switch-Konstrukte

```
// Grundlegendes Pattern Matching
object value = "Hello";

if (value is string message)
{
    // 'message' ist jetzt als string typisiert
    Console.WriteLine(message.Length);
}

// Switch Expression mit Pattern Matching
string GetDescription(object obj) => obj switch
{
    null => "Nothing",
    int i => $"Number: {i}",
    string s => $"Text: {s}",
    _ => "Unknown"
};
```

Pattern Matching: Grundlagen

- Leistungsstarke Technik zur Datenanalyse
- Ersetzt komplexe if/else oder switch-Konstrukte
- Mehrere Arten von Patterns:
  - Type Patterns
  - Property Patterns
  - Tuple Patterns
  - Positional Patterns
  - Logical Patterns (and, or, not)

```
1   // Grundlegendes Pattern Matching
2   object value = "Hello";
3
4   if (value is string message)
5   {
6       // 'message' ist jetzt als string typisiert
7       Console.WriteLine(message.Length);
8   }
9
10  // Switch Expression mit Pattern Matching
11  string GetDescription(object obj) => obj switch
12  {
13      null => "Nothing",
14      int i => $"Number: {i}",
15      string s => $"Text: {s}",
16      _ => "Unknown"
17  };
```

Erweiterte Pattern Matching Beispiele

```
// Property Pattern
bool IsValidPerson(Person person) => person is
{
    Age: >= 18,
    Name: { Length: > 0 },
    Address: not null,
    Role: "Admin" or "Manager"
};

// Kombinierte Patterns
string GetWeatherAdvice(Weather weather) => weather switch
{
    { Season: "Summer", Temperature: > 30 } => "Stay hydrated",
    { Season: "Winter", Temperature: < 0 } => "Dress warmly",
    { IsRaining: true } => "Take an umbrella",
    { IsWindy: true, Temperature: < 15 } => "Take a jacket",
    _ => "Enjoy your day"
};
```

Erweiterte Pattern Matching Beispiele

```
// Property Pattern
bool IsValidPerson(Person person) ⇒ person is
{
    Age: ⩾ 18,
    Name: { Length: > 0 },
    Address: not null,
    Role: "Admin" or "Manager"
};

// Kombinierte Patterns
string GetWeatherAdvice(Weather weather) ⇒ weather switch
{
    { Season: "Summer", Temperature: > 30 } ⇒ "Stay hydrated",
    { Season: "Winter", Temperature: < 0 } ⇒ "Dress warmly",
    { IsRaining: true } ⇒ "Take an umbrella",
    { IsWindy: true, Temperature: < 15 } ⇒ "Take a jacket",
    _ ⇒ "Enjoy your day"
};
```

Erweiterte Pattern Matching Beispiele

```
// Property Pattern
bool IsValidPerson(Person person) => person is
{
    Age: >= 18,
    Name: { Length: > 0 },
    Address: not null,
    Role: "Admin" or "Manager"
};

// Kombinierte Patterns
string GetWeatherAdvice(Weather weather) => weather switch
{
    { Season: "Summer", Temperature: > 30 } => "Stay hydrated",
    { Season: "Winter", Temperature: < 0 } => "Dress warmly",
    { IsRaining: true } => "Take an umbrella",
    { IsWindy: true, Temperature: < 15 } => "Take a jacket",
    _ => "Enjoy your day"
};
```

Erweiterte Pattern Matching Beispiele

```
// Property Pattern
bool IsValidPerson(Person person) ⇒ person is
{
    Age: ⩾ 18,
    Name: { Length: > 0 },
    Address: not null,
    Role: "Admin" or "Manager"
};

// Kombinierte Patterns
string GetWeatherAdvice(Weather weather) ⇒ weather switch
{
    { Season: "Summer", Temperature: > 30 } ⇒ "Stay hydrated",
    { Season: "Winter", Temperature: < 0 } ⇒ "Dress warmly",
    { IsRaining: true } ⇒ "Take an umbrella",
    { IsWindy: true, Temperature: < 15 } ⇒ "Take a jacket",
    _ ⇒ "Enjoy your day"
};
```

Mühlehner & Tavolato GmbH

Praxistipps

Praxistipps

- Pattern Matching verbessert Code-Lesbarkeit

Praxistipps

- Pattern Matching verbessert Code-Lesbarkeit
- Kombiniere Features für sauberen Code

Praxistipps

- Pattern Matching verbessert Code-Lesbarkeit
- Kombiniere Features für sauberen Code
- Halte dich für neue Features auf dem Laufenden:
  - Microsoft Docs
  - .NET Blog
  - GitHub Proposals