**Strukturen (struct)**

Unterschiede zwischen class und struct

* + **struct sind Werttypen**, class sind ReferenztypenDas heißt:Eine Variable eines Strukturtypes enthält die Daten der Struktur direkt**.**
  + Speicherort: **Stack** (struct) vs. **Heap** (class)

Wichtigsten Unterschiede zwischen class und struct

|  |  |  |
| --- | --- | --- |
| Eigenschaft | class (Klasse) | struct (Struktur) |
| Typ | Referenztyp | Werttyp |
| Speicherort | Heap | Stack (wenn in einer Methode) |
| Zuweisung | Referenz (Zeiger auf dasselbe Objekt) | Kopie des Werts |
| Vererbung | Unterstützt | Nicht unterstützt |
| ~~Standard-Konstruktor~~ | ~~Ja (parameterloser Konstruktor möglich)~~ | ~~Nein (kein parameterloser Konstruktor erlaubt)~~ |
| null-Wert möglich? | Ja | Nein (außer mit Nullable<T>) |
| Performance | Etwas langsamer (Heap-Allocation, Garbage Collector) | Schneller für kleine Objekte |
| Veränderbarkeit | Meist veränderlich | Sollte eher unveränderlich sein (readonly struct) |
| Typische Anwendung | Komplexe, referenzierte Objekte | Kleine, unveränderliche Datenstrukturen |

**Codebeispiel für class (Referenztyp)**

public struct Person

{

public string Name { get; set; }

public int Alter { get; set; }

}}

Person p1 = new Person();

Person p2 = p1; // p2 zeigt auf dasselbe Objekt wie p1

p2.Name = "Jürgen";

Console.WriteLine(p1.Name); // " Jürgen " (Änderung wirkt sich auf beide aus)

**Codebeispiel für struct (Werttyp)**

public struct Baum

{

public string Name { get; set; }

public int AnzahlBlaetter { get; set; }

}

Baum b1 = new Baum();

b1.AnzahlBlaetter = 50;

Baum b2 = b1; // Kopie der Werte, nicht die gleiche Referenz

b2.AnzahlBlaetter = 100;

Console.WriteLine(b1.AnzahlBlaetter); // 50 (b1 bleibt unverändert)

Console.WriteLine(b2.AnzahlBlaetter); // 100

**Wann class und wann struct?**

**Verwenden class, wenn:**

* Das Objekt **groß** und komplex ist
* Es **verändert** werden soll
* Es **Vererbung** benötigt
* Es **oft in Methoden übergeben** wird (keine unnötigen Kopien)

**Wann sollte ich struct verwenden?**

struct ist für **Werttypen (Value Types)** gedacht, die **klein, einfach und unveränderlich** (oder selten modifiziert) sind. Sie werden auf dem **Stack** gespeichert und nicht auf dem Heap.

**Kleine, kompakte Datentypen**

* Datentypen mit wenigen Feldern (z. B. Point, Color, DateTime).
* Strukturen, die nur wenige Bytes groß sind.

**Unveränderliche (immutable) Daten**

* struct sollte nach Möglichkeit **unveränderlich (immutable)** sein, d. h. **nur readonly-Felder** und keine **set-Methoden** haben.
* Beispiele: DateTime, TimeSpan, Guid.

**Mathematische Strukturen**

* Vector3D, Matrix, Complex (komplexe Zahlen).
* Werttypen, die numerische Berechnungen durchführen.

**Leistungsoptimierung bei vielen Instanzen**

* Wenn viele Objekte erstellt werden und die Performance wichtig ist, kann struct Speicher sparen.
* Beispiel: Ein struct mit 3 double-Feldern (24 Bytes) spart Platz gegenüber einer class, da keine **Heap-Header-Daten** erforderlich sind.

**Vermeidung von Garbage Collection**

* struct-Variablen leben auf dem Stack, wodurch weniger Arbeit für den Garbage Collector entsteht.
* Beispiel: Temporäre Daten wie Rectangle, Point oder Size in einem Grafikprogramm.

**Zur Errinerung:** int, double, bool usw. sind intern auch struct!

**Einschränkungen & Besonderheiten von struct**

**Keine Vererbung (struct kann nicht von einer anderen struct oder class erben)**

* struct kann keine andere struct oder class erben
* Aber: struct kann **Interfaces** implementieren

**Erlaubt:**

struct Punkt : IBeweglich

{

public int X, Y;

public Punkt(int x, int y) { X = x; Y = y; }

public void Bewegen(int x, int y)

{

X += x;

Y += y;

}

}}

**Kein parameterloser Konstruktor**

* In class können wir eigene Standardkonstruktoren definieren
* In struct gibt es **immer einen Standardkonstruktor**, der alle Felder auf Standardwerte setzt (0, false, null für Nullable<T>)
* ~~Benutzerdefinierte parameterlose Konstruktoren sind~~ **~~nicht erlaubt~~**

**Erlaubt (mit Parameter):**

struct Kreis

{

public double Radius;

public Kreis(double radius)

{

Radius = radius;

}

}

**struct ist standardmäßig sealed**

* class kann mit sealed gesperrt werden, sodass sie nicht vererbt werden kann
* struct ist **immer** sealed, da sie sowieso keine Vererbung unterstützt

**Kopierverhalten (Werttyp vs. Referenztyp)**

* struct wird **immer kopiert**, wenn es einer anderen Variablen zugewiesen wird
* class hingegen überträgt nur die **Referenz**

**Beispiel:**

struct Punkt

{

public int X, Y;

public Punkt(int x, int y) { X = x; Y = y; }

}

Punkt p1 = new Punkt(5, 10);

Punkt p2 = p1; // Kopie von p1

p2.X = 50;

Console.WriteLine(p1.X); // 5 (unverändert)

Console.WriteLine(p2.X); // 50 (geändert)

**readonly struct und readonly-Felder**

* **readonly struct** sorgt dafür, dass alle Felder unveränderlich sind
* readonly-Felder in struct schützen einzelne Eigenschaften vor Änderungen

**Beispiel:**

readonly struct Vektor

{

public readonly int X, Y;

public Vektor(int x, int y)

{

X = x;

Y = y;

}

}

**struct kann keine Destruktoren (~Konstruktor) haben**

* class kann einen Destruktor (~ClassName()) haben, der beim Löschen ausgeführt wird
* struct kann **keinen** **Destruktor** haben, weil struct nicht über den Garbage Collector verwaltet wird

**struct kann nicht null sein (außer mit Nullable<T>)**

* class kann null sein
* struct kann nicht null sein, aber man kann Nullable<T> oder T? verwenden

**Beispiel:**

Punkt? p = null; // Nullable struct

if (p.HasValue)

{

Console.WriteLine(p.Value.X);

}

**struct sollte klein sein**

* Da struct **immer kopiert** wird, sollte es **nicht zu groß** sein
* Faustregel: **Nicht mehr als 16 Bytes** (z. B. max. 4 int-Felder)
* Für große Objekte besser class verwenden

**Zusammenfassung:**

|  |  |
| --- | --- |
| Einschränkung | Beschreibung |
| Keine Vererbung | struct kann nicht von class oder struct erben (nur Interfaces). |
| Kein parameterloser Konstruktor | struct hat immer einen impliziten Standardkonstruktor. |
| Immer sealed | struct kann nicht von anderen struct geerbt werden. |
| Kopierverhalten | struct wird kopiert, class wird als Referenz übergeben. |
| Keine Destruktoren | struct kann keinen ~Destruktor() haben. |
| Nicht null | struct kann nicht null sein (außer Nullable<T>). |
| Performance | struct sollte klein sein (nicht mehr als 16 Bytes). |

**Besonderheit struct by ref**

Ein ref struct ist eine besondere Art von struct, die **ausschließlich auf dem Stapelspeicher (Stack)** gespeichert wird und **nicht im Heap** landen kann.

ref struct ist nützlich für **hochperformante, speicherkritische Operationen**, da es:

1. **Garbage Collection (GC) vermeidet**, weil es nicht auf dem Heap liegt.
2. **Direkt mit Stapelspeicher-Daten (Span<T>, stackalloc) arbeiten kann**, was ideal für **schnelle, temporäre Berechnungen** ist.
3. **Verhindert, dass es versehentlich auf den Heap gelangt**, wodurch Speicherlecks reduziert werden.

**Wann ist ref struct sinnvoll?**

1. **Mit Span<T> arbeiten**  
   Span<T> ist ein hochperformanter Datentyp für Speicherbereiche. Span<T> ist selbst ein ref struct und kann daher nur mit anderen ref struct-Typen verwendet werden.
2. **Für Stack-Only-Datenstrukturen**  
   Falls eine Datenstruktur niemals auf dem Heap gespeichert werden soll (z. B. um hohe Leistung sicherzustellen), kann ref struct verwendet werden.
3. **Effiziente Verarbeitung großer Datenmengen ohne Heap-Allokationen**  
   Beispiel: Wenn viele kleine temporäre Objekte erstellt werden, ist ref struct sinnvoll, um unnötige Speicherallokationen im Heap zu vermeiden.

**Warum ref struct hier sinnvoll ist?**

* Der Speicher wird direkt im **Stack** reserviert (stackalloc), nicht im Heap.
* **Keine Garbage Collection** (höchste Performance).

**Mehr zu Strukturen (struct):**

<https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/struct>

**Übung**

Erstelle eine struct für einen Zeitstempel (Stunde, Minute, Sekunde) und gebe die Werte in der ToString()-Methode aus.