Tupel

Ingo Köster

Diplom Informatiker (FH)

Tupel

- Gelegentlich kann es hilfreich sein, Datenelemente zu kombinieren

- Beispiel: Informationen zu Ländern sollen zusammengefasst werden
  - Name des Landes (string)
  - Hauptstadt (string)
  - Pro-Kopf-Bruttoinlandsprodukt (decimal)

- Für diese Daten könnte eine Klasse oder eine Struktur deklariert werden

Tupel

- Seit C# 7.0
- Ähnlich zu anonymen Typen

- Erlauben das Kombinieren mehrerer Variablen gleicher und verschiedener Typen in einer einzelnen Anweisung
  - Max. 8 Elemente

- Erlaubt es zudem mehrere Werte aus einer Methode zurückzugeben

Tupel erstellen

- Wichtiges Syntax Element für Tupel sind runde Klammern
- Deklaration eines Tupels:
  - `var tim = ("Tim", 23);`
- Fasst einen String und einen Integer zu einem Tupel zusammen

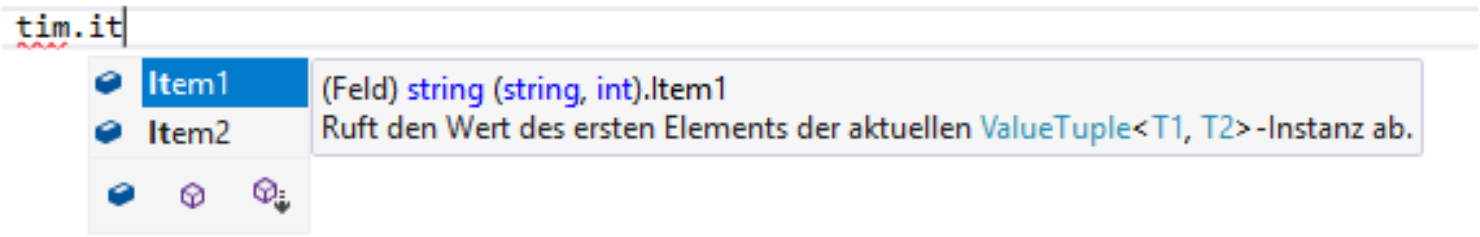
- Das Tupel kann z.B. auf der Konsole ausgegeben werden
  - `Console.WriteLine(tim);`

- Die Tupelvariable ist ein Tupeltyp mit den entsprechenden Typen

```
(Tim, 23)
System.ValueTuple`2[System.String,System.Int32]
```

Zugriff auf die Elemente des Tupels

- `var tim = ("Tim", 23);`
- Auf die Elemente des Tupels kann einzeln zugegriffen werden
- Diese beginnen mit `Item` und werden durchnummeriert
  - `Item1, Item2, Item3, usw.`



Benannte Tupel

› Alternativ können den Elementen des Tupels auch Namen zugewiesen werden

› Beispiel

```
var tim = (Name: "Tim", Alter: 23);
Console.WriteLine(tim.Name);
Console.WriteLine(tim.Alter);
```

› Die Elemente von Tupeln (benannt und unbenannt) sind nicht readonly und können daher verändert werden!

Typen der Elemente festlegen

- Bei dem folgenden Tupel werden `string` und `int` automatisch gewählt
  - `var tim = ("Tim", 23);`

- Der Typ des Tupels kann manuell festgelegt werden
  - `(string,short) tim = ("Tim", 23);`

Tupel erzeugen

› Tupel können auch aus Variablen erzeugt werden

```
› string text = "TextTextText";
› int zahl = 1;
› double irgendwas = 5.5;
› var einTupel = (text, zahl, irgendwas);
```

Tupel und Generische Typen

› Tupel und Generische Typen können kombiniert werden

› Beispiele
  › `List<(string, int)> personen = new List<(string, int)>();`
  › `IEnumerable<(float, double)> zahlen = new Stack<(float, double)>();`
  › `Task<(int, DateTime)> tasks  = …;`

Tupel als Rückgabewerte

› Da Typen von Tupeln festgelegt werden können, können Tupel als Rückgabewerte von Funktionen verwendet werden

```
public (string, int) GetPersonenDaten()
{
    return ("Tim", 23);
}
```

› Erlaubt die Rückgabe von mehr als einem Wert aus einer Methode

Tupel als Rückgabewerte

› Alternativ können den Eigenschaften des Tupel zur Rückgabe Namen vergeben werden

```
public (string Vorname, string Nachname, int Alter)
GetPersonenDaten()
{
  return ("Tim", "Müller", 23);
}
```

Tupel als Rückgabewerte

- Hat das Tupel Parameter mit Namen, kann auf diese nach der Rückgabe zugegriffen werden
  - `var result = GetPersonenDaten();`
  - `Console.WriteLine(result.Vorname);`
  - `Console.WriteLine(result.Nachname);`
  - `Console.WriteLine(result.Alter);`

- Mittels der sog. Dekonstruktoren kann dies noch weiter vereinfacht werden
  - Dazu später mehr

Tupel vergleichen

› Tupel können mit == und != verglichen werden

```
var person = (Name: "Tim", Alter: 23);
if (person == ("Tim", 23)) { … }
```

› Zwei Tupel vergleichen

```
var left = (a: 5, b: 10);
var right = (a: 5, b: 10);
Console.WriteLine(left == right);
```

Unboxing mit Tupeln

```
var tim = ("Tim", 21);

object obj = tim;

// Wichtig sind die „doppelten“ runden Klammern!
var einTupel = ((string, int))obj;

Console.WriteLine(einTupel.Item1);
Console.WriteLine(einTupel.Item2);
```

Tupel mit Linq

› Anstelle der Erzeugung von anonymen Typen mittels `new {…}` können in Linq Abfragen oft auch Tupel erzeugt werden

› Dazu bietet sich die Verwendung von `ValueTuple.Create` oder `ValueTuple<T>.Create` an

Anonyme Typen vs Tupel

› Anonymer Typ

```
› var result = pers
.Where(person => person.Alter > 30)
.Select(person => new { person.Nachname, person.Alter });
```

› Tupel

```
› var result = pers
.Where(person => person.Alter > 30)
.Select(person => ValueTuple.Create(person.Nachname,
person.Alter));
```

Dekonstruktoren

Dekonstruktoren

- Neu ab C# 7.0
- De**kon**struktoren sind keine Destruktoren!

- Ein Dekonstruktor stellt gewissenmaßen ein Gegenstück zum Konstruktor dar
- Ein Konstruktor kombiniert die Übergabeparameter und kapselt diese in einer Klasse
- Ein Dekonstruktor liefert die gekapselten Parameter einzeln wieder zurück

Klasse mit Konstruktor

```
class Adresse {
 public string Strasse { get; set; }
 public string Hausnummer { get; set; }
 public string Plz { get; set; }
 public string Ort { get; set; }

public Adresse(string strasse, string hausnummer, string plz, string ort) {
   this.Strasse = strasse;
   this.Hausnummer = hausnummer;
   this.Plz = plz;
   this.Ort = ort;
  }
}
```

Konstruktor und dekonstruieren

- Durch einen Aufruf des Konstruktors werden die Daten kombiniert
  - `Adresse lennershof  = new Adresse("Lennershofstraße", "160", "44801", "Bochum");`

- Soll ein Objekt zurück in seine Bestandteile zerleget werden (dekonstruiert), war vor C# 7 mehr als nur ein Aufruf notwendig
  - `string strasse = lennershof.Strasse;`
  - `string nummer = lennershof.Hausnummer;`
  - `string plz = lennershof.Plz;`
  - `string ort = lennershof.Ort;`

Dekonstruktor

- Seit C# 7.0 kann jeder Klasse ein Dekonstruktor hinzugefügt werden
  - Durch Überladung auch mehrere

- Diese spezielle Methode hat immer den Namen **`Deconstruct`** und einen oder mehrere out Parameter

```
public void Deconstruct(out string strasse, out string hausnummer, out string plz, out string ort)
{
    strasse = Strasse;
    hausnummer = Hausnummer;
    plz = Plz;
    ort = Ort;
}
```

Dekonstruktor aufrufen

- Aufruf des Dekonstruktors mit runden Klammern
  - `(string street, string number, string zip, string city) = lennershof;`

- Entspricht
  - `string street, string number, string zip, string city;`
  - `lennershof.Deconstruct(out street, out number, out zip, out city);`

Dekonstruktor und Typinferenz

› Dekonstruktor Aufrufe können `var` verwenden
  › `(var street, var number, var zip, var city) = lennershof;`

› Dekonstruktor Aufrufe können noch weiter vereinfacht werden
  › `var(street, number, zip, city) = lennershof;`

Dekonstruktor als Erweiterungsmethode

› Ein Dekonstruktor kann als eine Erweiterungsmethode angelegt werden, um fremde Typen dekonstruieren zu können

```
static class DateTimeHelper {
 public static void Deconstruct(this DateTime dateTime, out int
tag, out int monat, out int jahr) {
  tag = dateTime.Day;
  monat = dateTime.Month;
  jahr = dateTime.Year;
 }
}
```

Tupel und Dekonstruktoren

› Tupel verwenden Dekonstruktoren implizit

› Statt ein Tupel über seine Elemente einzeln zu zerlegen…

```
var tim = ("Tim", 23);
string name = tim.Item1;
int alter = tim.Item2;
```

› … Dekonstruktor verwenden

```
var tim = ("Tim", 23);
(string name, int alter) = tim;
```

Tupel und Dekonstruktoren

› Tupel können auch in bereits existierende Variablen dekonstruiert werden

```
var tim = ("Tim", 23);
string name;
int alter;
(name, alter) = tim;
```