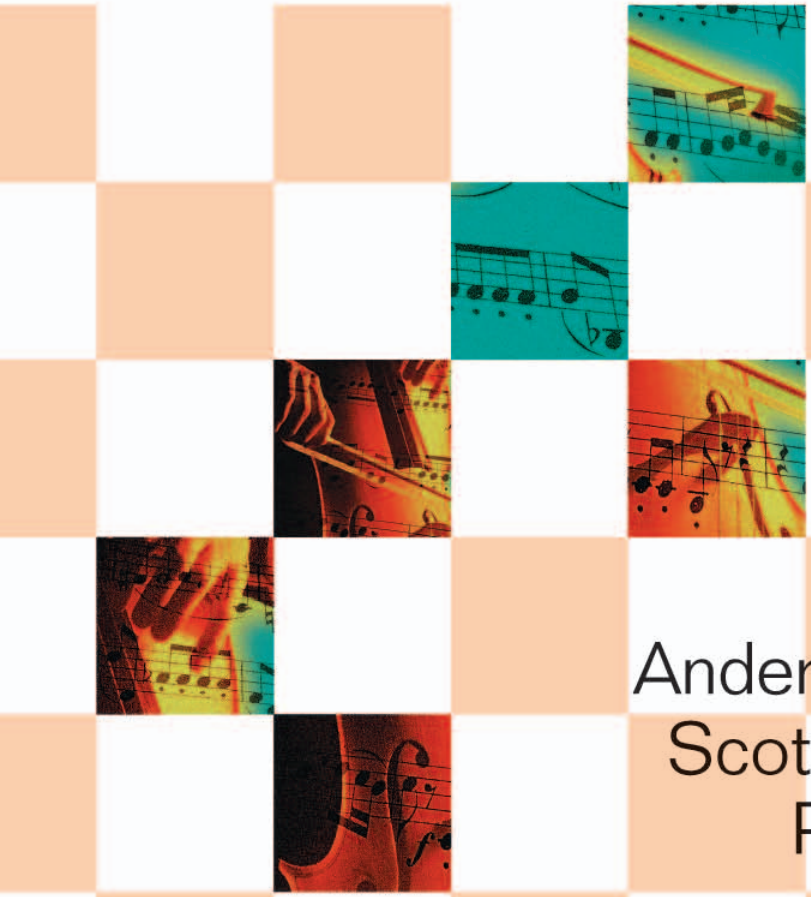
PEARSON Education

ADDISON-WESLEY

Die C#-Programmiersprache

Die komplette Referenz



Microsoft® .net™ Development Series

Anders Hejlsberg
Scott Wiltamuth
Peter Golde

ADDISON-WESLEY

21 Anonyme Methoden

21.1 Ausdrücke mit anonymen Methoden

Ein *anonymer-Methodenausdruck* definiert eine *anonyme Methode* und ergibt einen Wert, der auf diese Methode verweist.

primärer-Nicht-Array-Ausdruck:

. . .
anonymer-Methodenausdruck

anonymer-Methodenausdruck:

`delegate` *anonyme-Methodensignatur*$_{opt}$ *Block*

anonyme-Methodensignatur:

`(` *Parameterliste-der-anonymen-Methode*$_{opt}$ `)`

Parameterliste-der-anonymen-Methode:

Parameter-der-anonymen-Methode
Parameterliste-der-anonymen-Methode `,` *Parameter-der-anonymen-Methode*

Parameter-der-anonymen-Methode:

Parametermodifizierer$_{opt}$ *Typ Bezeichner*

Ein *anonymer-Methodenausdruck* wird als Wert mit besonderen Konvertierungsregeln klassifiziert (*§ 21.3*). Dieser Wert weist keinen Typ auf, kann aber implizit in einen kompatiblen Delegate-Typ konvertiert werden.

Der *anonyme-Methodenausdruck* definiert einen neuen Deklarationsraum für Parameter, lokale Variablen und Konstanten sowie für Bezeichner.

21.2 Signaturen anonymer Methoden

Die optionale *anonyme-Methodensignatur* definiert die Namen und Typen der formalen Parameter für die anonyme Methode. Der Bereich dieser Parameter ist der *Block*. Wenn der Name eines Parameters der anonymen Methode mit dem Namen einer lokalen Variablen, einer lokalen Konstanten oder eines Parameters übereinstimmt, dessen Gültigkeitsbereich den *anonymen-Methodenausdruck* einschließt, tritt ein Fehler bei der Kompilierung auf.

Wenn ein *anonymer-Methodenausdruck* eine *anonyme-Methodensignatur* aufweist, ist die Menge der kompatiblen Delegate-Typen auf diejenigen beschränkt, die über dieselben Parametertypen und Modifizierer in derselben Reihenfolge verfügen (*§ 21.3*). Im Gegensatz zur Konvertierung von Methodengruppen (*§ 21.9*) wird die Kontravarianz anonymer Methodenparametertypen nicht unterstützt. Wenn ein *anonymer-Methodenausdruck* keine *anonyme-Methodensignatur* aufweist, ist die Menge der kompatiblen Delegate-Typen auf diejenigen beschränkt, die keine `out`-Parameter haben.

Beachten Sie, dass *anonyme-Methodensignaturen* weder Attribute noch Parameter-Arrays enthalten dürfen. Dennoch kann eine *anonyme-Methodensignatur* mit einem Delegate-Typ kompatibel sein, dessen Parameterliste ein Parameter-Array einschließt.

21.3 Konvertierung anonymer Methoden

Ein *anonymer-Methodenausdruck* wird als Wert ohne Typ klassifiziert und kann in *Delegate-Erstellungsausdrücken* (*§ 21.10*) eingesetzt werden. Alle anderen zulässigen Verwendungsmöglichkeiten hängen von den im Folgenden definierten impliziten Konvertierungen ab.

Für einen *anonymen-Methodenausdruck* gibt es eine implizite Konvertierung (*§ 6.1*) in jeden *kompatiblen* Delegate-Typ. Wenn `D` ein Delegate-Typ ist und `A` ein *anonymer-Methodenausdruck*, dann ist `D` mit `A` dann und nur dann kompatibel, wenn die folgenden beiden Bedingungen zutreffen:

- Zunächst muss der Parametertyp von `D` kompatibel mit `A` sein:
  - Wenn `A` keine *anonyme-Methodensignatur* enthält, kann `D` null oder mehr Parameter eines beliebigen Typs aufweisen, sofern keiner von ihnen mit dem Modifizierer `out` versehen ist.

  - Wenn A eine *anonyme-Methodensignatur* enthält, muss D über dieselbe Anzahl von Parametern verfügen, die jeweils vom selben Typ sein müssen wie der entsprechende Parameter von A. Auch die Modifizierer ref und out müssen jeweils übereinstimmen. Ob der letzte Parameter von D ein *Parameter-Array* ist, hat keinen Einfluss auf die Kompatibilität von A und D.
- Zweitens muss der Rückgabetyp von D kompatibel mit A sein. Bei den folgenden Regeln wird vorausgesetzt, dass A keinen *Block* einer anderen anonymen Methode enthält:
  - Wenn D mit dem Rückgabetyp void deklariert ist, darf keine return-Anweisung in A einen Ausdruck spezifizieren.
  - Wenn D mit einem Rückgabetyp R deklariert ist, muss jede return-Anweisung in A einen Ausdruck angeben, der sich implizit in R konvertieren lässt (*§ 6.1*). Außerdem darf der Endpunkt des *Blocks* von A nicht erreichbar sein.

Für *anonyme-Methodenausdrücke* gibt es neben der impliziten Konvertierung in kompatible Delegate-Typen keine anderen Formen der Umwandlung, nicht einmal in den Typ object.

Das folgende Beispiel veranschaulicht diese Regeln:

```
delegate void D(int x);

D d1 = delegate { }; // Ok
D d2 = delegate() { };
                    // Fehler, Signaturen stimmen nicht überein
D d3 = delegate(long x) { };
                    // Fehler, Signaturen stimmen nicht überein
D d4 = delegate(int x) { };
                    // OK
D d5 = delegate(int x) { return; };
                    // OK
D d6 = delegate(int x) { return x; };
                    // Fehler, Rückgabetypen stimmen nicht überein

delegate void E(out int x);

E e1 = delegate { };
                    // Fehler, E hat keinen out-Parameter
E e2 = delegate(out int x) { x = 1; };
                    // OK
E e3 = delegate(ref int x) { x = 1; };
                    // Fehler, Signaturen stimmen nicht überein

delegate int P(params int[] a);

P p1 = delegate { };
                    // Fehler, Ende des Blocks erreichbar
P p2 = delegate { return; };
                    // Fehler, Rückgabetypen stimmen nicht überein
P p3 = delegate { return 1; };
                    // OK
```

```
P p4 = delegate { return "Hello"; };
                    // Fehler, Rückgabetypen stimmen nicht überein
P p5 = delegate(int[] a) {
                    // OK
  return a[0];
};

P p6 = delegate(params int[] a) {
                    // Fehler, Modifizierer params
  return a[0];
};

P p7 = delegate(int[] a) {
                    // Fehler, Rückgabetypen stimmen nicht überein
  if (a.Length > 0) return a[0];
  return "Hello";
};

delegate object Q(params int[] a);

Q q1 = delegate(int[] a) { // Ok
  if (a.Length > 0) return a[0];
  return "Hello";
};
```

21.4 Blöcke anonymer Methoden

Der *Block* eines *anonymen-Methodenausdrucks* unterliegt den folgenden Regeln:

- Wenn die anonyme Methode eine Signatur enthält, stehen die in ihr angegebenen Parameter innerhalb des *Blocks* zur Verfügung. Ist keine Signatur vorhanden, kann die Methode in einen Delegate-Typ mit Parametern konvertiert werden (*§ 21.3*), wobei ein Zugriff auf diese Parameter innerhalb des *Blocks* aber nicht möglich ist.
- Es tritt ein Fehler bei der Kompilierung auf, wenn der *Block* auf einen `ref`- oder `out`-Parameter zugreift. Ausgenommen davon sind nur die gegebenenfalls in der Signatur der nächsten einschließenden anonymen Methode angegebenen `ref`- und `out`-Parameter.
- Wenn der Typ von `this` ein Strukturtyp ist, verursacht es einen Fehler bei der Kompilierung, wenn der *Block* auf `this` zugreift. Dies gilt sowohl für den expliziten (wie in `this.x`) als auch den impliziten Zugriff (in der Form `x`, wobei `x` ein Instanzmember der Struktur ist). Diese Regel verhindert einfach einen Zugriff dieser Art und hat keinen Einfluss darauf, ob eine Membersuche ein Member der Struktur zum Ergebnis hat.
- Der *Block* hat Zugriff auf die äußeren Variablen (*§ 21. 5*) der anonymen Methode. Der Zugriff auf eine solche Variable verweist auf diejenige ihrer Instanzen, die während der Auswertung des *anonymen-Methodenausdrucks* aktiv ist (*§ 21.6*).

- Es tritt ein Fehler bei der Kompilierung auf, wenn der *Block* eine `goto`-, `break`- oder `continue`-Anweisung enthält, deren Ziel außerhalb des Blocks liegt oder im Block einer darin eingeschlossenen anonymen Methode.
- Eine `return`-Anweisung im *Block* gibt die Steuerung vom Aufruf der nächstgelegenen äußeren anonymen Methode zurück, nicht vom einschließenden Funktionsmember. Ein Ausdruck innerhalb der `return`-Anweisung muss mit dem Delegate-Typ kompatibel sein, in den der nächstgelegene einschließende *anonyme-Methodenausdruck* konvertiert werden kann (*§ 21.3*).

Es ist ausdrücklich nicht festgelegt, ob es eine Möglichkeit gibt, um den *Block* einer anonymen Methode auf andere Art und Weise auszuführen als durch den Aufruf und die Auswertung des *anonymen-Methodenausdrucks*. Insbesondere kann der Compiler eine anonyme Methode dadurch implementieren, dass er eine oder mehr benannte Methoden oder Typen erstellt. Die Namen dieser synthetischen Elemente müssen aus dem für den Compiler reservierten Raum stammen (d.h., die Namen müssen zwei aufeinander folgende Unterstriche enthalten).

21.5 Äußere Variablen

Alle lokalen Variablen, Werteparameter und Parameter-Arrays, deren Gültigkeitsbereich den *anonymen-Methodenausdruck* umfasst, werden als *äußere Variablen* dieses Ausdrucks bezeichnet. In einem Instanzfunktionsmember einer Klasse wird der Wert `this` als Werteparameter angesehen und ist damit eine äußere Variable für jeden in dem Funktionsmember enthaltenen *anonymen-Methodenausdruck*.

21.5.1 »Einfangen« (Capturing) äußerer Variablen

Wenn eine anonyme Methode auf eine äußere Variable verweist, spricht man davon, dass Letztere von der Methode *eingefangen* wird. Normalerweise ist die Gültigkeitsdauer einer lokalen Variablen auf die Ausführung des Blocks oder der Anweisung beschränkt, mit der sie verbunden ist (*§ 5.1.7*). Bei einer eingefangenen äußeren Variablen verlängert sie sich hingegen zumindest bis zu dem Zeitpunkt, an dem der Delegate, der auf die anonyme Methode verweist, für die Garbage Collection vorgesehen wird.

Im folgenden Beispiel wird die lokale Variable `x` von der anonymen Methode eingefangen, so dass sich ihre Gültigkeitsdauer mindestens so lange ausdehnt, bis der von `F` zurückgegebene Delegate für die Garbage Collection vorgesehen ist (was nicht vor dem endgültigen Ende des Programms geschieht):

```
using System;

delegate int D();

class Test
{
  static D F() {
    int x = 0;
    D result = delegate { return ++x; }
    return result;
  }

  static void Main() {
    D d = F();
    Console.WriteLine(d());
    Console.WriteLine(d());
    Console.WriteLine(d());
  }
}
```

Da jeder Aufruf der anonymen Methode Operationen an derselben Instanz von `x` vornimmt, ergibt sich folgende Ausgabe:

```
1
2
3
```

Wenn lokale Variablen oder lokale Parameter von einer anonymen Methode eingefangen werden, gelten sie nicht länger als feste Variablen (*§ 18.3*), sondern stattdessen als bewegliche. Unsafe Code, der die Adresse einer eingefangenen äußeren Variablen verwendet, muss daher zunächst die Anweisung `fix` einsetzen, um sie zu fixieren.

21.5.2 Instanziierung lokaler Variablen

Eine lokale Variable wird als *instanziiert* angesehen, wenn die Ausführung in ihren Gültigkeitsbereich eintritt. Beim Aufruf der folgenden Methode wird die lokale Variable `x` zum Beispiel dreimal instanziiert und initialisiert – jeweils einmal bei jedem Schleifendurchgang:

```
static void F() {
  for (int i = 0; i < 3; i++) {
    int x = i * 2 + 1;
    ...
  }
}
```

Die Deklaration von `x` aus der Schleife herauszunehmen führt dazu, dass nur eine Instanziierung von `x` auftritt:

```
static void F() {
  int x;
  for (int i = 0; i < 3; i++) {
```

```
    x = i * 2 + 1;
    ...
  }
}
```

Normalerweise gibt es keine Möglichkeit, um genau zu beobachten, wie oft eine lokale Variable instanziiert wird, denn da sich die Gültigkeitsdauern der einzelnen Instanziierungen nicht überlappen, können sie alle denselben Speicherort nutzen. Beim Einfangen einer lokalen Variablen durch eine anonyme Methode werden die Auswirkungen der Instanzbildung jedoch sichtbar. Das Beispiel

```
using System;

delegate void D();

class Test
{
  static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
      int x = i * 2 + 1;
      result[i] = delegate { Console.WriteLine(x); };
    }
    return result;
  }
  static void Main() {
    foreach (D d in F()) d();
  }
}
```

führt zu folgender Ausgabe:

```
1
3
5
```

Wird die Deklaration von `x` aus der Schleife herausgenommen, wie in folgendem Code:

```
static D[] F() {
  D[] result = new D[3];
  int x;
  for (int i = 0; i < 3; i++) {
    x = i * 2 + 1;
    result[i] = delegate { Console.WriteLine(x); };
  }
  return result;
}
```

so ändert sich auch die Ausgabe:

```
5
5
5
```

Beachten Sie, dass die drei Delegates in der gerade dargestellten Version von `F` aufgrund des Gleichheitsoperators identisch sind (*§ 21.7*). Darüber hinaus kann der Compiler die drei Instanzbildungen zu einer einzigen Delegate-Instanz optimieren, muss es aber nicht (*§ 21.6*).

Delegates anonymer Methoden können sich einige der eingefangenen Variablen teilen und gleichzeitig getrennte Instanzen von anderen verwenden. Wenn `F` zum Beispiel wie folgt geändert wird, fangen die drei Delegates dieselbe Instanz von `x`, aber unterschiedliche Instanzen von `y` ein:

```
static D[] F() {
  D[] result = new D[3];
  int x = 0;
  for (int i = 0; i < 3; i++) {
    int y = 0;
    result[i] = delegate
      { Console.WriteLine("{0} {1}", ++x, ++y); };
  }
  return result;
}
```

Dies führt zu folgender Ausgabe:

```
1 1
2 1
3 1
```

Verschiedene anonyme Methoden können dieselbe Instanz einer äußeren Variablen einfangen. Im folgenden Beispiel fangen die beiden anonymen Methoden dieselbe Instanz der lokalen Variablen `x` ein und können daher über diese Variable miteinander »kommunizieren«:

```
using System;

delegate void Setter(int value);
delegate int Getter();

class Test
{
  static void Main() {
    int x = 0;
    Setter s = delegate(int value) { x = value; };
    Getter g = delegate { return x; };
    s(5);
    Console.WriteLine(g());
    s(10);
    Console.WriteLine(g());
  }
}
```

Die Ausgabe lautet wie folgt:

```
5
10
```

21.6 Auswertung anonymer Methoden

Die Auswertung eines *anonymen-Methodenausdrucks* zur Laufzeit ruft eine Delegate-Instanz hervor, die auf die anonyme Methode und die (möglicherweise leere) Menge eingefangener, zum Zeitpunkt der Auswertung aktiver äußerer Variablen verweist. Wird ein solcher Delegate aufgerufen, so wird der Rumpf der anonymen Methode ausgeführt, und zwar unter Verwendung eben dieser eingefangenen, äußeren Variablen.

Die Aufrufliste eines aus einem *anonymen-Methodenausdruck* hervorgegangenen Delegates enthält nur einen einzigen Eintrag. Das genaue Zielobjekt und die Zielmethode des Delegates sind nicht angegeben. Vor allem steht nicht fest, ob das Zielobjekt `null`, der Wert `this` des einschließenden Funktionsmembers oder ein anderes Objekt ist.

Die Auswertung semantisch identischer *anonymer-Methodenausdrücke* mit derselben (möglicherweise leeren) Menge eingefangener äußerer Variableninstanzen ist zulässig, um dieselbe Delegate-Instanz zurückzugeben (wenn auch nicht erforderlich). »Semantisch identisch« bedeutet hier, dass die Ausführung der anonymen Methoden bei denselben Argumenten in jedem Fall dieselben Auswirkungen hervorruft. Diese Regel erlaubt es, Code wie den folgenden zu optimieren:

```
delegate double Function(double x);

class Test
{
  static double[] Apply(double[] a, Function f) {
    double[] result = new double[a.Length];
    for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
    return result;
  }

  static void F(double[] a, double[] b) {
    a = Apply(a, delegate(double x) { return Math.Sin(x); });
    b = Apply(b, delegate(double y) { return Math.Sin(y); });
    ...
  }
}
```

Da die beiden Delegates der anonymen Methoden dieselbe (leere) Menge von eingefangenen äußeren Variablen aufweisen und die Methoden selbst semantisch identisch sind, darf der Compiler die Delegates auf dieselbe Zielmethode verweisen lassen. Tatsächlich kann er bei beiden anonymen Methodenausdrücken dieselbe Delegate-Instanz zurückgeben.

21.7 Die Gleichheit von Delegate-Instanzen

Die folgenden Regeln bestimmen die Ergebnisse beim Einsatz des Gleichheitsoperators (*§ 7.9.8*) und der Methode `Object.Equals()` für die Delegate-Instanzen anonymer Methoden:

- Delegate-Instanzen, die der Auswertung semantisch identischer *anonymer-Methodenausdrücke* mit derselben (möglicherweise leeren) Menge eingefangener äußerer Variableninstanzen entstammen, dürfen gleich sein (müssen es aber nicht).
- Delegate-Instanzen, die der Auswertung semantisch unterschiedlicher *anonymer-Methodenausdrücke* entstammen oder verschiedene Mengen eingefangener äußerer Variableninstanzen aufweisen, sind niemals gleich.

21.8 Eindeutige Zuweisung

Der Zustand der eindeutigen Zuweisung eines Parameters für eine anonyme Methode entspricht dem eines Parameters für eine benannte Methode. Das bedeutet, dass Verweis- und Werteparametern vor dem Methodenaufruf ein Wert zugewiesen sein muss, Ausgabeparametern aber nicht. Darüber hinaus muss Ausgabeparametern ein Wert zugewiesen werden, bevor die anonyme Methode die Steuerung regulär zurückgibt (*§ 5.1.6*).

Der Zustand einer Variablen *v* (bezogen auf die Wertzuweisung) bei der Übergabe der Steuerung an den *Block* eines *anonymen-Methodenausdrucks* entspricht dem, den sie vor dem Ausdruck hatte. Der Status äußerer Variablen wird also vom Kontext des *anonymen-Methodenausdrucks* geerbt. Innerhalb des Blocks eines solchen Ausdrucks entwickelt sich der Status der Variablen wie in einem regulären Block (*§ 5.3.3*).

Der Zustand einer Variablen *v* ist nach einem *anonymen-Methodenausdruck* derselbe wie vorher.

Das folgende Beispiel verursacht einen Fehler bei der Kompilierung, da `max` bei der Deklaration der anonymen Methode nicht eindeutig zugewiesen ist:

```
delegate bool Filter(int i);

void F() {
  int max;
  // Fehler, max ist nicht eindeutig zugewiesen
  Filter f = delegate(int n) { return n < max; }
  max = 5;
  DoWork(f);
}
```

Auch das folgende Beispiel führt zu einem Fehler bei der Kompilierung, da die Zuweisung von `n` in der anonymen Methode keine Auswirkung auf den Status von `n` außerhalb der Methode hat:

```
delegate void D();

void F() {
  int n;
  D d = delegate { n = 1; };
  d();
  // Fehler, n ist nicht eindeutig zugewiesen
  Console.WriteLine(n);
}
```

21.9 Konvertierung von Methodengruppen

Für Methodengruppen (*§ 7.1*) gibt es ähnliche implizite Konvertierungen (*§ 6.1*) in einen kompatiblen Delegate-Typ wie die in *§ 21.3* beschriebenen impliziten Konvertierungen anonymer Methoden. Bei einem gegebenen Delegate-Typ `D` und einem Ausdruck `E`, der als eine Methodengruppe klassifiziert ist, gibt es eine implizite Konvertierung von `D` in `E`, wenn `E` mindestens eine Methode enthält, die in ihrer normalen Form (*§ 7.4.2.1*) auf eine Argumentliste mit Typen und Modifizierern anwendbar ist, die mit den Parametertypen und Modifizieren von `D` übereinstimmen.

Die Konvertierung einer Methodengruppe `E` in einen Delegate-Typ `D` zur Kompilierungszeit wird im Folgenden beschrieben. Beachten Sie, dass das Vorhandensein einer impliziten Konvertierung von `E` in `D` nicht gewährleistet, dass die Konvertierung zur Kompilierungszeit fehlerfrei abläuft.

- Eine einzelne Methode `M` wird entsprechend einem Methodenaufruf (*§ 20.9.7*) der Form `E(A)` mit den folgenden Änderungen ausgewählt:
  - Die Parametertypen und Modifizierer (`ref` oder `out`) von `D` werden als die Argumenttypen und Modifizierer der Argumentliste `A` verwendet.
  - Als mögliche Methodenkandidaten werden nur diejenigen Methoden berücksichtigt, die in ihrer normalen Form (*§ 7.4.2.1*) und nicht ausschließlich in ihrer erweiterten Form verwendet werden können.
  - Wenn der Algorithmus aus *§ 20.9.7* einen Fehler verursacht, tritt ein Fehler zur Kompilierungszeit auf. Andernfalls erzeugt der Algorithmus eine einzelne beste Methode `M()` mit derselben Anzahl Parameter wie `D`.
- Die ausgewählte Methode `M` muss (wie im Folgenden beschrieben) konsistent zum Delegate-Typ `D` sein. Andernfalls tritt ein Fehler bei der Kompilierung auf.

- Wenn die ausgewählte Methode `M` eine Instanzmethode ist, bestimmt der mit `E` verknüpfte Ausdruck das Zielobjekt des Delegates.
- Das Ergebnis der Konvertierung ist ein Wert vom Typ `D`, ein neu erstellter Delegate, der auf die ausgewählte Methode des Zielobjekts verweist.

Eine Methode `M` ist konsistent mit einem Delegate-Typ `D`, wenn alle folgenden Aussagen zutreffen:

- `D` und `M` haben dieselbe Anzahl Parameter, und jeder Parameter in `D` hat dieselben Modifizierer wie der entsprechende Parameter in `M`.
- Für jeden Werteparameter (ein Parameter ohne `ref`- oder `out`-Modifizierer) gibt es eine Identitätskonvertierung (*§ 6.1.1*) oder implizite Referenzkonvertierung (*§ 6.1.4*) vom Parametertyp in `D` in den entsprechenden Parametertyp in `M`.
- Für jeden `ref`- oder `out`-Parameter ist der Parametertyp in `D` derselbe wie der Parametertyp in `M`.
- Eine Identitätskonvertierung oder implizite Referenzkonvertierung erfolgt vom Rückgabetyp von `M` in den Rückgabetyp von `D`.

Die oben genannten Regeln zur Delegate-Konsistenz ersetzen die früheren Regeln zur Delegate-Kompatibilität, die in *§ 15.1* beschrieben werden, und sind zudem weniger restriktiv als diese.

Im folgenden Beispiel wird die Konvertierung von Methodengruppen gezeigt:

```
delegate string D1(object o);
delegate object D2(string s);
delegate string D3(int i);
class Test
{
static string F(object o) {...}
static void G() {
D1 d1 = F;  // Ok
D2 d2 = F;  // Ok
D3 d3 = F;  // Fehler
}
}
```

Die Zuweisung zu `d1` konvertiert die Methodengruppe `F` implizit in einen Wert vom Typ `D1`. Zuvor war es notwendig, `new D1(F)` zu schreiben, um den Delegate-Wert zu erzeugen. Dies ist zwar immer noch zulässig, aber nicht mehr zwingend erforderlich.

Die Zuweisung zu `d2` zeigt die Möglichkeiten einer Delegierung an eine Methode, die weniger stark abgeleitete (kontravariante) Parametertypen und einen stärker abgeleiteten (kovarianten) Rückgabetyp aufweist. Diese Methode ist sehr sicher und unproblematisch, da die als Parameter- und Rückgabewerte weitergegebenen Verweise einfach wie Verweise auf weniger stark abgeleitete Typen behandelt werden.

Die Zuweisung zu `d3` zeigt, dass sich die Parameter- und Rückgabetypen des Delegates und der Methode nur bei Verweistypen unterscheiden dürfen.

Wie bei allen anderen impliziten und expliziten Konvertierungen kann auch hier der Konvertierungsoperator verwendet werden, um explizit eine bestimmte Konvertierung einer Methodengruppe vorzunehmen. Das folgende Beispiel:

```
object obj = new EventHandler(myDialog.OkClick);
```

kann demnach auch so geschrieben werden:

```
object obj = (EventHandler)myDialog.OkClick;
```

Methodengruppen und anonyme Methodenausdrücke können die Überladungsauflösung beeinflussen, sind aber nicht am Schlussfolgern von Typen beteiligt. Weitere Einzelheiten dazu entnehmen Sie bitte *§ 20.6.4*.

21.10 Ausdrücke zur Delegate-Erstellung

Ausdrücke zur Delegate-Erstellung (*§ 7.5.10.3*) werden erweitert, damit das Argument ein als Methodengruppe klassifizierter Ausdruck, ein als anonyme Methode klassifizierter Ausdruck oder ein Wert eines Delegate-Typs sein kann.

Die Verarbeitung eines *Delegate-Erstellungsausdrucks* der Form `new D(E)`, wobei `D` ein *Delegate-Typ* und `E` ein *Ausdruck* ist, läuft zur Kompilierungszeit in den folgenden Schritten ab:

- Wenn `E` eine Methodengruppe ist, muss es eine Methodengruppenkonvertierung (*§ 21.9*) von `E` in `D` geben, und der Delegate-Erstellungsausdruck wird genauso wie diese Konvertierung verarbeitet.
- Wenn E eine anonyme Methode ist, muss es eine anonyme Methodenkonvertierung (*§ 21.3*) von `E` in `D` geben, und der Delegate-Erstellungsausdruck wird genauso wie diese Konvertierung verarbeitet.
- Wenn `E` ein Wert eines Delegate-Typs ist, muss die Methodensignatur von `E` konsistent (*§ 21.9*) zu `D` sein. Das Ergebnis ist ein Verweis auf einen neu erstellten Delegate vom Typ `D`, der auf dieselbe Aufrufliste wie `E` verweist. Wenn `E` nicht konsistent zu `D` ist, tritt ein Fehler bei der Kompilierung auf.

21.11 Implementierungsbeispiel

Dieser Abschnitt beschreibt die mögliche Implementierung einer anonymen Methode durch standardmäßige C#-Konstrukte. Sie beruht auf den Prinzipien, die der C#-Compiler von Microsoft verwendet, ist aber keineswegs zwingend erforderlich oder die einzig mögliche Implementierung.

Im weiteren Verlauf dieses Abschnitts finden Sie mehrere Codebeispiele, die anonyme Methoden mit unterschiedlichen Merkmalen aufweisen. Bei jedem von

ihnen ist eine Übersetzung in Code angegeben, der nur standardmäßige C#-Konstrukte verwendet. Der Bezeichner `D` soll dabei jeweils für den folgenden Delegate-Typ stehen:

```
public delegate void D();
```

Die einfachste Form einer anonymen Methode fängt keine äußeren Variablen ein:

```
class Test
{
  static void F() {
    D d = delegate { Console.WriteLine("test"); };
  }
}
```

Dies lässt sich in eine Delegate-Instanz übersetzen, die auf eine vom Compiler angelegte statische Methode mit dem Code der anonymen Methode verweist:

```
class Test
{
  static void F() {
    D d = new D(__Method1);
  }

  static void __Method1() {
    Console.WriteLine("test");
  }
}
```

Im folgenden Beispiel verweist die anonyme Methode auf Instanzmember von `this`:

```
class Test
{
  int x;

  void F() {
    D d = delegate { Console.WriteLine(x); };
  }
}
```

Dies lässt sich in eine vom Compiler hervorgerufene Instanzmethode übersetzen, die den Code der anonymen Methode enthält:

```
class Test
{
  int x;

  void F() {
    D d = new D(__Method1);
  }

  void __Method1() {
    Console.WriteLine(x);
  }
}
```

Im folgenden Beispiel fängt die anonyme Methode eine lokale Variable ein:

```
class Test
{
  void F() {
    int y = 123;
    D d = delegate { Console.WriteLine(y); };
  }
}
```

Die Gültigkeitsdauer der lokalen Variablen muss nun mindestens bis zum Ende der Gültigkeitsdauer des Delegates der anonymen Methode verlängert werden. Dies lässt sich dadurch erreichen, dass man die lokale Variable in das Feld einer vom Compiler erstellten Klasse befördert. Die Instanziierung der lokalen Variablen (*§ 21.5.2*) entspricht dann der Instanziierung der vom Compiler erstellten Klasse und der Zugriff auf die lokale Variable dem Zugriff auf ein Feld in der Instanz dieser Klasse. Darüber hinaus wird die anonyme Methode zu einer Instanz dieser Klasse:

```
class Test
{
  void F() {
    __Locals1 __locals1 = new __Locals1();
    __locals1.y = 123;
    D d = new D(__locals1.__Method1);
  }
  class __Locals1
  {
    public int y;
    public void __Method1() {
      Console.WriteLine(y);
    }
  }
}
```

Die folgende anonyme Methode schließlich fängt `this` ebenso wie zwei lokale Variablen mit unterschiedlicher Gültigkeitsdauer ein:

```
class Test
{
  int x;

  void F() {
    int y = 123;
    for (int i = 0; i < 10; i++) {
      int z = i * 2;
      D d = delegate { Console.WriteLine(x + y + z); };
    }
  }
}
```

Hierbei erstellt der Compiler für jeden Anweisungsblock, der lokale Variablen einfängt, eine Klasse, so dass die Gültigkeitsdauern der Variablen in den verschiedenen Blöcken voneinander unabhängig sind. Eine Instanz von `__Locals2`, der vom Compiler angelegten Klasse für den inneren Anweisungsblock, enthält die lokale Variable `z` und ein Feld, das auf eine Instanz von `__Locals1` verweist. Eine Instanz von `__Locals1` wiederum, der vom Compiler erstellten Klasse für den äußeren Anweisungsblock, enthält die lokale Variable `y` und ein Feld, das auf das `this` des einschließenden Funktionsmembers verweist. Mit diesen Datenstrukturen ist es möglich, alle eingefangenen äußeren Variablen über eine Instanz von `__Locals2` zu erreichen, so dass der Code der anonymen Methode als Instanzmethode dieser Klasse implementiert werden kann:

```
class Test
{
  void F() {
    __Locals1 __locals1 = new __Locals1();
    __locals1.__this = this;
    __locals1.y = 123;
    for (int i = 0; i < 10; i++) {
      __Locals2 __locals2 = new __Locals2();
      __locals2.__locals1 = __locals1;
      __locals2.z = i * 2;
      D d = new D(__locals2.__Method1);
    }
  }

  class __Locals1
  {
    public Test __this;
    public int y;
  }

  class __Locals2
  {
    public __Locals1 __locals1;
    public int z;
    public void __Method1() {

      Console.WriteLine(__locals1.__this.x + __locals1.y + z);
    }
  }
}
```