**Delegaten, Anonyme Methoden, Lambda, Func<> und Action<>**

**Delegate**

Ein Objekt von Typ Delegate ist ein **Methodenzeiger** d.h. es zeigt bzw. leitet auf eine oder mehrere **Methoden** weiter.

**Beispielv**

**1.** Einen Delegaten-Typ deklarieren

delegate string **StrDelegate**(string str1, string str2);

**delegate**: hier wichtiges Schlüsselwort

**string**: Rückgabetyp der aufzurufenden (an dem Delegaten gebundenen) Methode.

**StrDelegate**: Freiwählbarer Name des Delegaten

**(string str1, string str2):** Etwaige Übergabeparameter der aufzurufenden (an dem Delegaten gebundenen) Methode.

**2.** Ein Objekt von Typ StrDelegate **instanziieren** und damit gleich auf eine bestimmte Methode zeigen (hier: func1).

var strDelegate = new StrDelegate(func1);

**3.** Delegaten ausführen und dessen Wert in Variable result1 speichern. Bei Ausführung des Delegaten wird in Wirklichkeit, die an ihm gebundene Methode func1 durchgeführt.

var result1 = strDelegate("Apfel", "Birne");

**Hinweis:** Die Signatur der Methode (Rückgabetyp und Parameter), auf die das Delegate zeigt muss zwingend in Delegate-Definition abgebildet sein. Hier, in diesem Beispiel sind es: Rückgabetyp: string und zwei Übergabeparameter, ebenfalls vom Typ string.

**Übung**

Löse folgendes Beispiel mit einem geeigneten Delegaten

int a = 7;

int b = 3;

int resultAdd = Add(a, b);

int resultMultiply = Multiply(a, b);

Console.WriteLine("Ergebnis Addition: " + resultAdd);

Console.WriteLine("Ergebnis Multiplikation: " + resultMultiply);

static int Add(int x, int y)

{

return x + y;

}

static int Multiply(int x, int y)

{

return x \* y;

}

**Multicast-Delegates**

Mit Delegaten kann man auch auf mehrere Methoden gleichzeitig zeigen. Dies wird durch die sogenannte **Multicast-Delegates** ermöglicht. Ein Multicast-Delegate enthält eine Aufrufliste von Methoden und ruft diese in der Reihenfolge ihres Hinzufügens auf.

**Beispiel**

// Instanzieren und erstes Ziel hinzufügen

var multiDelegate = new MultiDelegate(schreibe);

// Zweites Ziel hinzufügen

multiDelegate += loescheA;

// Aufruf des Delegates

multiDelegate("Programmieren macht glücklich");

// So geht's entfernen:

multiDelegate -= schreibe;

**Anonyme Methoden**

Eine Vereinfachung obiger Vorgangsweise kann mittels Verwendung von s.g. anonymen Funktionen erreichen. „**Anonym**“, weil Methodenkörper der Funktion direkt an die Instanziierung des Delegaten-Objektes angefügt wird und die Funktionalität selbst keinen Namen hat.

var strDelegate = delegate (string str1, string str2)

{

return string.Concat(str1, str2); // Dies ist der Code aus der ehemaligen func1

};

**Wie wäre es aber noch kompakter mit Lambda?**

Bei Lambda-Ausdrücken handelt es sich eigentlich um nichts anderes, als um **funktional erweiterte anonyme Methoden**. Der offensichtlichste Unterschied zu anonymen Methoden zeigt sich in der Syntax. Auf die Parameterliste folgt ein Istgleichzeichen und ein Pfeil **=>** gefolgt von einer Anweisung bzw. einem Anweisungsblock. Lambda-Ausdrücke in C# sind eine **kompakte** und elegante Möglichkeit Logik, die man sonst in Methoden unterbringen würde, zu definieren. Sie ermöglichen es, kurze „Funktionen“ **direkt dort zu schreiben, wo sie gebraucht werden**, ohne eine separate Methode zu definieren zu müssen.

**Beispiel**

var strDelegate = (string str1, string str2) => string.Concat(str1, str2);

**Legende:**

**(string str1, string str2)** : Anweisung bzw. Anweisungsblock.

**=>** : wird oft als "geht über in" oder "wird zu" gelesen. Trennzeichen zwischen Parameterliste und Anweisung bzw. einem Anweisungsblock.

**string.Concat(str1, str2)** : Beispiel für beliebige Anweisung

**Func<> und Action<>**

**Anstatt** ein eigenes Delegate zu definieren, kann man stattdessen **Func<>** und **Action<>** verwenden.

**Func<>**

Func<string, string, string> strDelegate = (str1, str2) => string.Concat(str1, str2);

var result1 = strDelegate("Apfel", "Birne");

**Legende**

**Func** : hat immer einen Rückgabewert

**<string, string, string>** : rot: Typen der Übergabeparameter, grün: Rückgabetyp

**(str1, str2)** : Stelle, wo Parameter übergeben werden.

**=>** : wird oft als "geht über in" oder "wird zu" gelesen. Trennzeichen zwischen Parameterliste und Anweisung bzw. einem Anweisungsblock.

**string.Concat(str1, str2)** : Beispiel für beliebige Anweisung

**Action<>**

Action<string, string> strDelegate = (str1, str2) => Console.WriteLine(string.Concat(str1, str2));

**Legende**

**Action** : hat keinen Rückgabewert, es ist void

**<string, string>** : Typen der Übergabeparameter

**(str1, str2)** : Stelle, wo Parameter übergeben werden.

**=>** : wird oft als "geht über in" oder "wird zu" gelesen. Trennzeichen zwischen Parameterliste und Anweisung bzw. einem Anweisungsblock.

**Console.WriteLine(string.Concat(str1, str2))** : Anweisung wird ausgeführt ohne etwas zurückzuliefern.

**Übung**

Verwandle die folgende Methode in

* eine anonyme Methode
* und dann in einen Lambda-Ausdruck:

static int Subtrahiere(int x, int y) { return x - y; }

**Gemeinsame Übung**

Für eine Liste von Büchern erstellen wir **eine einzige universelle Filterungs-Methode**, die mit Bücher nach beliebigen Kriterien filtern kann und **Func<Buch, bool>** als Parameter akzeptiert.

**Interessantes**

Der Begriff **Lambda-Ausdruck** kommt aus der **Lambda-Kalkulation** (λ-Kalkül), die der Mathematiker **Alonzo Church** in den 1930er-Jahren entwickelt hat.

Die Idee war, eine sehr einfache formale Sprache zu schaffen, um **Funktionen und deren Anwendung** mathematisch zu beschreiben – unabhängig von Zahlen oder arithmetischen Operationen.

Im λ-Kalkül schreibt man eine anonyme Funktion so:

λx.x+1λx. x + 1

Das bedeutet: „Eine Funktion, die ein Argument xx nimmt und x+1x+1 zurückgibt.“  
In moderner Programmiersprache wäre das z. B. in C#:

x => x + 1

**„Lambda“?**

* Das griechische **λ** (Lambda) wurde von Church als Symbol gewählt, um **Funktionsabstraktion** zu markieren (also: „ich definiere hier eine Funktion“).
* Der Name hat sich bis heute gehalten – viele Programmiersprachen (C#, Java, Python, JavaScript, Haskell usw.) nennen ihre Kurzschreibweise für **anonyme Funktionen** deshalb „Lambda-Ausdrücke“.
* Ein „Lambda-Ausdruck“ ist also einfach die **praktische Umsetzung einer Funktion im Sinne des Lambda-Kalküls** – nur eben in moderner Programmiersyntax.

Es heißt **Lambda-Ausdruck**, weil die Idee aus dem **λ-Kalkül** stammt, wo das Symbol **λ** für die Definition anonymer Funktionen steht.