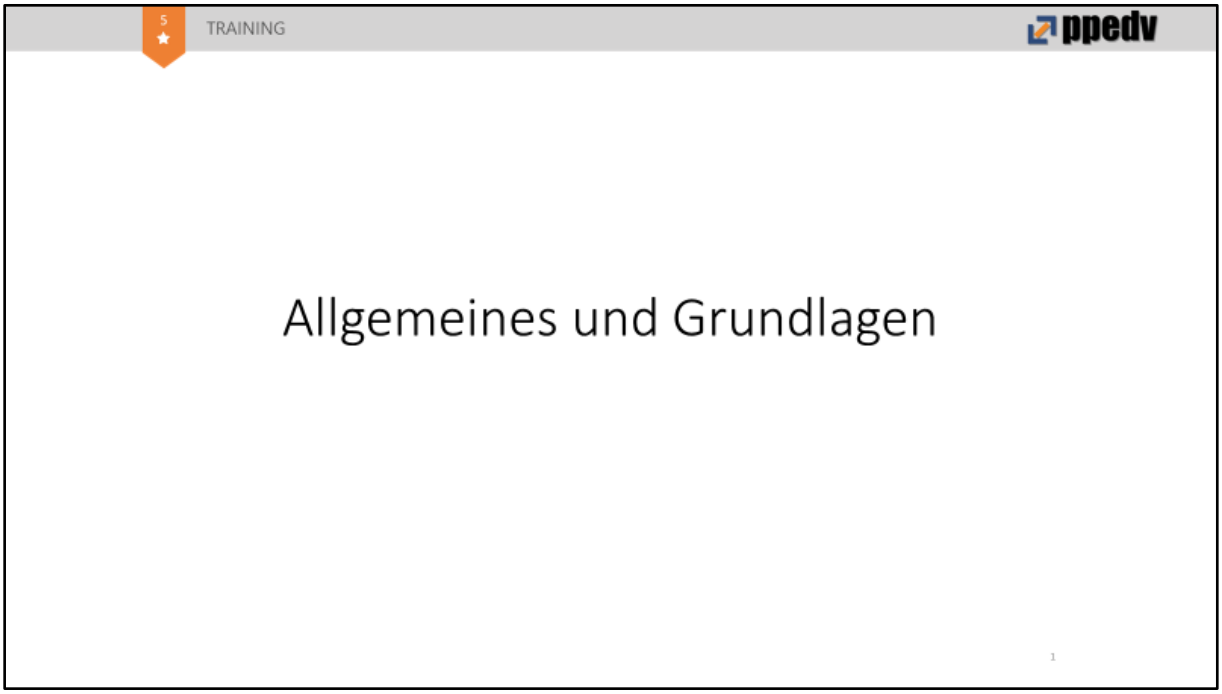
C# Grundlagen

[Untertitel des Dokuments]

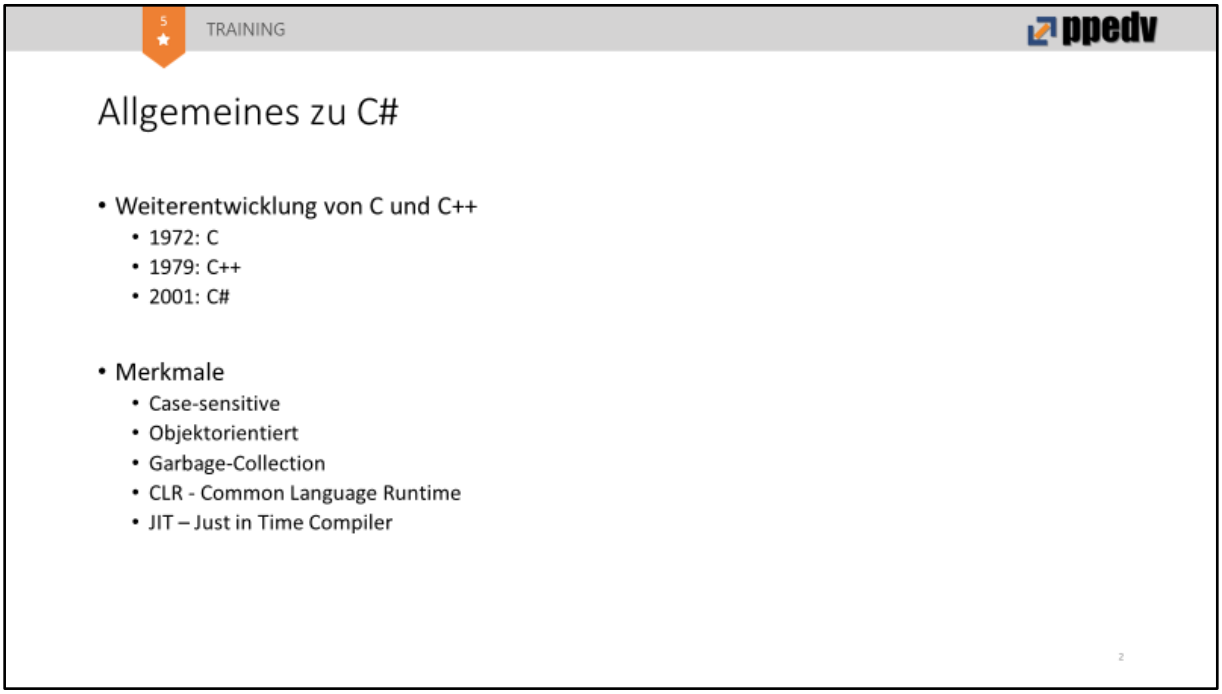
ppedv AG

Inhaltsverzeichnis

Allgemeines und Grundlagen



Allgemeines zu C#



C# wurde 2001 als die aktuellste Generation der C-Sprachen veröffentlicht und vereint Elemente der Vorgänger mit erfolgreichen Konzepten von objektorientierten Programmiersprachen wie z.B. Java. Visual C# ist fester Bestandteil des .NET-Frameworks.

Anders als viele andere Sprachen unterscheidet C#, wie seine direkten Vorgänger, bei sämtlichen Bezeichnern zwischen großen und kleinen Zeichen. Im Unterschied zu seinem direkten Vorgänger ist es im Normalfall nicht mehr notwendig sich um das Ressourcenmanagement zu kümmern, da die integrierte Garbage-Collection automatisch Objekte aus dem Speicher löscht, welche im Programm durch keine Variable mehr referenziert werden.

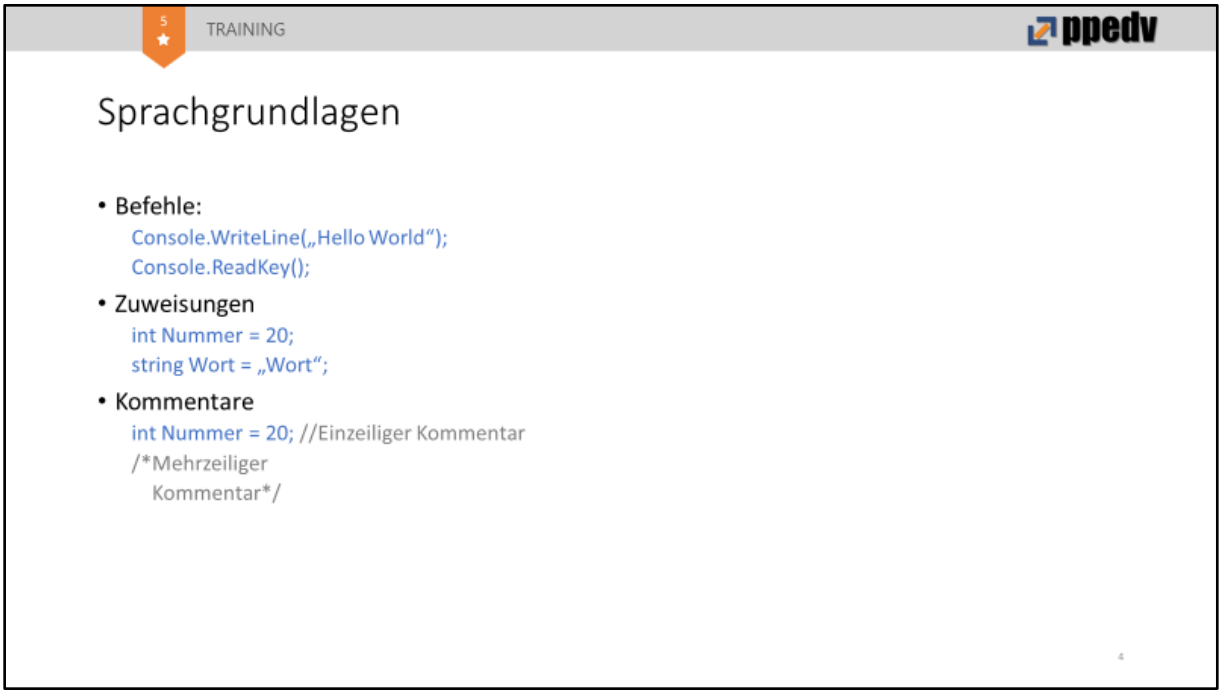
Als Teil der .NET-Umgebung, welche die Ausführbarkeit eines C#-Programmes auf allen mit dem .NET-Framework ausgestatteten Systemen ermöglicht, wird der Programmcode bei Compilierung in eine ‚Common Intermediate Language (CIL)' übersetzt. Erst zur Laufzeit wird das Programm durch den ‚Just-In-Time-Compiler (JIT)' der ‚Common Language Runtime (CLR)' des .NET-Frameworks in die für das jeweilige System verständliche Sprache übersetzt.

Allgemeines zu C# und .NET

Das .NET-Framework unterstützt neben C# auch andere Sprachen, wie Visual Basic oder C++. Aufgeteilt ist das Framework in mit verschiedene Teile, welche jeweils unterschiedliche Aufgaben auf unterschiedlichen Systemen erfüllen können.

Neben der am häufigsten benutzten Entwicklungsumgeben VisualStudio existieren weitere Umgebungen, welche sich auf bestimmte Aufgaben spezialisiert haben. Während sich zum Beispiel Xamarin-Studio für die Entwicklung von Xamarin-Programmen (also mobilen Apps) eignet, ist die Umgebung Unity speziell für die Entwicklung von Spielen mit 3D-Komponenten gedacht.

Sprachgrundlagen

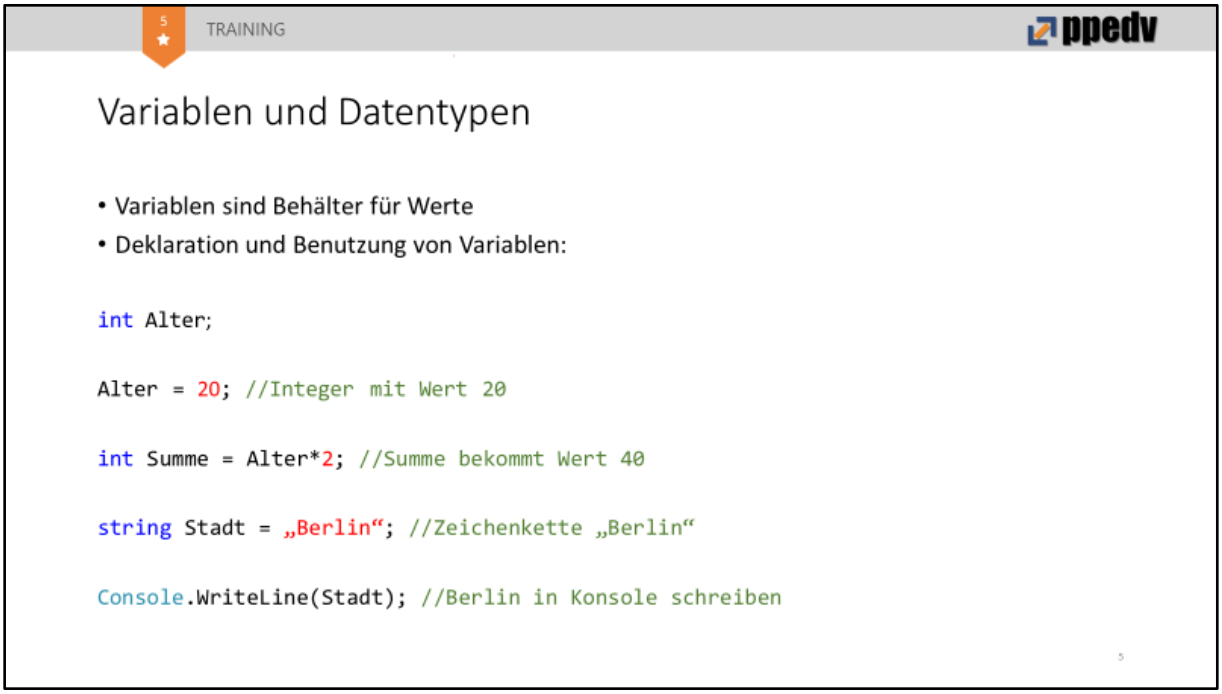


Jeder Befehl (zumeist Methodenaufrufe) in C# muss mit einem Semikolon (;) abgeschlossen werden, damit der Compiler den Befehl als abgeschlossen interpretiert. In den Fällen, in welchen einem Befehl ein durch geschweifte Klammern ({ }) begrenzter Körper folgt, ersetzen diese das Semikolon.

Zuweisungen von Werten oder Objekten zu Variablen werden in C# mit dem einfachen Ist-Gleich (=) durchgeführt. Dies ist nicht mit dem Vergleichsoperator zu verwechseln, welcher in C# mit dem doppelten Ist-Gleich (==) dargestellt werden.

Einzeilige Kommentare werden in C# mit einem doppelten Schrägstrich (//) eingeleitet. Mehrzeilige Kommentare werden durch Stern-Schrägstrich (/*) ein- und Schrägstrich-Stern (*/) ausgeleitet.

Variablen und Datentypen



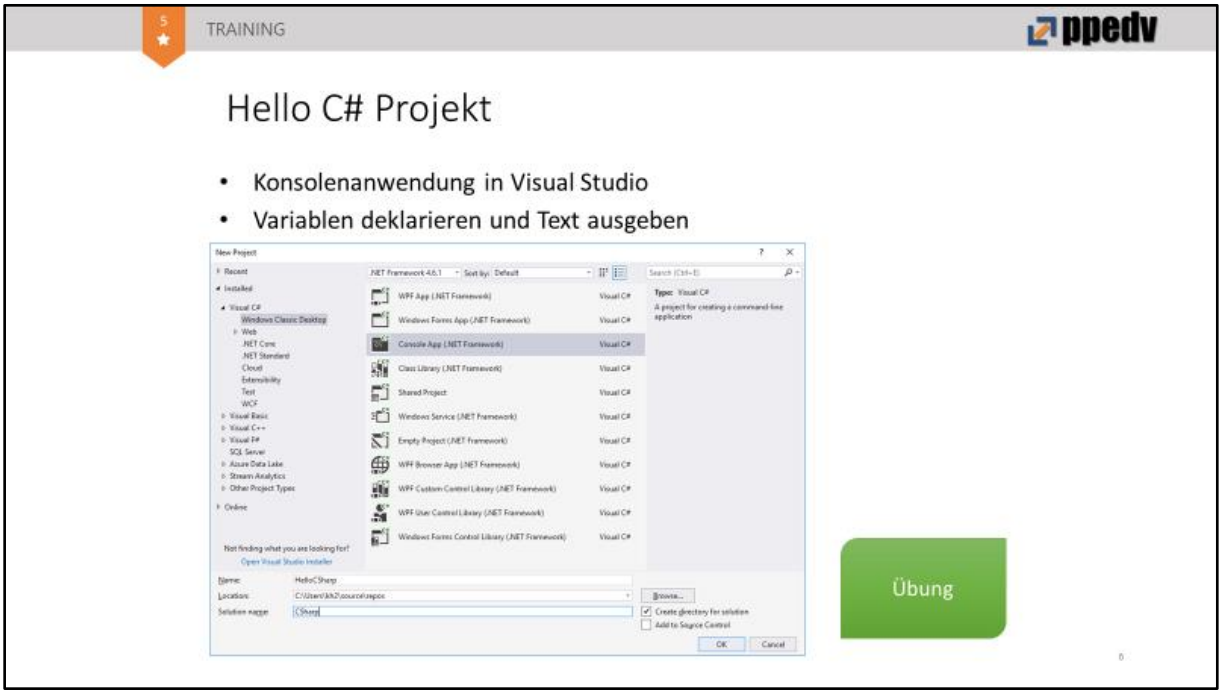
Es existiert ein eklatanter Unterschied zwischen Variablen und den Werten und Objekten, welche man diesen Variablen zuweist.

Die Variablen selbst sind ‚Behälter‘ welche jeweils eine ganz bestimmte Art von Wert/Objekt beinhalten können. Bei der Erstellung (Deklaration) der Variablen gibt man an, welchen Datentyp die jeweilige Variable beinhalten kann, indem man diesen vor den selbst gewählten Bezeichner schreibt.

Durch eine Zuweisung (=) ‚befüllt‘ man die Variable dann mit einem Wert/Objekt des jeweiligen Datentyps. Die erstmalige ‚Befüllung‘ heißt Initialisierung. Dabei kann bei der Zuweisung ein absoluter Wert bzw. ein neues Objekt in die Variable gefüllt werden oder unter Bezugnahme auf bereits existierende Variablen ein vorhandener Wert/Objekt manipuliert oder vollständig übernommen werden.

Über den Bezeichner der Variablen gereift man auf die Werte/Objekte zu, welche sich momentan in der Variablen befinden.

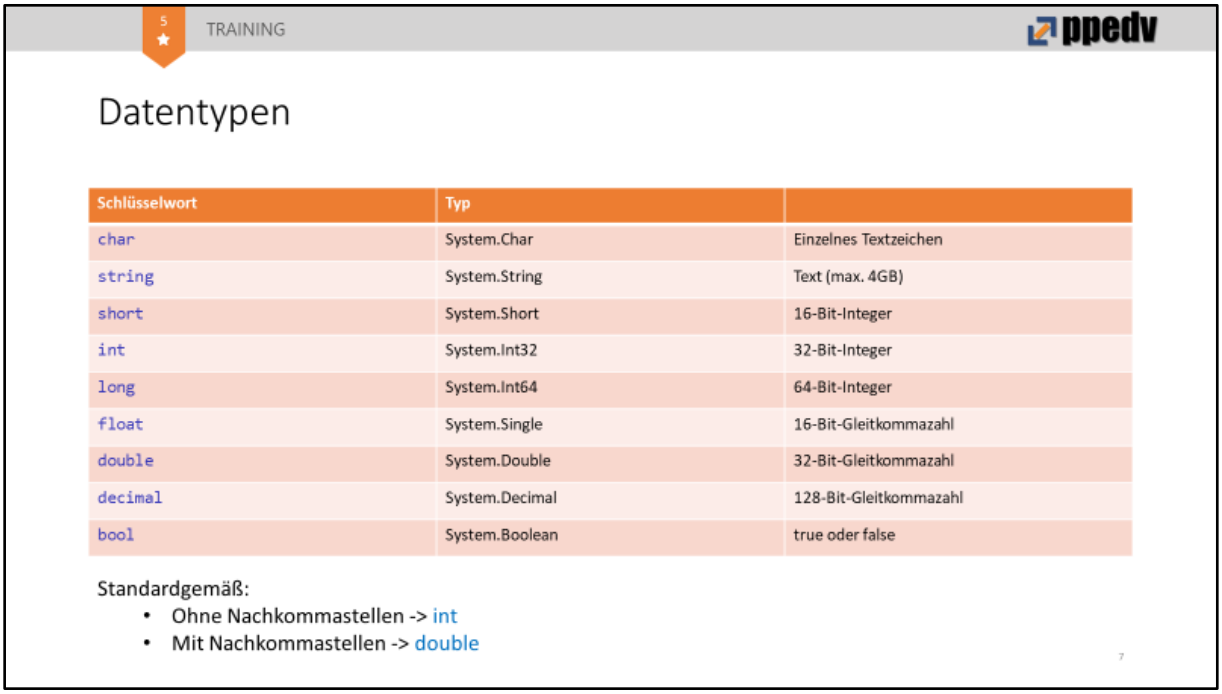
Hello C# Projekt



In VisualStudio wird jedes Projekt (welches im Normalfall einem Programm / einer Klassenbibliothek entspricht) eine Solution (Projektmappe) zugeordnet, welche generell beliebig viele Projekte beinhalten kann.

Um einer bestehenden Solution ein neues Projekt hinzuzufügen muss im Projekt-Explorer (rechts im Bild) per Rechtsklick auf die Solution über das Add-Menü ein neues Projekt beigefügt werden.

Datentypen



Jeder Wert und jedes Objekt besitzt einen Datentypen, durch welchen seine Struktur und seine Speicherung definiert wird. In der obigen Tabelle sind die Basisdatentypen aufgelistet, welche vom .NET-Framework zur Verfügung gestellt werden.

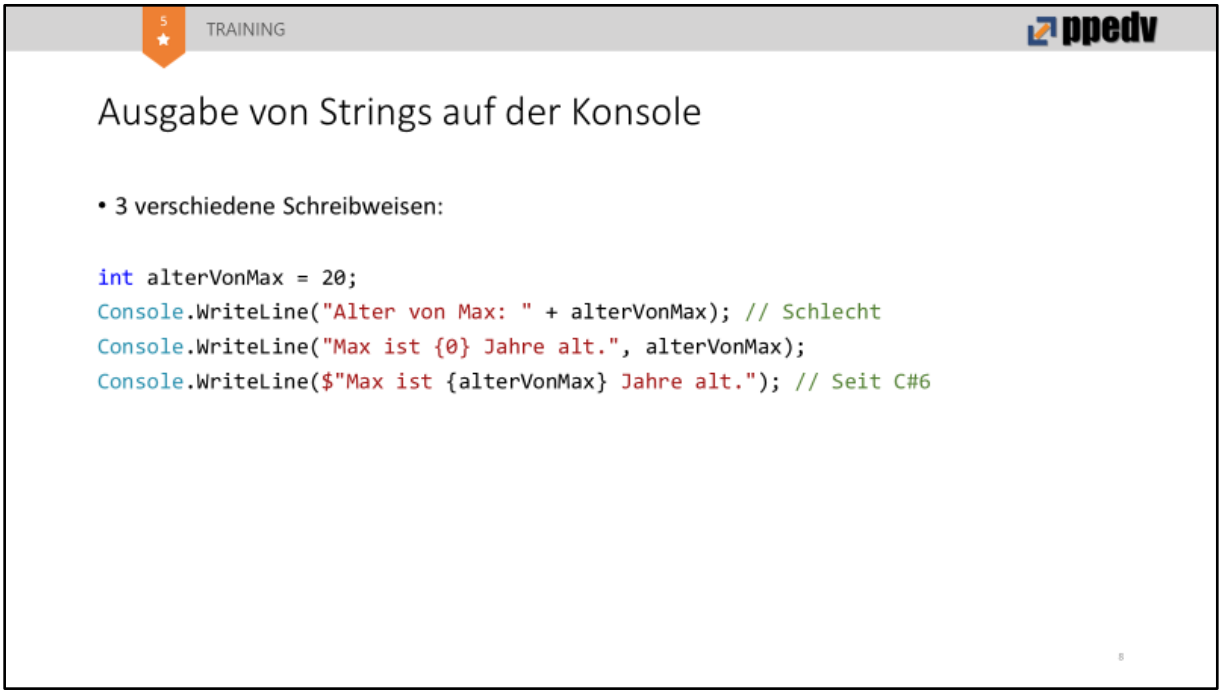
Dabei ist zu beachten, dass mit Ausnahme des ‚strings' alle oben beschriebenen Datentypen als sogenannte Wertetypen behandelt werden. Das heißt bei einer Neuzuweisung eines Werts an eine weitere Variable oder einer Übergabe an eine Funktion wird eine Kopie des Werts erstellt und übergeben. Eine Veränderung dieses neuen Werts hat keine Auswirkungen auf das ursprüngliche Objekt. Strings sowie alle komplexen Datentypen werden als Referenztypen behandelt, was bedeutet, dass bei einer Neuzuweisung eine Referenz auf das ursprüngliche Objekt übergeben wird. Eine Veränderung an der neuen Variable wirkt sich so auch auf die ursprüngliche Variable aus.

Jeder Datentyp (auch die oben gezeigten Basisdatentypen) erben von der Object-Klasse, welche als Grundgerüst für alle weiteren Objekte funkgiert.

Wird bei den numerischen Typen keine genauere Spezifizierung angegeben, werden alle Zahlen ohne Nachkommastellen als Integer (Int32) und alle Zahlen mit Nachkommastellen als Double interpretiert.

Für Ganzzahl-Typen gibt es auch eine Unsigned-Variante, d.h. eine Variante, die keine negativen Zahlen speichern kann: ushort (0 bis 65.535) uint (0 bis 4.294.967.295) ulong (0 bis 18.446.744.073.709.551.615)

Ausgabe von Strings auf der Konsole



Jeder String muss in C# von doppelten Anführungszeichen („“) eingerahmt sein, damit er als String interpretiert wird.

Um Strings in der Konsole auszugeben wird (u.a.) die WriteLine()-Funktion der Console-Klasse verwendet. Dabei können mehrere Strings auf verschiedene Art und Weise kombiniert werden.

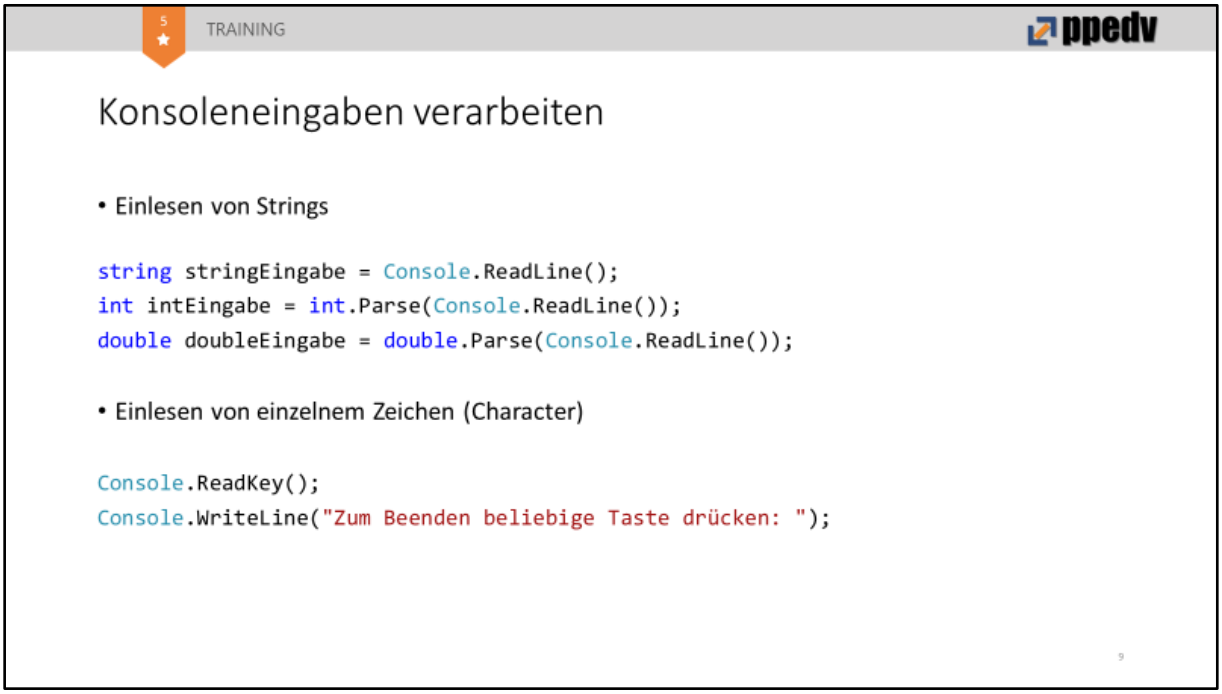
Die ‚traditionelle‘ Variante, bei welcher zwischen den Strings (in Form von festen Strings oder dynamischen Inhalten, sprich Variablen) der Plus-Operator (+) eingefügt wird, wurde von älteren Sprachen übernommen.

In der Index-Form werden in den Strings an Stellen, an welchen ein dynamischer Inhalt erscheinen soll, ein in geschweifte Klammern geschriebener Index gestellt.

Dem String werden die Variablen, durch Kommata getrennt, nachgestellt.

Seit C# Version 6 können in Strings denen ein Dollar-Zeichen ($) vorangestellt wird die Variablenbezeichner direkt in die geschweifte Klammern im String geschrieben werden.

Konsoleneingaben verarbeiten

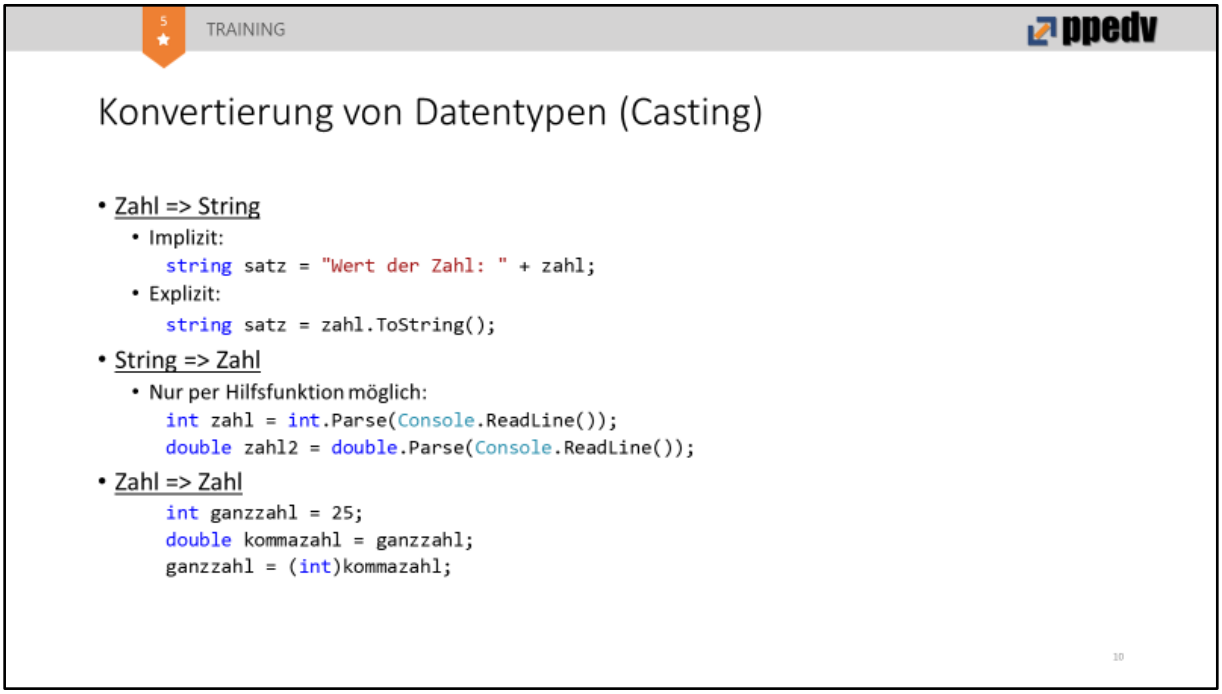


Um Benutzereingaben über die Konsole abzugreifen wird (u.a.) die ReadLine()-Funktion der Console-Klasse verwendet, welche einen String an den Aufrufer zurück gibt und in einer String-Variablen gespeichert werden muss.

Damit mit diesem String z.B. gerechnet werden kann, muss der String zunächst in einen anderen Datentyp konvertiert werden (vgl. nächste Folie).

Mittels der ReadKey()-Methode der Console-Klasse greift man ein einzelnes Zeichen (char) vom Benutzer ab. Diese Funktion wird in Konsolenanwendungen auch als Warte-Befehl verwendet.

Konvertierung von Datentypen (Casting)



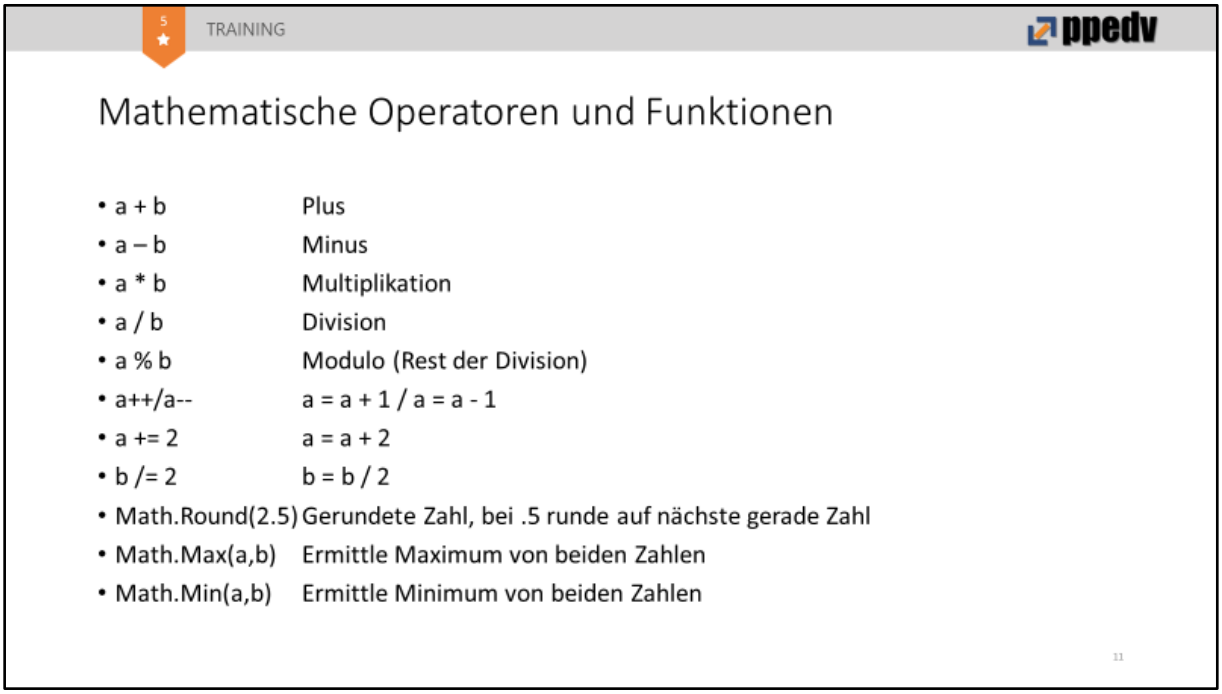
Damit ein Wert/Objekt in einem anderen Kontext als in dem ihres eigenen Datentyps verwendet werden (z.B. mit einer Benutzereingabe rechnen) kann, muss dieser zunächst in einen anderen Typen konvertiert werden.

Dabei wird zwischen einer impliziten (d.h. nicht explizit geforderten) und expliziten Umwandlung unterschieden. Konvertierungen von ungenaueren zu genaueren Datentypen (z. B. int -> double) funktionieren immer implizit, da es hierbei in keinem Fall zu einem Informationsverlust kommen kann. Die umgekehrte Richtung muss explizit erfolgen.

Diese wird durch einen sogenannten Cast durchgeführt, wobei man den neuen Datentyp in runden Klammern vor die umzuwandelnde Variable schreibt. Dabei erlaubt man praktisch den zu erwartenden Informationsverlust (z.B. bei der Umwandlung double -> int werden die Nachkommastellen abgeschnitten).

Eine Umwandlung von String in einen numerischen Datentyp ist nur über die Hilfsfunktion Parse() des jeweiligen neuen Datentyps möglich. Dabei darf der zu parsende String nur Ziffern enthalten.

Mathematische Operatoren und Funktionen

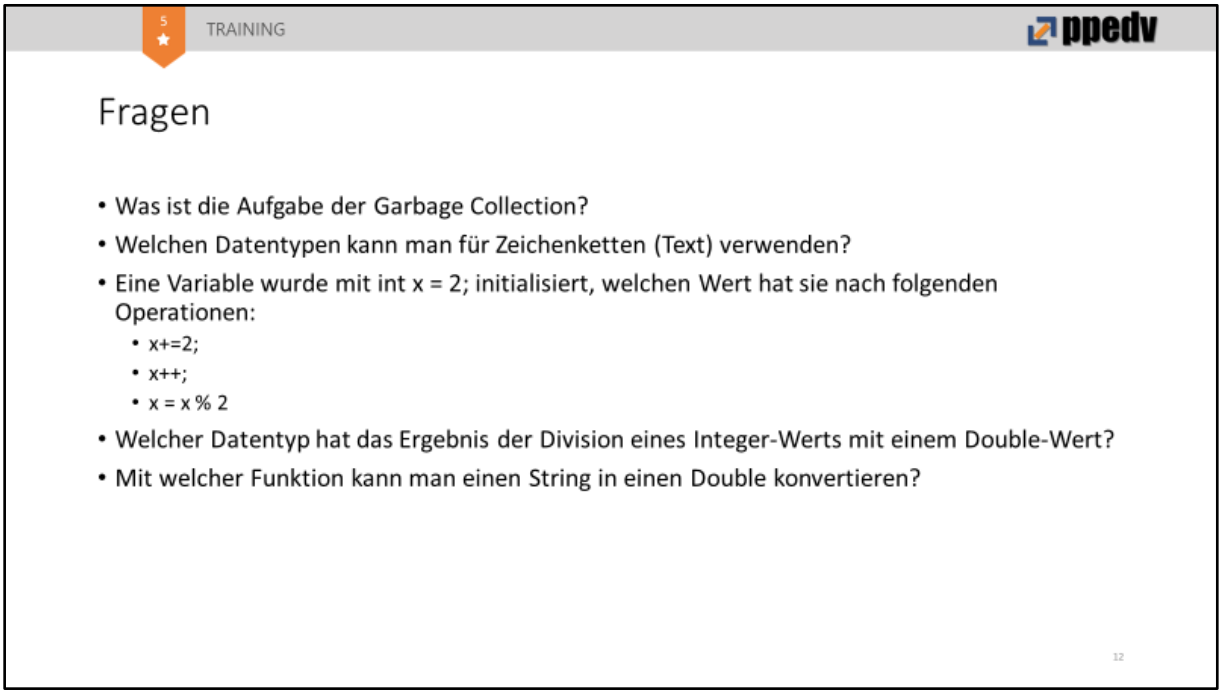


Zur Ausführung von mathematischen Operationen werden die Standart-Zeichen (+ - * /) verwendet.

Will man einen Wert selbst manipulieren kann man die verkürzten Schreibweisen (+= -= *= /=) verwenden.

Die Funktionen der Math-Klasse bieten zusätzliche mathematische Operationen, wie beispielsweise das Runden an.

Fragen



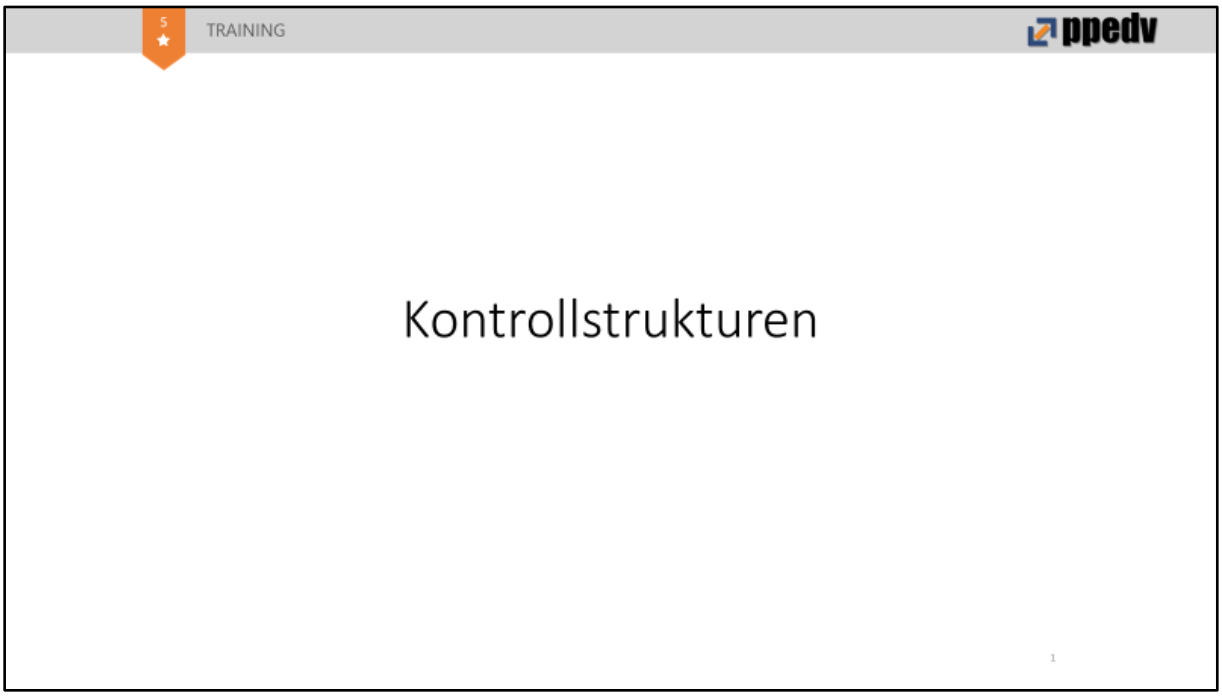
Antworten: Die Garbage Collection löscht alle Objekte aus dem Speicher, welche nicht mehr referenziert (d.h. in Variablen gespeichert) sind.

Für Zeichenketten muss der Datentyp ‚String' verwendet werden.

x=2 -> x+=2 -> x=4 -> x++ -> x=5 -> x=x%2 -> x=1

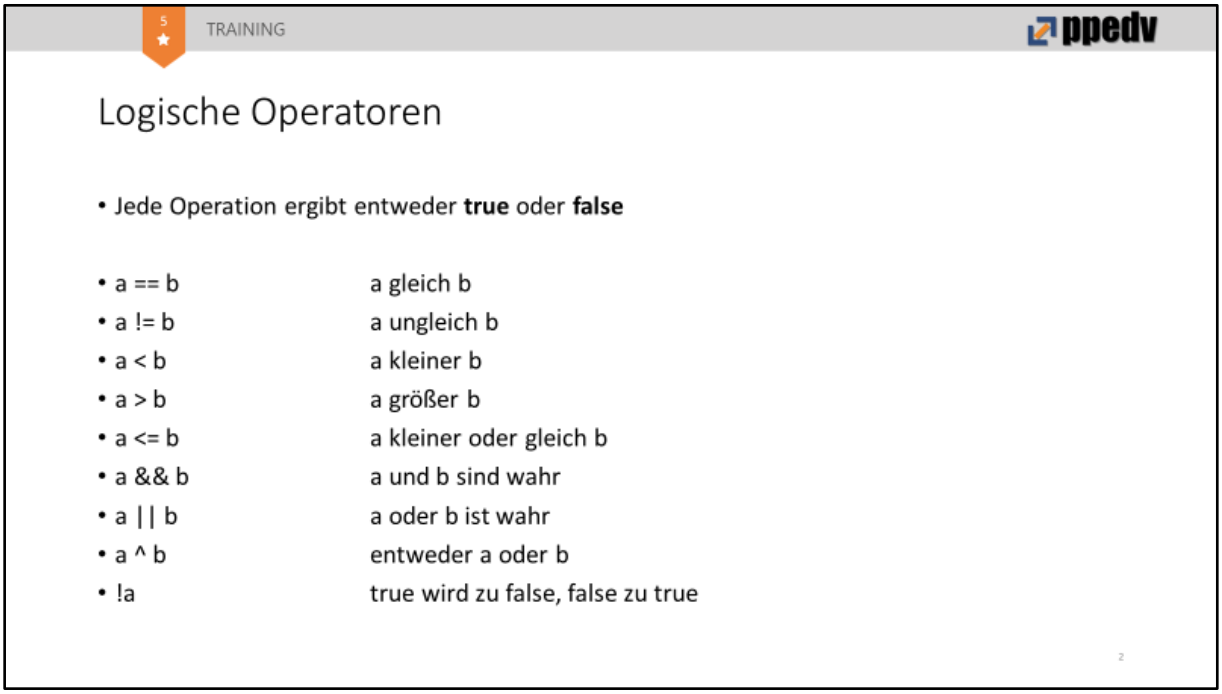
FEHLEN HIER NOCH 2 ANTWORTEN?

Kontrollstrukturen



Kontrollstrukturen bieten diverse Methoden mit denen das Programm auf verschiedene Zustände reagieren und entsprechend unterschiedliche Anweisungen ausführen kann.

Logische Operatoren



Die Zustände mit denen die Kontrollstrukturen kontrolliert werden, werden in boolschen Werten angegeben. Diese können entweder durch eine bool-Variable oder einen logischen Ausdruck dargestellt werden.

Logische Ausdrücke sind Ausdrücke, welche per Definition nur mit den objektiven Werten wahr (true) und unwahr (false) belegt werden können. Die Werte ergeben sich durch die oben abgebildeten logischen Operatoren. Dabei können die größer/kleiner-Operatoren nur auf numerische Werte und der und- (&&), oder- (||) und xor- (^) Operator nur auf boolesche Werte angewendet werden.

Es ist ebenfalls zu beachten, dass bei den hier gezeigten Versionen der und- und oder-Operatoren bedingte Operatoren sind. Dies bedeutet, dass, wenn bei einer Prüfung des ersten Werts bereits ein eindeutiges Ergebnis des Ausdrucks feststeht, eine weitere Prüfung nicht stattfindet.

Möchte man, dass trotzdem der vollständige Ausdruck geprüft wird, sollte man die bitweisen Operatoren (& bzw. |) verwendet.

Bedingungen

TRAINING ppedv

Bedingungen

```
if (Bedingung A)
{
    //Anweisungen wenn Bedingung A wahr
}
else if (Bedingung B)
{
    //Anweisungen wenn Bedingung A falsch und B wahr
}
else
{
    //Anweisungen wenn Bedingung A und B falsch sind
}
```

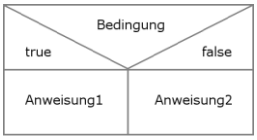
Mit IF-ELSE-Blocken können Bedingungen und deren Konsequenzen definiert werden. Ein IF-ELSE-Block besteht mindestens aus einem IF-Block mit dessen Bedingung. Optional können beliebig viele ELSE-IF-Blocke mit weiteren Bedingungen sowie ein ELSE-Block mit einem Default-Fall definiert werden.

Das Programm wird nun nacheinander die angegebenen Bedingungen prüfen und den Körper der ersten als wahr erkannten Bedingung ausführen. Danach wird das Programm mit dem ‚normalen' Code nach dem Block weitermachen.

Wenn keine Bedingungen der IF/ELSE-IF-Blöcke wahr ist, wird das Programm den Körper des ELSE-Block ausführen.

IF-Blöcke mit nur zwei Möglichkeiten können in einer Kurznotation geschrieben werden: (Bedingung) ? (true-Anweisung) : (else-Anweisung) Die true-Anweisung wird ausgeführt, wenn die Bedingung wahr ist. Ansonsten wird die else-Anweisung ausgeführt.

IDEE: Schleifen grafisch darstellen



Zahlenratespiel (Teil 1)

Mittels der Zuweisung ‚Random generator = new Random();' wird ein neues Objekt der Klasse Random in einer Variable vom Typ Random mit dem Namen ‚generator' erzeugt.

Objekte der Random-Klasse können mittels der Next()-Funktion zufällige Zahlen in dem angegeben Bereich erzeugen.

Zu beachten ist dabei, dass der erste Parameter der Next()-Funktion ein inklusiver Wert, der zweite Parameter aber ein exklusiver Wert ist. D.h. die erzeugte Zahl wird maximal eins kleiner als der zweite Parameter sein.

Schleifen

TRAINING ppedv

Schleifen

Kopfgesteuert

```
while (Bedingung)
{
    //wiederholt solange bis Bedingung falsch ist
}
```

Fußgesteuert

```
do
{
    //führt mindestens 1 Mal aus
} while (Bedingung);
```

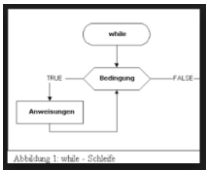
Schleifen wiederholen den Code in ihrem Körper solange die Bedingung wahr ist. Um Endlosschleifen zu vermeiden, sollte die Bedingung zu Laufzeit der Schleife manipuliert werden.

Die WHILE-Schleife prüft die Bedingung, bevor sie den nächsten Durchlauf startet. Ist die Bedingung falsch, führt das Programm den Code hinter der Schleife fort. Ist die Bedingung von vornherein falsch, wird die die Schleife übersprungen.

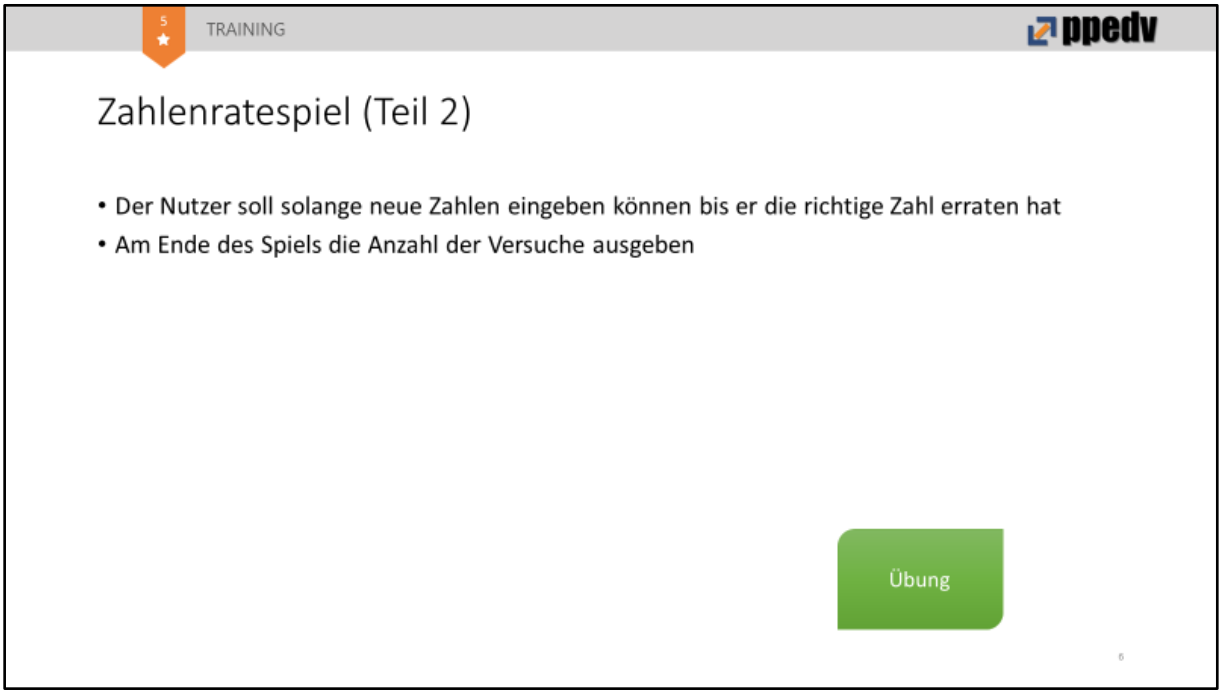
Die DO-WHILE-Schleife prüft die Bedingung immer am Ende eines Schleifendurchlaufs und beginnt im Wahrheitsfall einen neuen Durchlauf. Daraus folgt, dass diese Schleife vor der ersten Prüfung der Bedingung in jedem Fall einmal durchläuft.

Innerhalb von Schleifen können zwei Schlüsselwörter verwendet werden, um die Schleife zu manipulieren: Mit ‚break' wird die Schleife sofort verlassen und der Code nach der Schleife wird ausgeführt. Mit ‚continue' wird der aktuelle Schleifendurchlauf abgebrochen und es wird eine erneute Prüfung der Bedingung erzwungen. Ist diese wahr wird ein erneuter Schleifendurchlauf begonnen.

IDEE: Schleifen grafisch darstellen

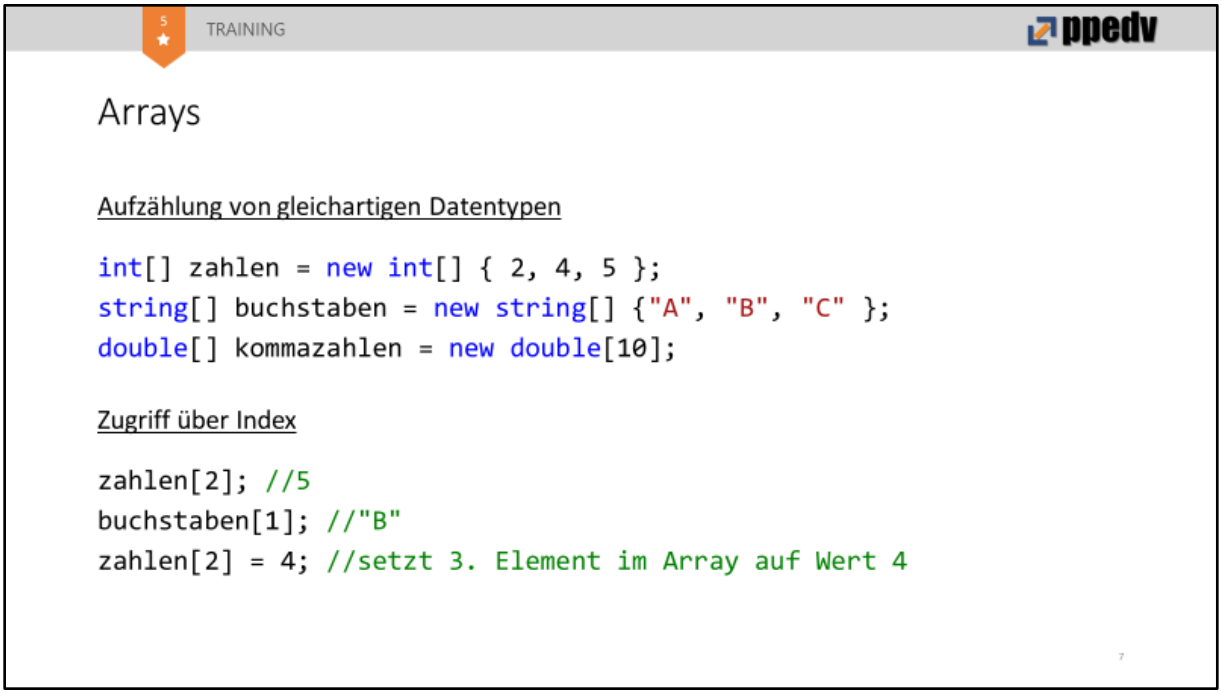


Zahlenratespiel (Teil 2)



Quellcode kommentiert einbinden?!?!?

Arrays



Arrays sind in ihrer Länge (Anzahl der maximalen Elemente) feste Sammlungen von Werten/Objekten eines einzelnen Datentyps. Deklariert werden sie mit eckigen Klammern [ ] hinter dem Datentyp.

Bei ihrer Initialisierung muss ihre Länge definiert werden. Entweder muss dies durch die direkte Angabe der Länge in den eckigen Klammern des initialisierten Objekts geschehen oder durch explizite Angabe der beinhaltenden Elemente in geschweiften Klammern.

Zugegriffen wird auf die Elemente eines Arrays über die Angabe dessen Nullbasierenden Index, wobei die Elemente in der Reihenfolge ihres Vorkommens in den geschweiften Klammern in das Array geschrieben werden.

Man kann bei der Array-Initialisierung sowohl auf das ‚new'-Stichwort sowie auf den Datentyp verzichten, wenn man die Elemente in die geschweiften Klammern schreibt. Beispiel: string[] stringArray = {„ab", „cd", „ef"};

Zähl-Schleifen

TRAINING ppedv

Zähl-Schleifen

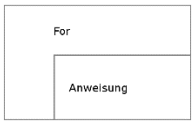
```
for(Initialisierung;Bedingung;Anweisung)

Hochzählen(Inkrementieren):
for (int i = 0; i<10; i++)
{
    //Zähle von 0 bis 9
}

Runterzählen(Dekrementieren):
for (int i = 10; i>0; i--)
```

FOR-Schleifen sind Schleifen, bei welchen die Anzahl der Durchläufe durch einen Laufindex festgelegt wird. Bei der Initialisierung muss eine Variable als Laufindex deklariert und initialisiert werden.

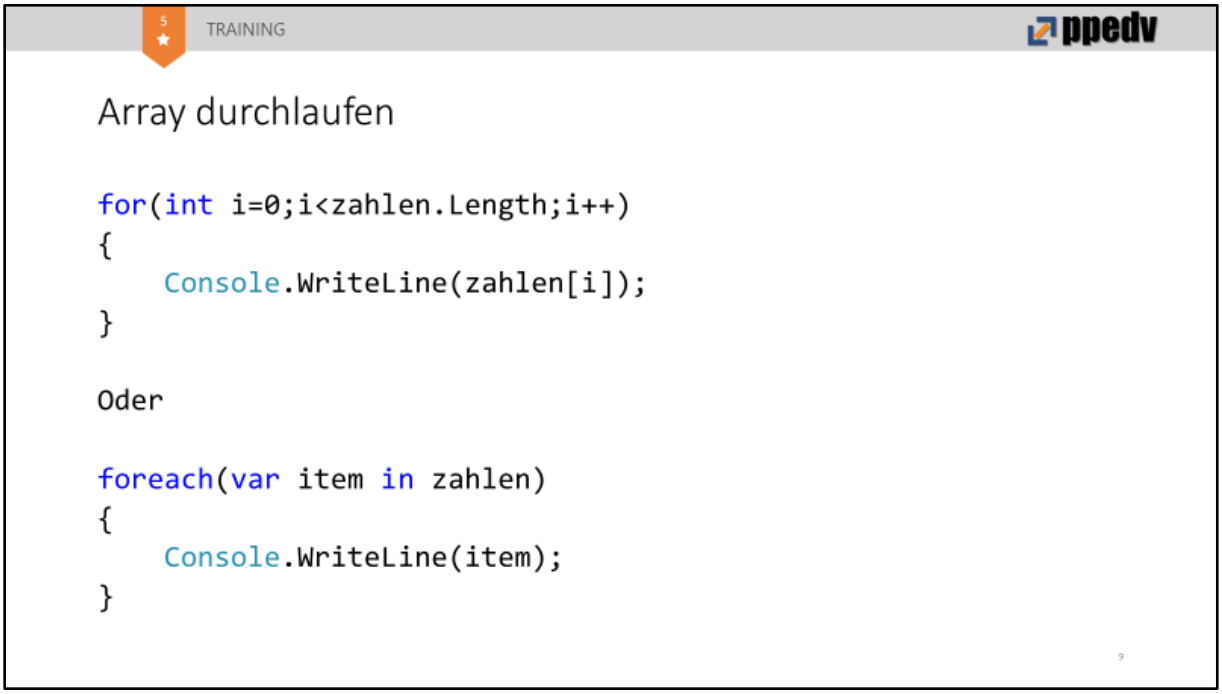
Daneben müssen eine Bedingung sowie eine Anweisung angegeben werden. Zu Beginn jedes Durchlaufs wird die Bedingung geprüft und im wahr-Fall wird der Durchlauf gestartet.

Am Ende jedes Durchlaufs wird die Anweisung ausgeführt. Dies wird solange wiederholt, bis die Anweisung nicht mehr wahr ist.

SCHLEIFE ALS GRAFIK



Array durchlaufen

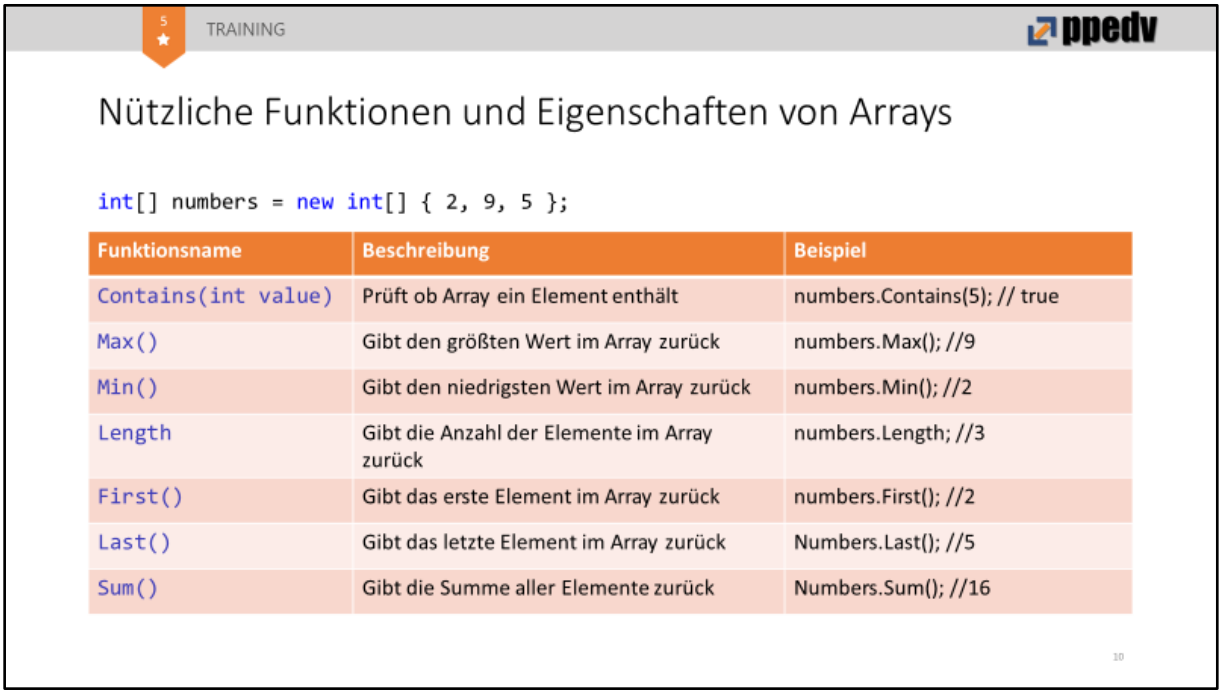


Möchte man eine Sammlung, wie ein Array, durchlaufen lassen kann man in eine FOR-Schleife deren Länge (über die Array-Eigenschaft .Length) als Bedingung eintragen (s.o.). Nun kann man den Laufindex über das Array laufen lassen und spricht so jedes Element genau einmal an.

Zudem hat jede Sammlung in C# die Möglichkeit diese mit einer speziellen FOREACH-Schleife durchlaufen zu lassen. Diese Schleifen sind so definiert, dass sie jedes Element genau einmal betrachten.

Die Platzhaltervariable ‚item' entspricht dabei immer dem Sammlungs-Element, welches im aktuellen Schleifendurchlauf gerade betrachtet wird.

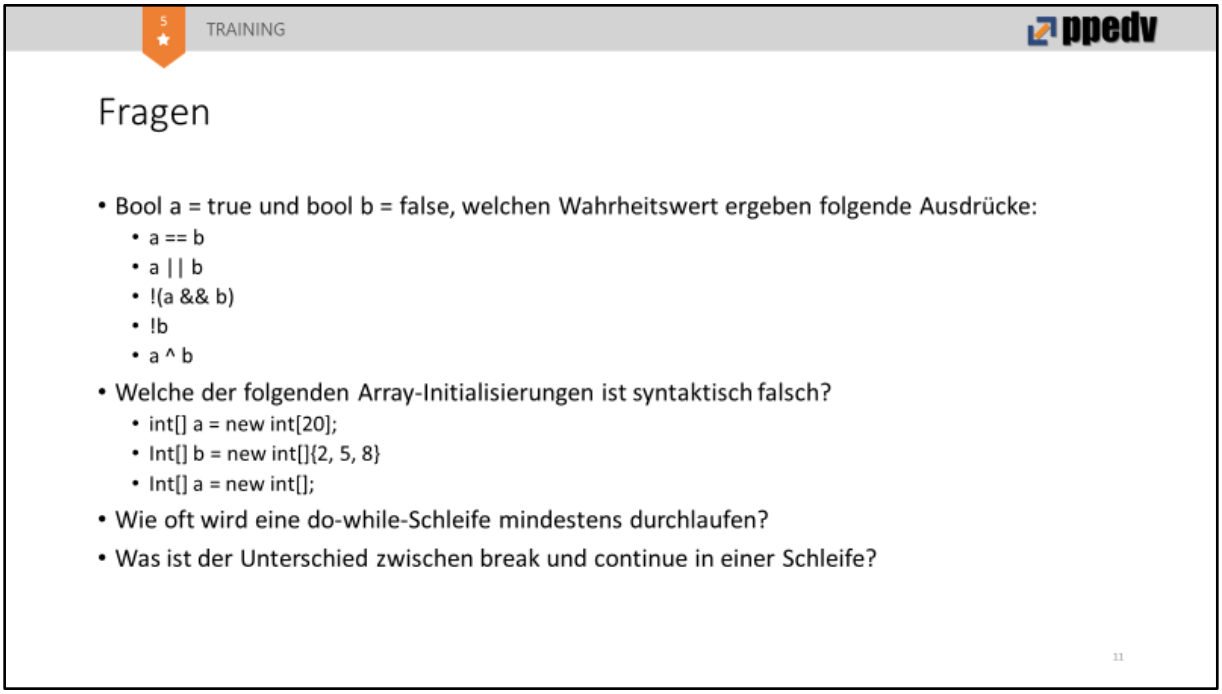
Nützliche Funktionen und Eigenschaften von Arrays



Neben der .Length-Eigenschaft, welche die Anzahl der Elemente eines Arrays ausgibt, besitzen Arrays noch andere nützliche Eigenschaften und Funktionen.

Zu beachten ist, dass nicht alle diese Funktionen in allen Array-Typen vorhanden sind. So ist z.B. die Funktion Sum() nur bei numerischen Datentypen vorhanden.

Fragen



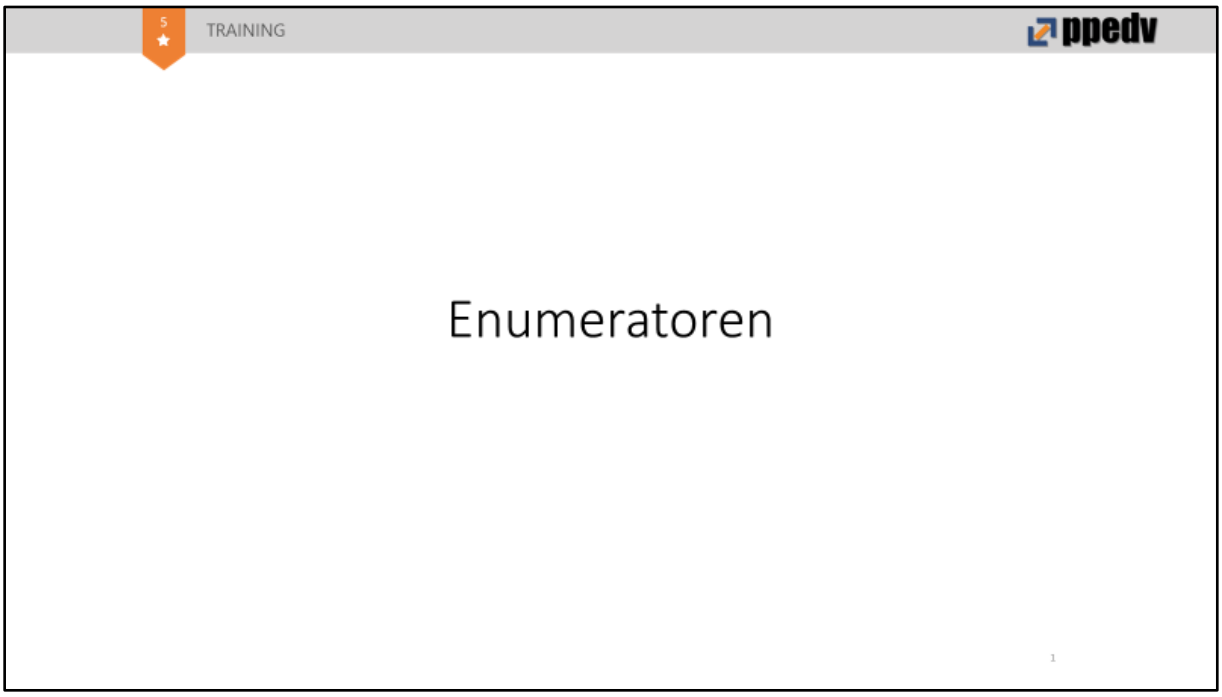
Antworten: a==b ◊ true==false ◊ false a||b ◊ true || false ◊ true !(a&&b) ◊ !(true && false) ◊ !(false) ◊ true !b ◊ !false ◊ true a^b ◊ true^false ◊ true

Bei ‚int[] a = new int[];' ist keine Länge definiert.

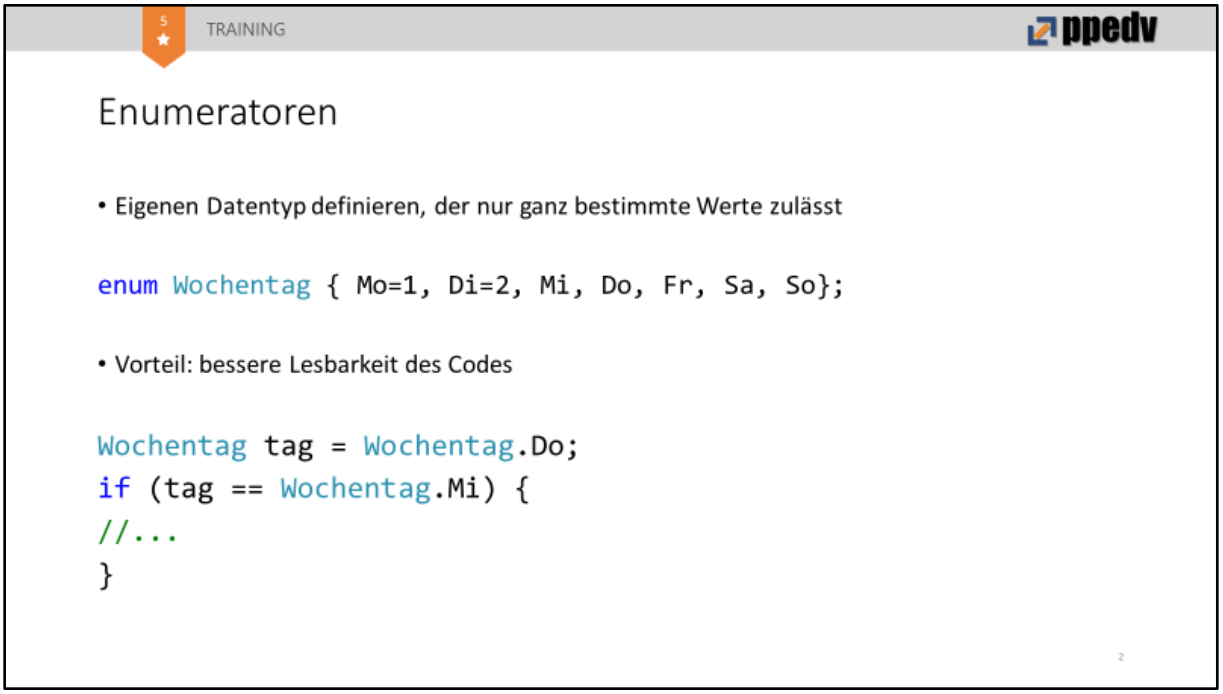
Jede DO-WHILE-Schleife wird mindestens einmal durchlaufen, da die Bedingung erst am Ende jedes Durchlaufs geprüft wird.

'Break' beendet die aktuelle Schleife vollständig während ‚continue' nur den aktuellen Durchlauf unterbricht und die Bedingung erneut prüfen lässt.

Enumeratoren



Enumeratoren



Enumeratoren sind selbst definierte Datentypen, welche nur fest vorgegebene Zustände zulassen.

Dabei ist jedem der Zustände ein Integer-Wert zugewiesen, so dass zwischen diesen Werten eine explizite Umwandlung per Cast möglich ist. Werden diese Werte nicht selbst zugewiesen, geschieht dies automatisch, wobei der erste Wert der 0 zugewiesen wird.

Der Vorteil eines Enumerators bei der Zustandsanzeige gegenüber anderen Datentypen wie beispielsweise String oder Integer, ist die Sicherheit, dass bei der Übergabe von Enumeratortypen nur die definierten Zustände erreicht werden können. Zudem entsteht bei menschlichen Lesern eine verbesserte Lesbarkeit des Codes.

Switch

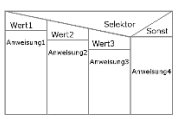
TRAINING ppedv

Switch

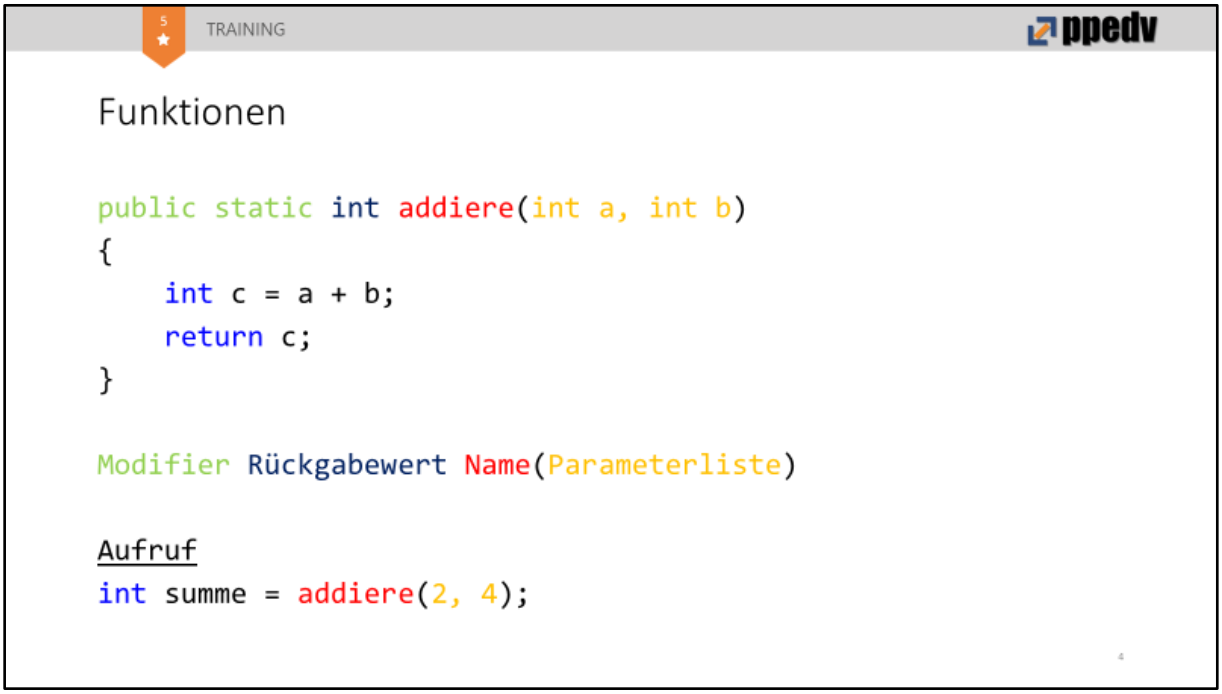
```
switch (tag)
{
    case Wochentag.Mo:
          Anweisung A;
        break;
    case Wochentag.Di:
    case Wochentag.Mi:
          Anweisung B;
        break;
    default:
          Anweisung C;
        break;
}
```

Der SWITCH ermöglich eine verkürzte Schreibweise eines IF-ELSE-Blocks in Bezug auf eine einzelne Variable. Dabei werden die möglichen Zustände, welche die oben angegebene Variable einnehmen kann, in CASES aufgeteilt, welche jeweils die auszuführenden Anweisungen beinhalten. Jeder CASE muss mit einem break; beendet werden. Die einzige Ausnahme diesbezüglich ist ein CASE, welcher keine Anweisungen enthält. Wird hier das ‚break' weggelassen, führt das Programm die Anweisungen des nachfolgenden CASES aus. Nicht jeder mögliche Zustand muss angegeben werden. Nimmt die Variable einen nicht näher definierten Zustand an, werden die Anweisungen des Default-CASE ausgeführt.

- Anweisung A wird ausgeführt wenn tag = Montag ist
- Anweisung B wird ausgeführt wenn tag = Dienstag oder Mittwoch ist
- An allen anderen Tagen wird Anweisung C ausgeführt.
- Seit C#7 kann auf alle Datentypen switch angewendet werden (zuvor nur Grunddatentypen)
- Wird ein komplexer Datentyp in switch abgefragt, kann dieser auf bestimmte Typen geprüft werden
- z.B.: Switch(obj){case Datentyp variable when variable.X > 0: Anweisung;break;}
- when-Operator beschreibt Zustand des Objektes näher



Funktionen



Eine Funktion (Methode) ist ein Bündel von Anweisungen, welches einer bestimmten Klasse zugeordnet sind.

Jede Funktionsdeklaration besteht aus einer Signatur und einem Funktionskörper.

Deie Signatur besteht aus den Modifiern, welche u.a. die Zugreifbarkeit der Funktion bestimmen, einem Rückgabetyp, welcher nach Abschluss der Funktion an den Aufrufer zurückgegeben wird, einem selbst zu bestimmenden Namen sowie einer Liste mit Datentypen und Bezeichnern, welche als Parameter vom Aufrufer an die Funktion übergeben werden.

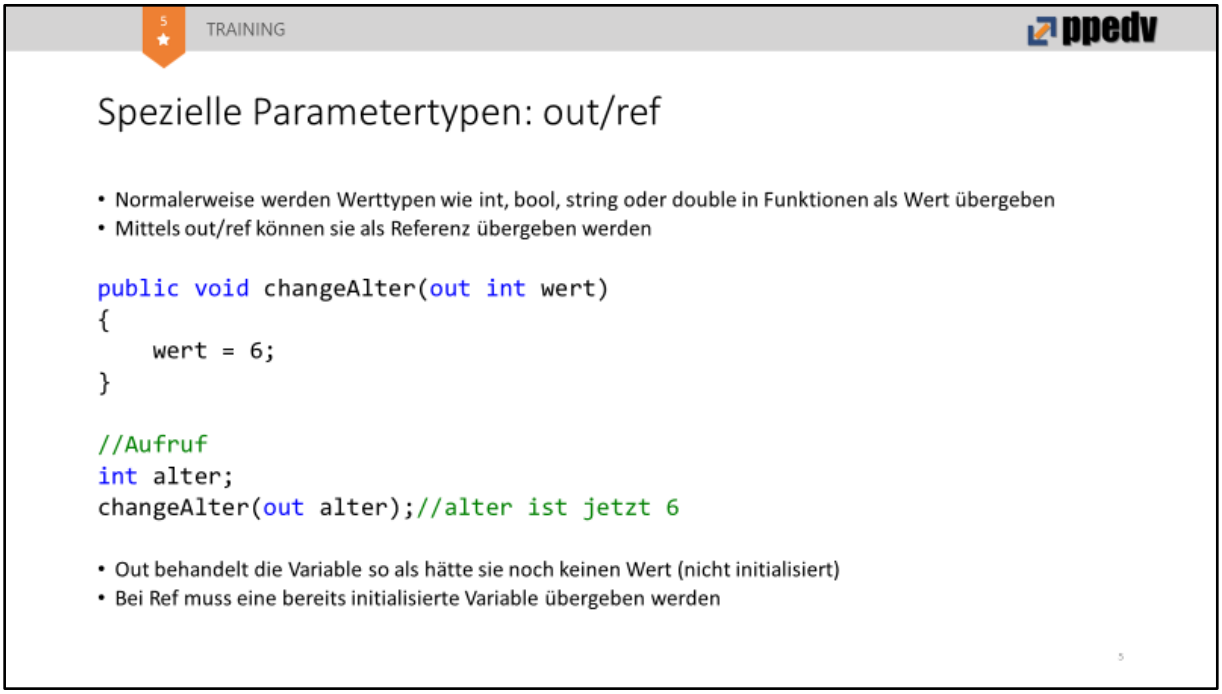
Im Körper stehen die Anweisungen, welche nach Funktionsaufruf ausgeführt werden. Jeder Kausalitätspfad in einer Funktion benötigt einen ‚return'-Befehl, auf welchen ein Wert/Objekt vom Typ des Rückgabetyps folgt. Dies wird an den Aufrufer zurückgegeben. Im Moment der Ausführung des ‚return'-Befehls wird die Funktion beendet.

Alternativ kann anstelle eines Rückgabetypen in der Signatur auch das Stichwort ‚void' angegeben werden. Dies sagt aus, dass die Funktion keinen Rückgabetyp besitzt. In diesem Fall kann das ‚return'-Stichwort nicht verwendet werden.

Der Aufruf einer Funktion erfolgt über ihren Namen in Kombination mit den verlangten Übergabeparametern. Besitzt die Funktion einen Rückgabewert, den man verwenden möchte, muss dieser in einer Variablen des entsprechenden Typs gespeichert werden.

Es können in einer Klasse mehrere Funktionen mit dem selben Namen definiert werden, wenn diese unterschiedliche Übergabeparameter besitzen. Diese Funktionen nennt man überladene Funktionen.

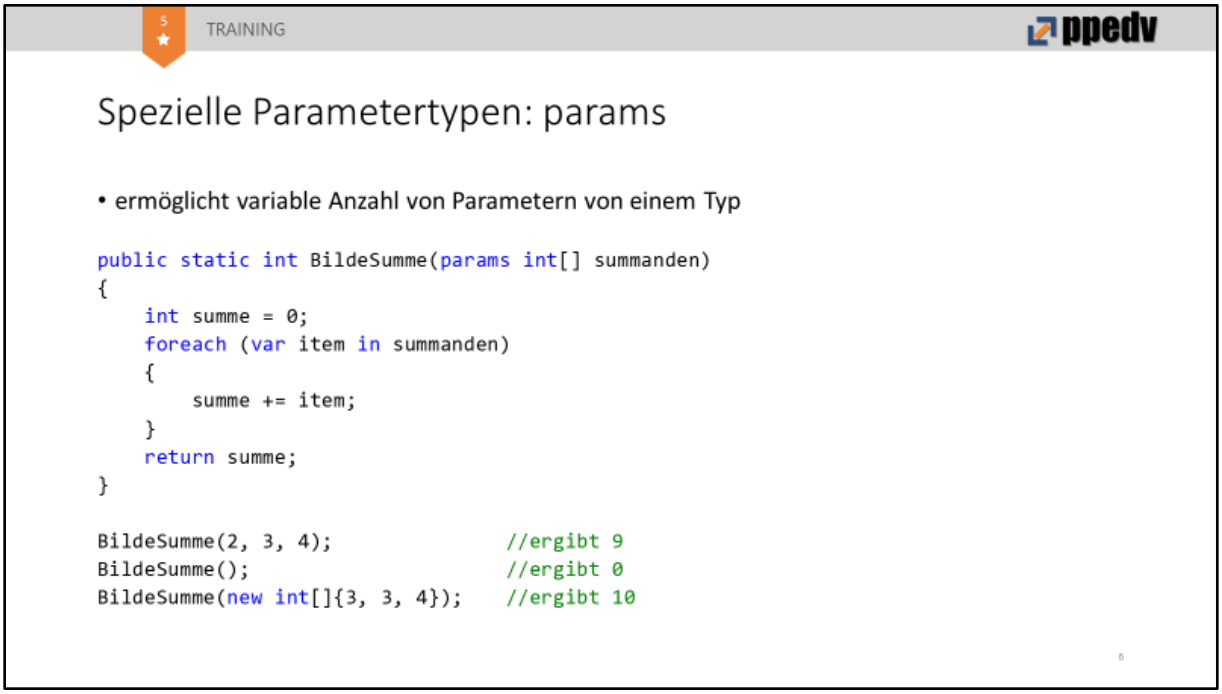
Spezielle Parametertypen: out/ref



Verwendet man in den Parametern einer Funktion das Stichwort ‚out', kann man an dieser Stelle einen weiteren Rückgabewert definieren. Man übergibt eine Variable des gewünschten Typs (welche auch erst im Aufruf deklariert werden kann) an die Funktion. Ruft man diese Variable nach Funktionsaufruf erneut auf besitzt diese den Wert, welcher ihr in der Funktion zugewiesen wurde.

Mit dem Stichwort ‚ref' kann man einer Funktion eine bereits initialisierte Variable übergeben, welche von der Funktion als Referenzwert behandelt wird. Dies bedeutet, dass in der Funktion keine Kopie des Werts behandelt wird, sondern der Wert der Variable selbst. Greift man nach Funktionsaufruf auf die Variable zu, hat sie den neuen, durch die Funktion definierten Wert.

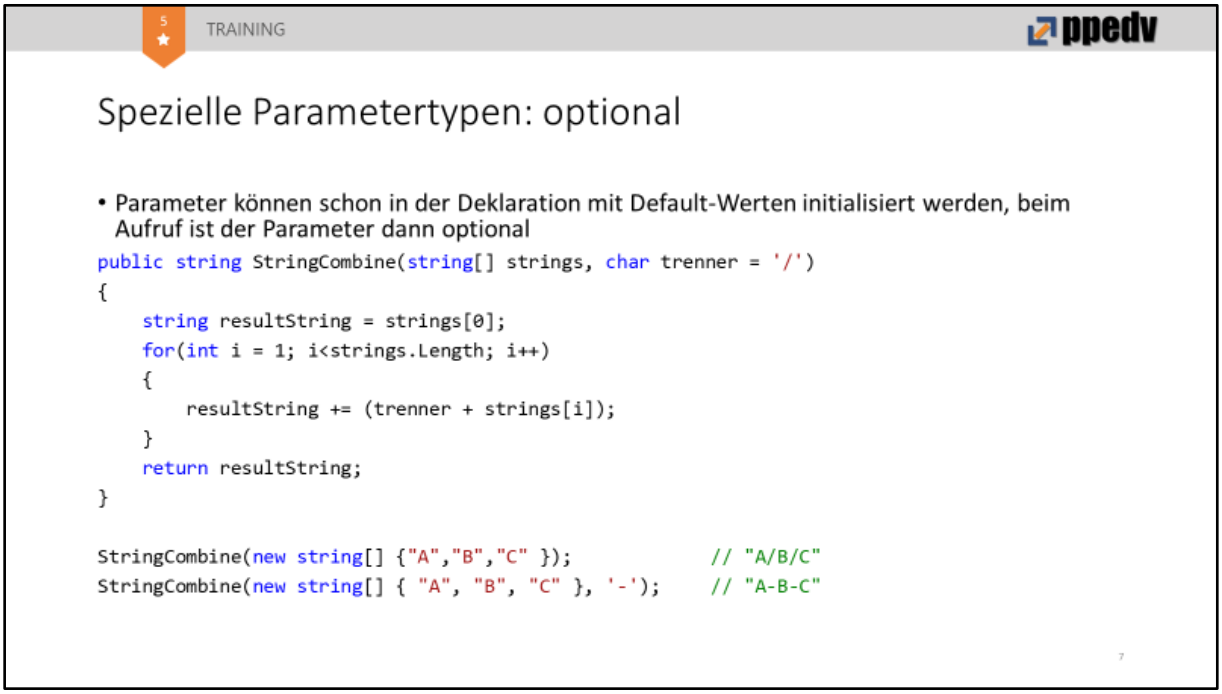
Spezielle Parametertypen: params



Mittels des ‚params'-Stichwort kann man einer Funktion die Möglichkeit geben, als Übergabeparameter ein Array unbestimmter Größe zu verlangen.

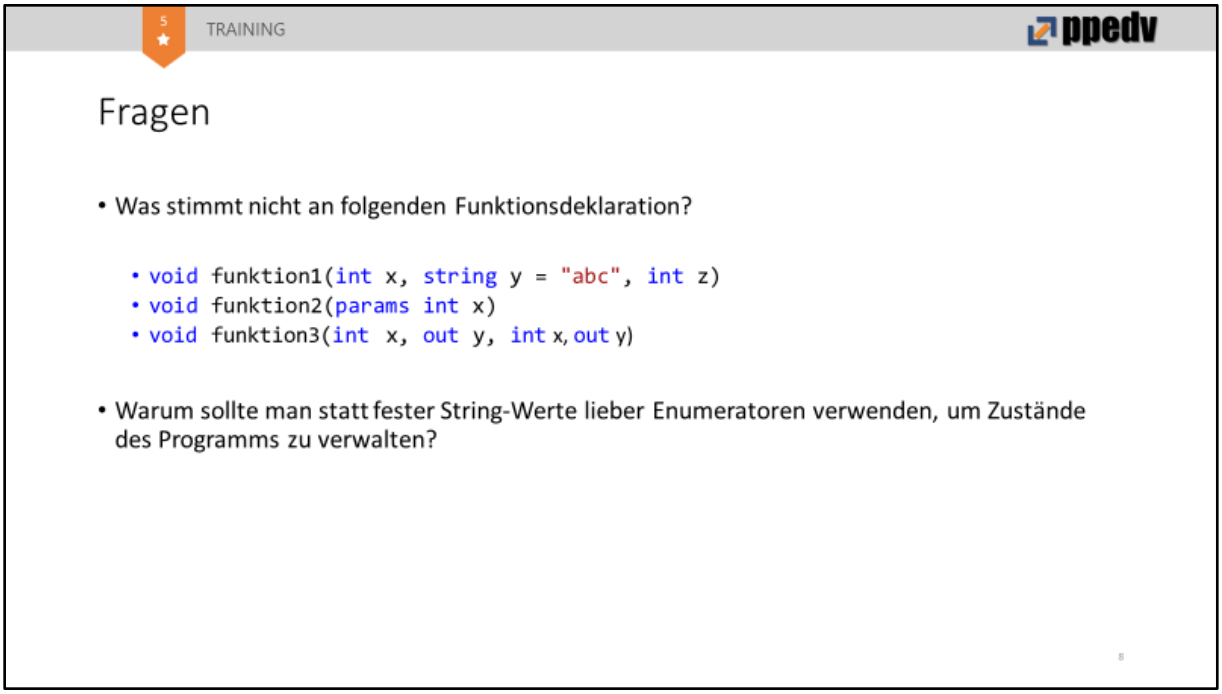
Dies kann bei Aufruf in Form eines Arrays oder als durch Kommata getrennte Einzelwerte übergeben werden. In der Funktionen werden beide Fälle als Array behandelt.

Spezielle Parametertypen: optional



Man kann in Funktionsdeklarationen Übergabeparametern Strandartwerte zuweisen, in dem man diese mittels =-Zeichen mit dem Parameter verknüpft. Diese sogenannten optionalen Parameter müssen immer die letzten Positionen der Parameterliste einnehmen. Werden beim Aufruf für diese Parameter keine expliziten Werte übergeben, verwendet die Funktion die Standartwerte.

Fragen



Antworten:

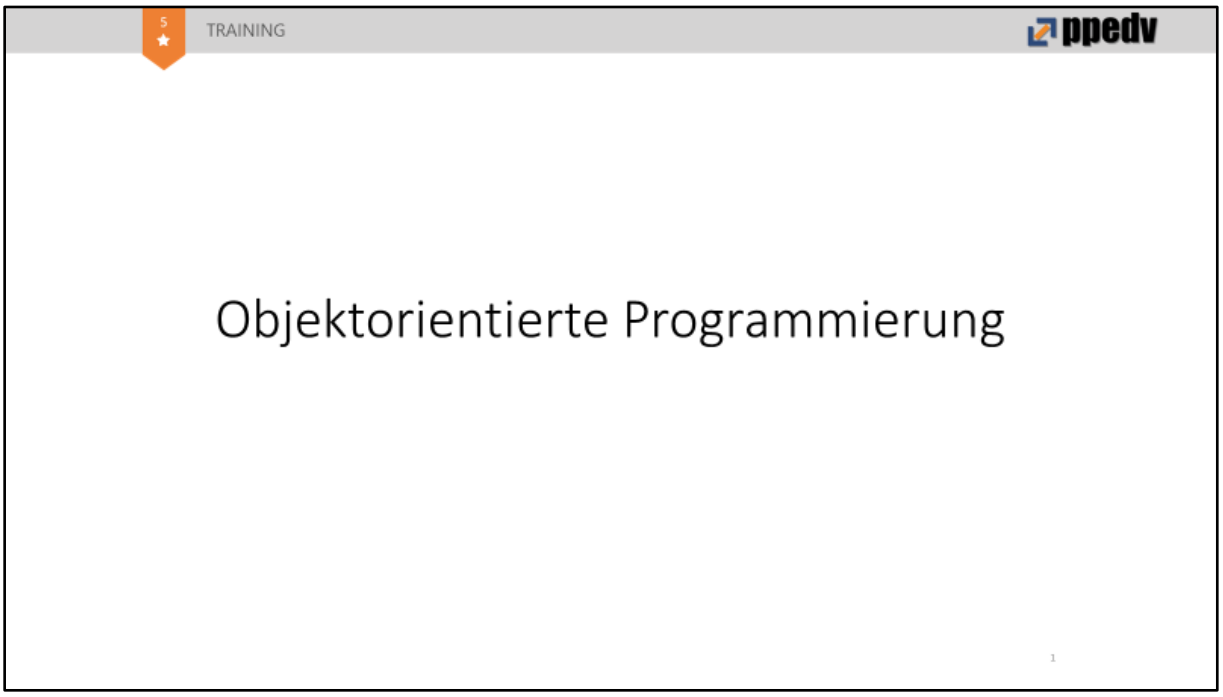
Funktion 1: Der optionale Parameter ist nicht an der letzten Position der Parameterliste.

Funktion 2: Wenn das ‚params'-Stichwort verwendet wird, muss als Parameter ein Array angegeben werden.

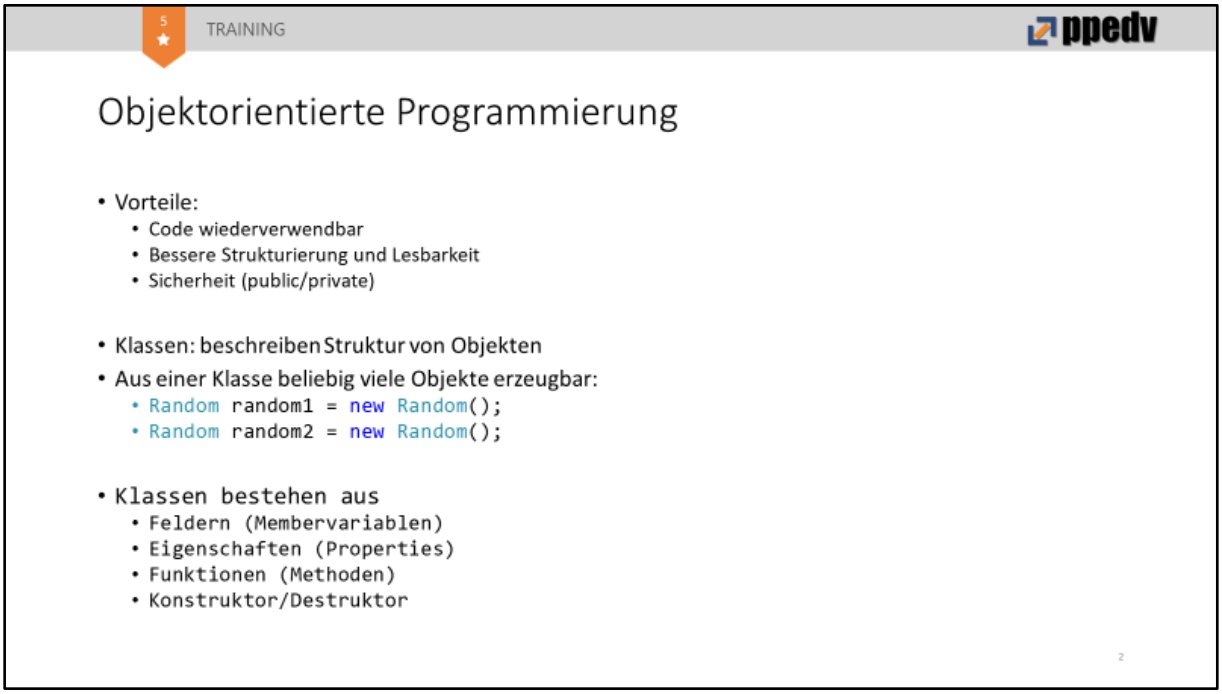
Funktion 3: Auch, wenn das ‚out'-Stichwort verwendet wird, muss ein Datentyp angegeben werden.

Enumeratoren definieren die möglichen Zustände als feste Werte. Würde man einen String verwenden, könnte einer aus einer unendlichen Anzahl an Zuständen übergeben werden

Objektorientierte Programmierung



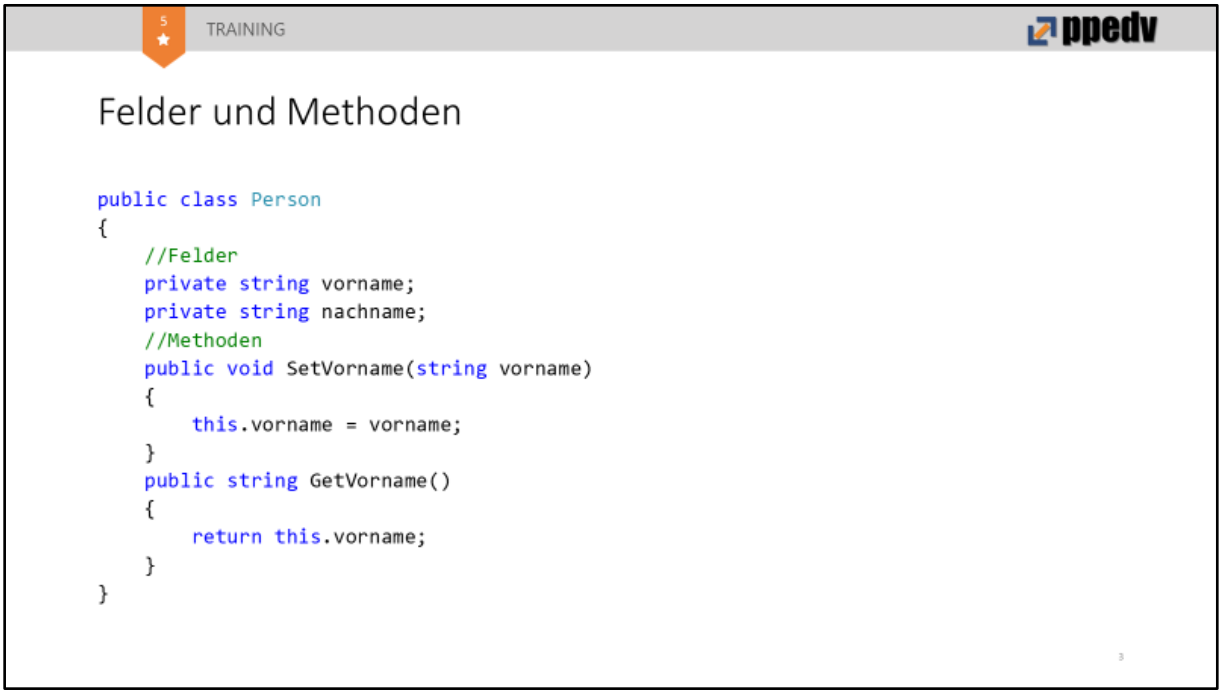
Objektorientierte Programmierung



In der objektorientierten Programmierung werden Programmfunktionen in sogenannten ‚Klassen' strukturiert, welche jeweils hoch spezialisierte Aufgaben erfüllen. Diese Klassen fungieren als Schablonen für gleichartige Objekte. Jedes Objekt einer Klasse besitzt die gleichen Variablen und Funktionen. Dies führt zu einer verbesserten Strukturierung und einer erhöhten Lesbarkeit und Übersichtlichkeit des Codes im Gegensatz zur prozeduralen/imperative Programmierung. Zudem kann durch die Zugangsmodifier die Zugriffsmöglichkeit für einzelne Variablen und Funktionen genauer gesteuert werden.

Aus jeder Klasse können beliebig viele Objekte erstellt (instanziiert) werden. Jedes dieser Objekte besitzt die von der Klasse vorgegebene Struktur, agiert jedoch völlig unabhängig von den anderen Objekten bzw. der Klasse selbst.

Felder und Methoden

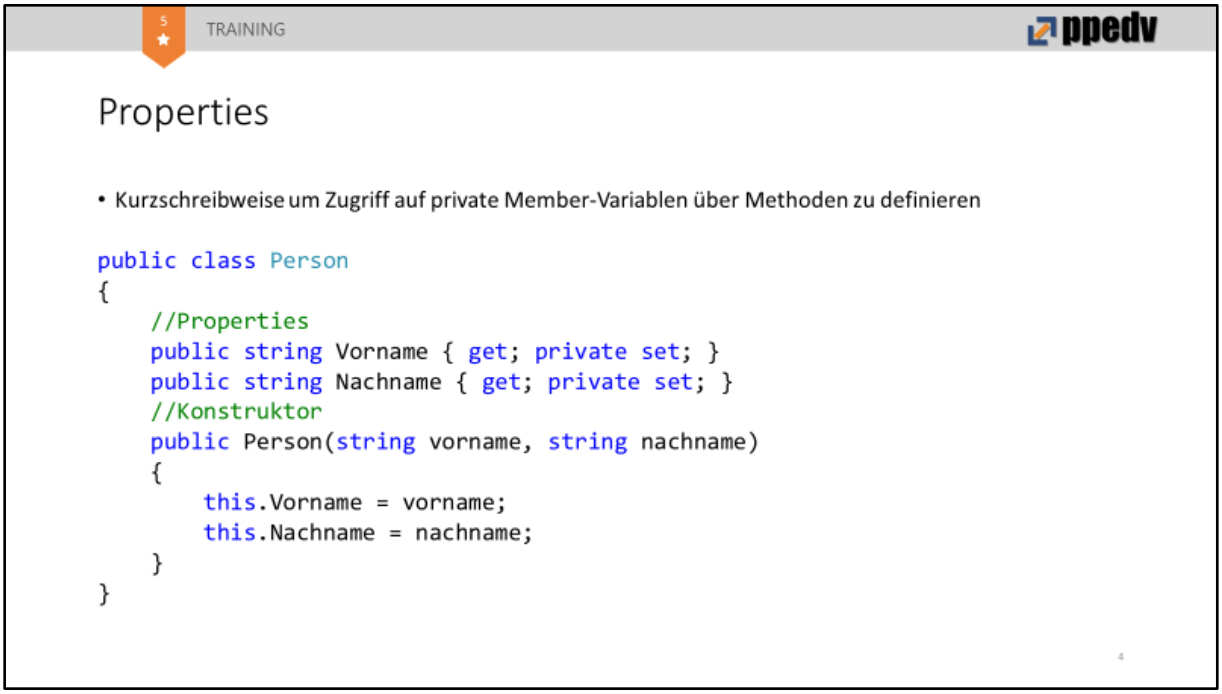


Felder (Membervariablen) sind am Objekt hängende Variablen, welche dieses beschreiben. Dabei ist das Prinzip der Kapselung zu beachten. Dieses besagt, dass Felder nur durch das Objekt selbst direkt manipuliert werden dürfen.

Daraus folgt, dass sowohl Lese- als auch Schreibzugriff von außen nur über Funktionen des Objekts selbst erfolgen dürfen und die Felder selbst einen ‚private'-Modifier besitzen. Traditionell erfolgt dieser Zugriff über Get- und Set- Methoden.

Funktionen (Methoden) eines Objekts sind fest mit diesem Objekt verbundene Funktionen, welche dieses in irgendeiner Art und Weise manipulieren. Genau wie die Felder werden sie in die Klasse geschrieben, existieren aber pro instanziiertem Objekt einmal.

Properties

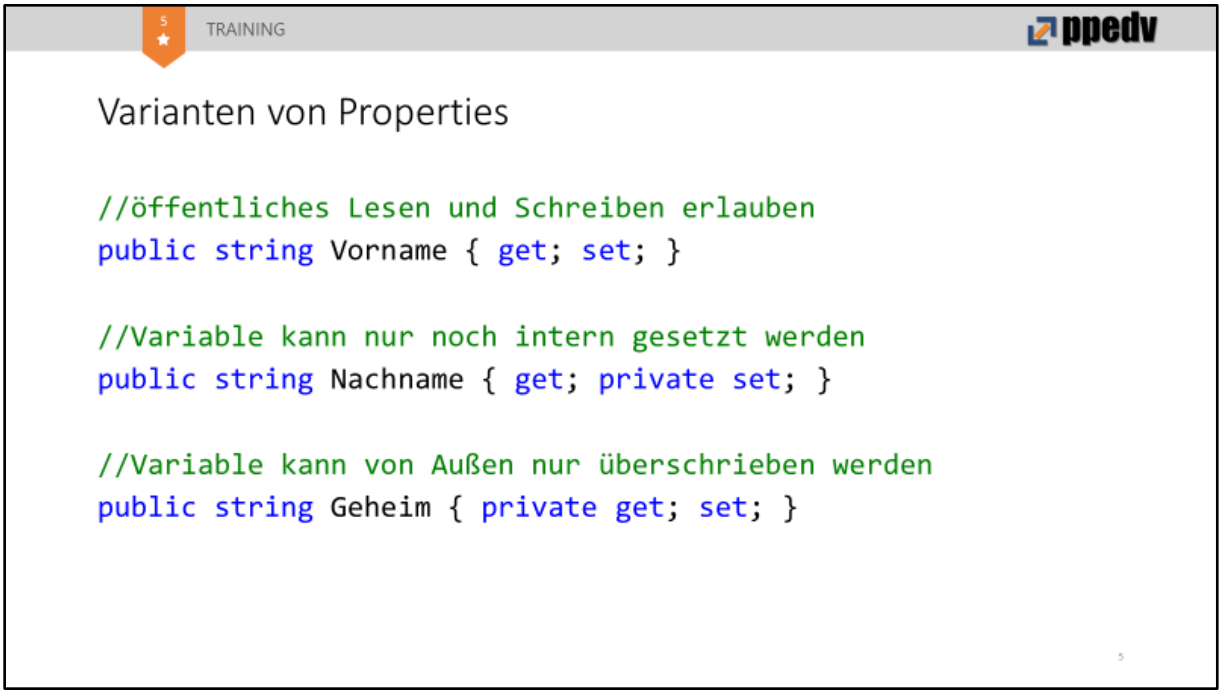


Properties (Eigenschaften) vereinfachen die Formulierung von Get- und Set-Methoden. Dabei ist jeder Property ein Feld zugewiesen.

⚠ Es ist zu beachten, dass die Property selbst kein Feld ist, sondern nur eine ‚Tür' zu dem jeweiligen Feld bereitstellt, welche den Zugriff auf das Feld kontrolliert.

Wenn man keine manuelle Verknüpfung zwischen einem Feld und der Property herstellt, generiert der Compiler ein unsichtbares Feld, welches er mit der Property verknüpft.

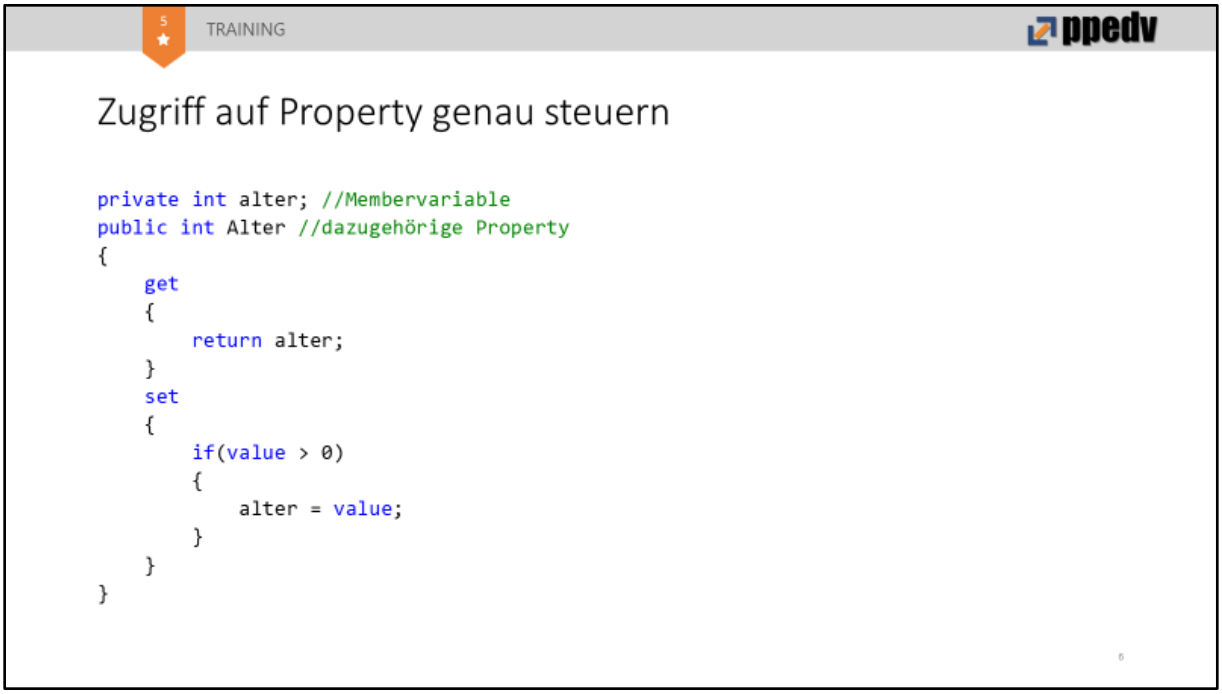
Varianten von Properties



In dem Properties können Lese- und Schreibzugriff getrennt gesteuert werden, indem man dem ‚get' bzw. dem ‚set'-Sitchwort eigene Zugriffsmodifier zuweist.

Es kann z.B. auch vollständig auf das ‚set'-Stichwort verzichtet werden, wodurch die Variable nur noch bei der Objekterstellung gesetzt werden kann.

Zugriff auf Property genau steuern



Will man das Verhalten von Lese- bzw. Schreibzugriff detaillierter steuern, als nur durch Zugriffsmodifier, müssen die Methoden vollständig ausgeschrieben werden und wie oben gezeigt mit einem Feld verknüpft werden.

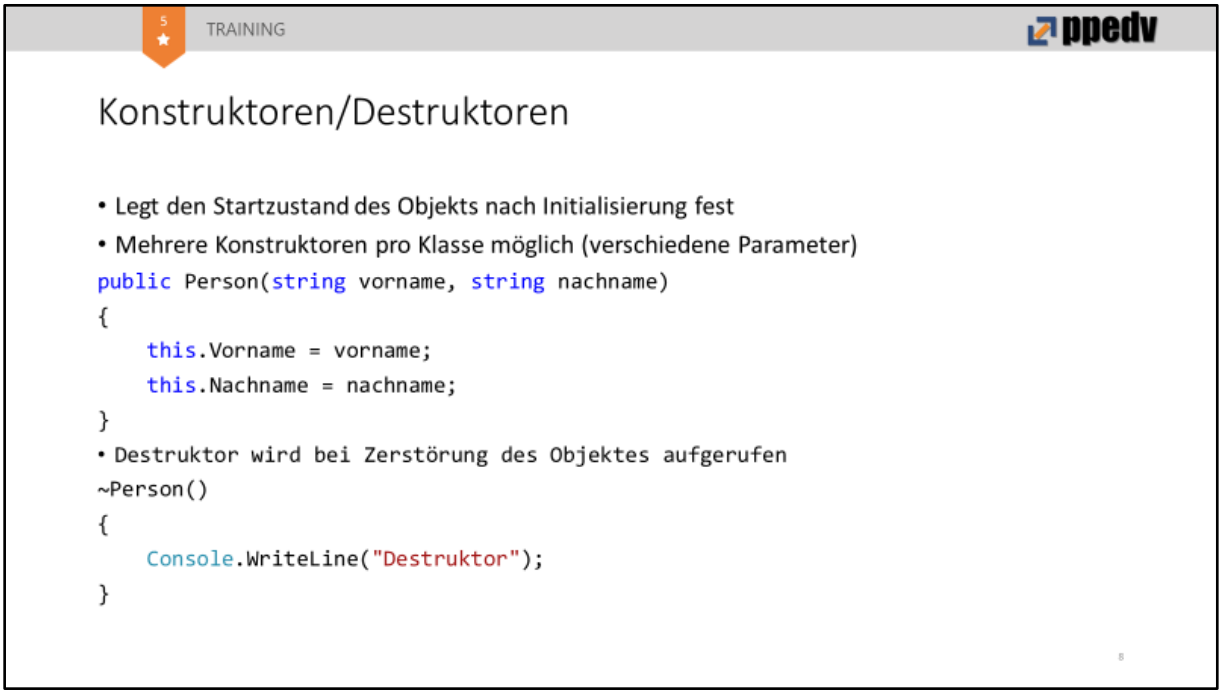
In der Set-Methode wird das Stichwort ‚value' verwendet um den übergebenen und neu zuzuweisenden Wert zu repräsentieren.

Modifier

Durch Zugriffsmodifier kann der Zugang zu Properties, Klassen, Feldern und Funktionen gesteuert werden.

Neben den oben beschriebenen Modifieren existiert noch die Kombination ‚protected internal' wodurch das so geschützte Element innerhalb der gleichen Assembly ‚public' ist, in anderen Assemblies jedoch ‚protected', d.h. es kann in anderen Assemblies nur in abgeleiteten Klassen verwendet werden.

Konstruktoren/Destruktoren



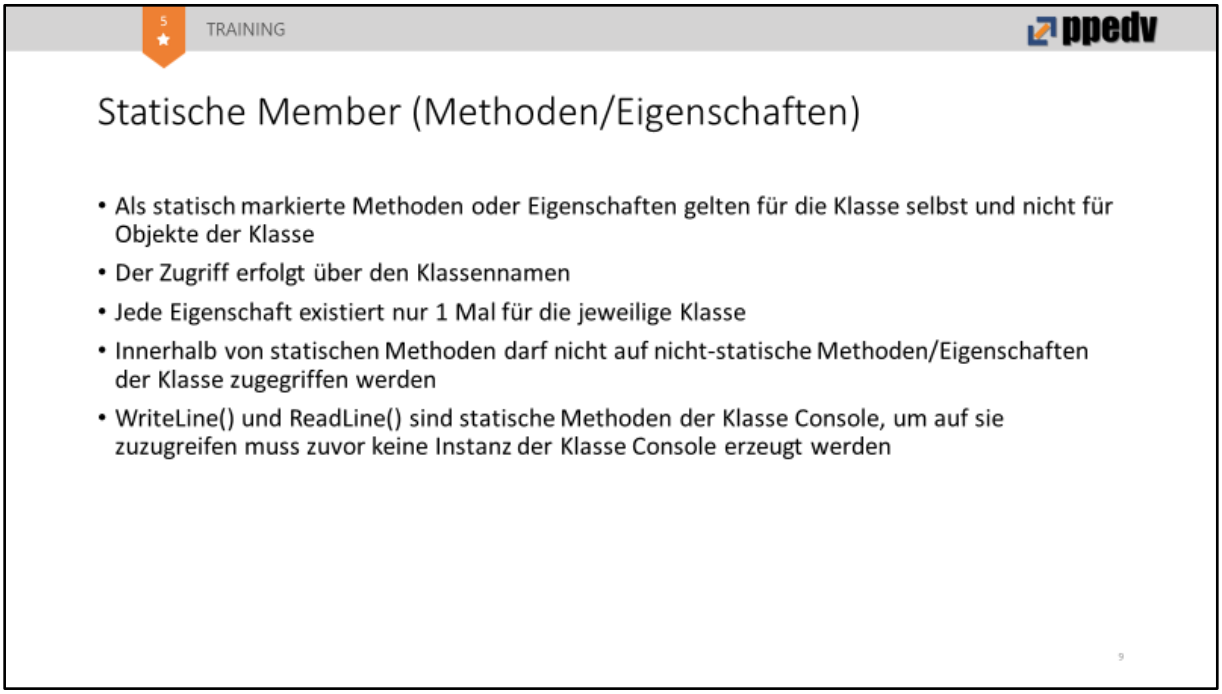
Konstruktoren sind spezielle Funktionen einer Klasse, welche für die Instanziierung (d.h. die Erstellung) von neuen Objekten dieser Klasse zuständig sind. Konstruktoren heißen dabei immer grundsätzlich wie die Klasse selbst und haben keine Rückgabetypangabe (nicht einmal void), da der Rückgabewert dieser Funktionen das neue Objekt selbst ist. In den Konstruktoren werden die Objekte in ihren Startzustand versetzt, was bedeutet, dass Felder mit Übergabeparametern initialisiert werden.

Jede Klasse kann eine beliebige Anzahl von Konstruktoren besitzen, wobei aber auch hier die Regeln für überladene Funktionen gelten.

Der Destruktor ist für die Löschung des Objekts aus dem Speicher zuständig und wird von der GarbageCollection aufgerufen, wenn das Objekt nirgendwo mehr referenziert ist.

Anders als in C++ können Objekte nicht mehr direkt gelöscht werden. Destruktoren müssen im Normalfall nicht geschrieben werden, da dies automatisch passiert und der Destruktor unsichtbar existiert. Möchte man im Zerstörungsfall noch andere Aktionen durchgeführt wissen, als die Löschung des Objekts, kann man einen Destruktor mit der Tilde (~) einleiten.

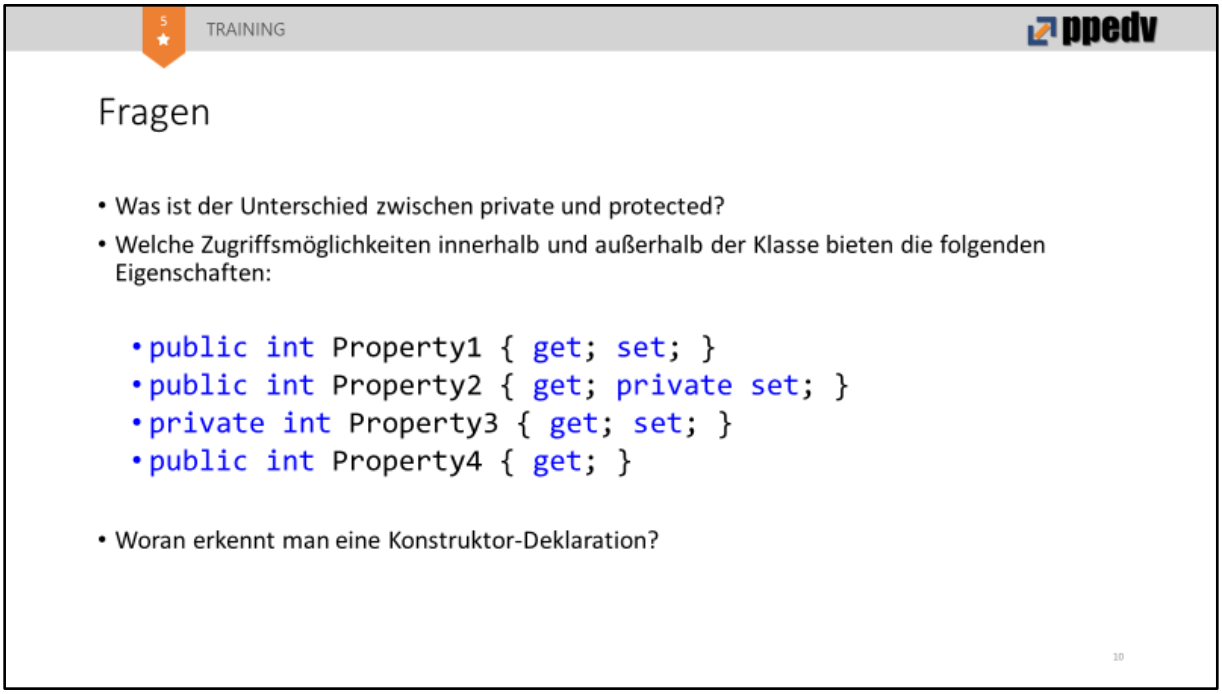
Statische Member (Methoden/Eigenschaften)



Variablen und Funktionen, welche mit dem Stichwort ‚static' markiert sind, sind Elemente, die nicht an Objekten der jeweiligen Klasse hängen, sondern global für die Klasse selbst gelten. Um sie zu verwenden muss kein Objekt erstellt werden.

Wenn eine ganze Klasse auf static gesetzt wird, kann von dieser Klasse keine Instanzen erstellt werden.

Fragen



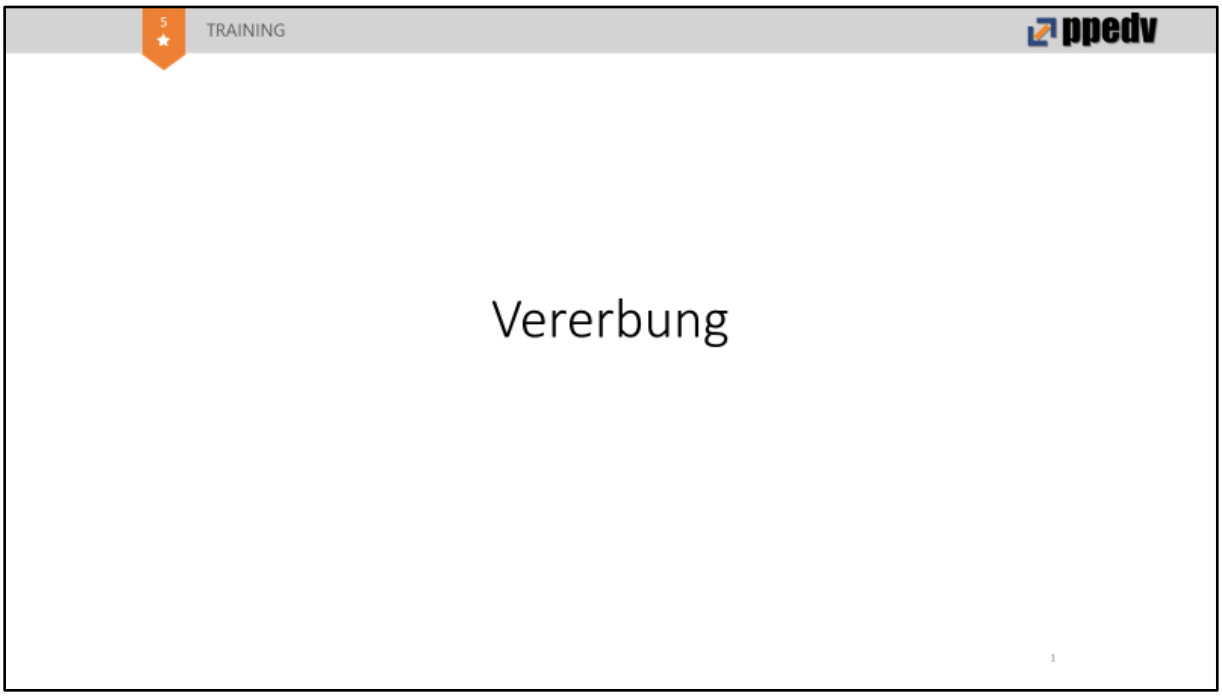
Antworten:

‚private' erlaubt den Zugriff auf ein Element nur aus der Klasse in der sich das Element befindet. ‚protected' erlaubt den Zugriff auch aus abgeleiteten Klassen.

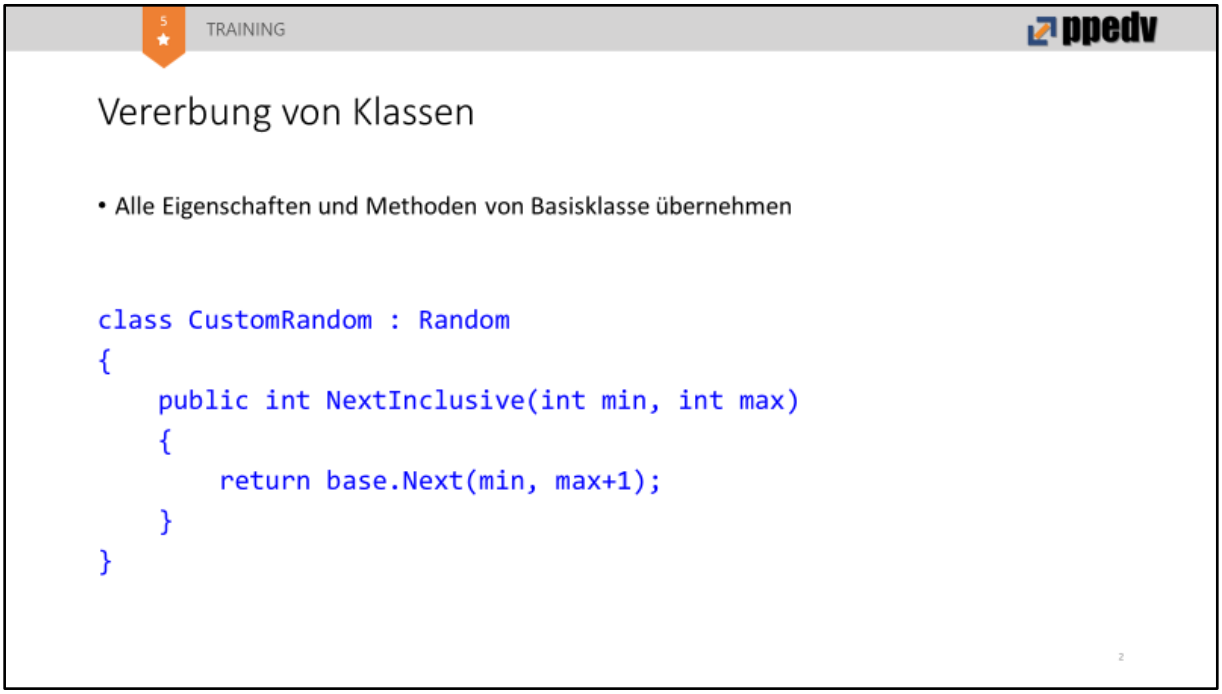
Property 1: Lese- und Schreibzugriff von überall
Property 2: Lesezugriff von überall, Schreibzugriff nur aus der eigenen Klasse
Property 3: Lese- und Schreibzugriff nur aus der eigenen Klasse
Property 4: Lesezugriff von überall, Schreibzugriff nicht möglich. Die Eigenschaft kann nur durch den Konstruktor gesetzt werden.

Konstruktoren heißen immer so wie die Klasse selbst und haben keinen Rückgabetypen (nicht einmal void).

Vererbung



Vererbung von Klassen

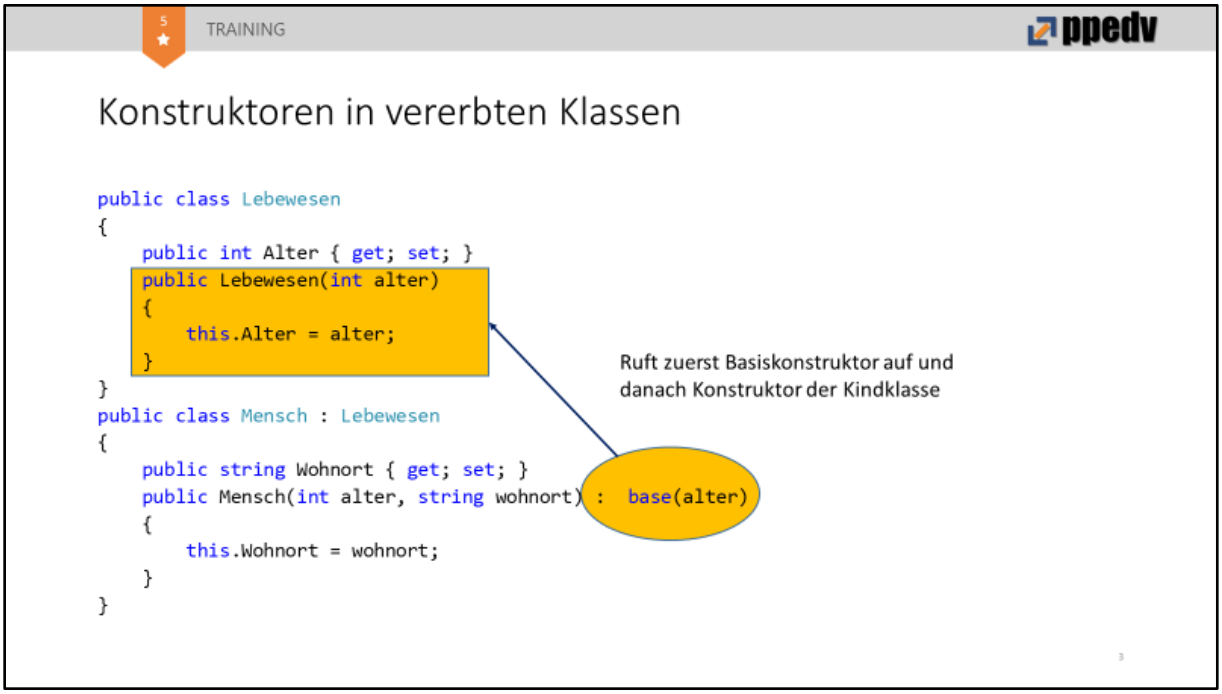


Mittels Vererbung kann eine Klasse alle Methoden und Eigenschaften einer anderen Klasse annehmen. Die abgeleitete Klasse kann dann zusätzlich weitere Eigenschaften und Methoden implementieren. Alle Objekte der abgeleiteten Klasse gelten zusätzlich auch als Objekte der Mutterklasse. Eine Vererbung wird in der Klassendeklaration durch einen Doppelpunkt (:) markiert. Jede Klasse kann dabei maximal von einer anderen Klasse erben.

Anders als in C++ ist keine Mehrfacherbung mehr möglich.

In diesem Beispiel übernimmt die Klasse ‚CustomRandom' sämtliche Elemente der Random-Klasse und implementiert zusätzlich die Methode NextInclusive().

Konstruktoren in vererbten Klassen

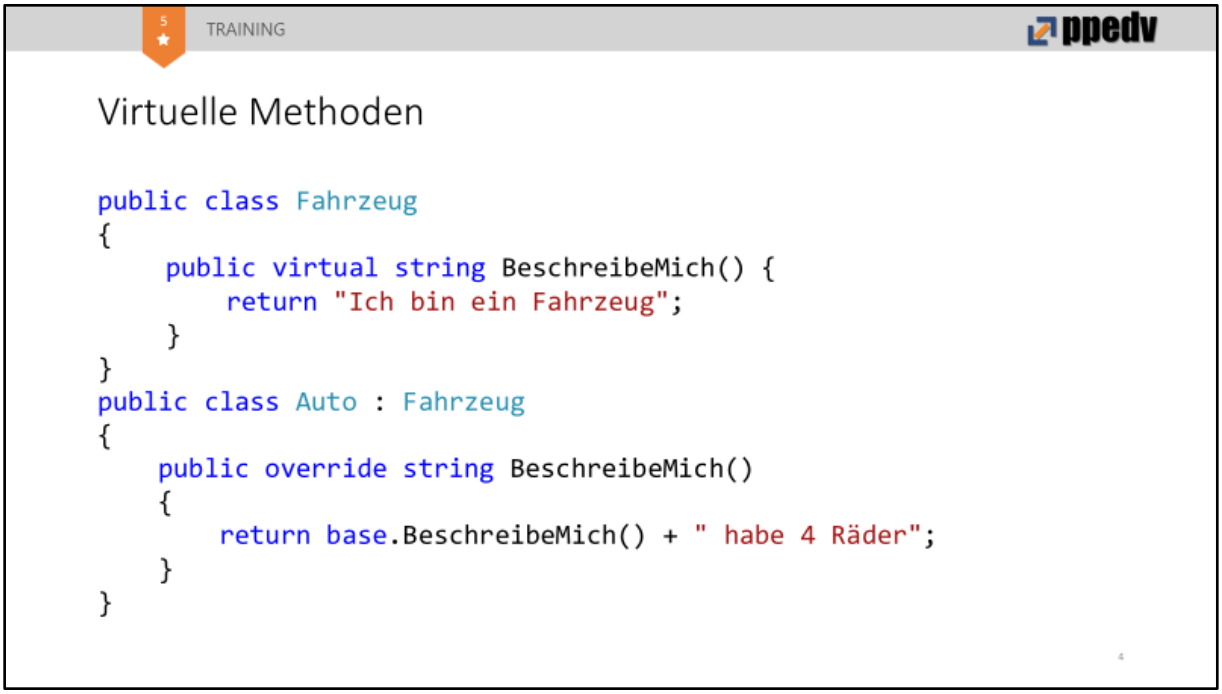


Bei der Instanziierung eines Objekts einer Kindklasse muss im Konstruktor zunächst der Konstruktor der Mutterklasse aufgerufen werden.

Dies geschieht mittels des Stichworts ‚base‘, welches per Doppelpunkt (:) dem Methodenkopf des Kindklassenkonstruktors nachgestellt wird und dem die entsprechenden Übergabeparameter mitgeben werden.

Jede Kindklasse benötigt mindestens einen eigenen Konstruktor, welcher immer einen gültigen Konstruktor der Mutterklasse aufrufen muss.

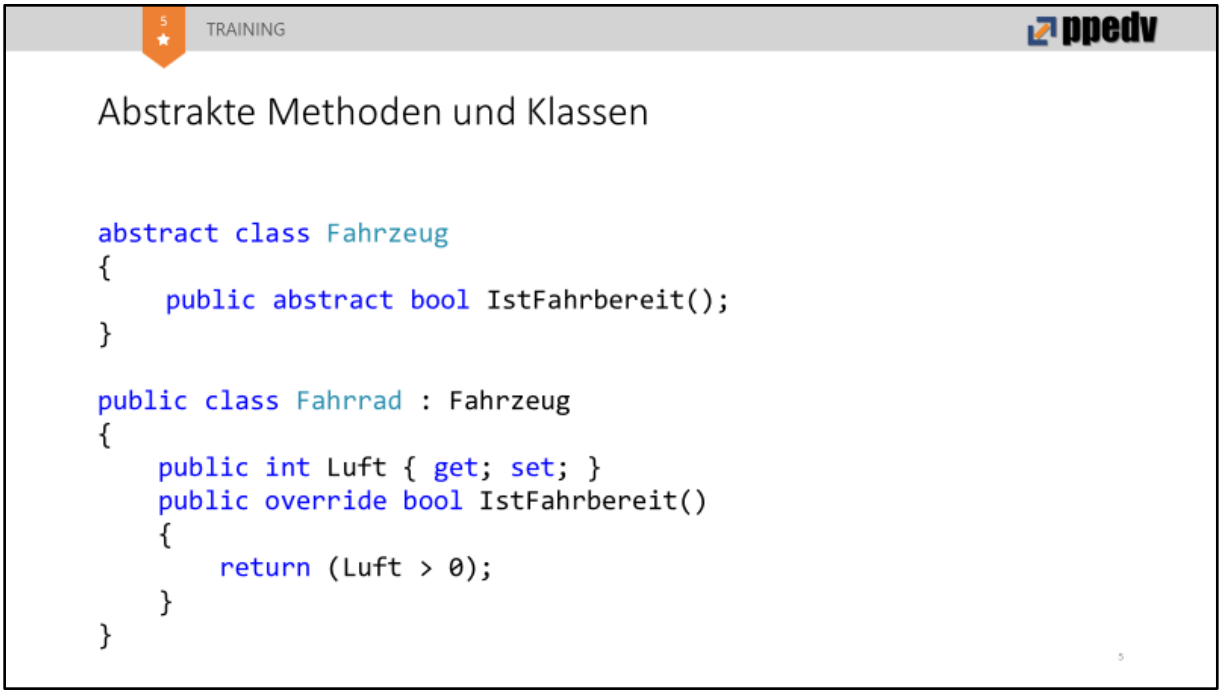
Virtuelle Methoden



Wenn in der Mutterklasse Methoden durch den Modifier ‚virtual' markiert werden, heißt diese Methode virtuell. Diese können in den Kindklassen mittels des Modifiers ‚override' überschrieben werden. Objekte der Kindklasse werden bei dem entsprechenden Methodenaufruf immer die in ihrer Klasse implementierte Methode ausführen.

Mittels des ‚base'-Stichworts kann aus den Kindklassen heraus auf die Implementierung der überschriebenen Methode der Mutterklasse zugegriffen werden.

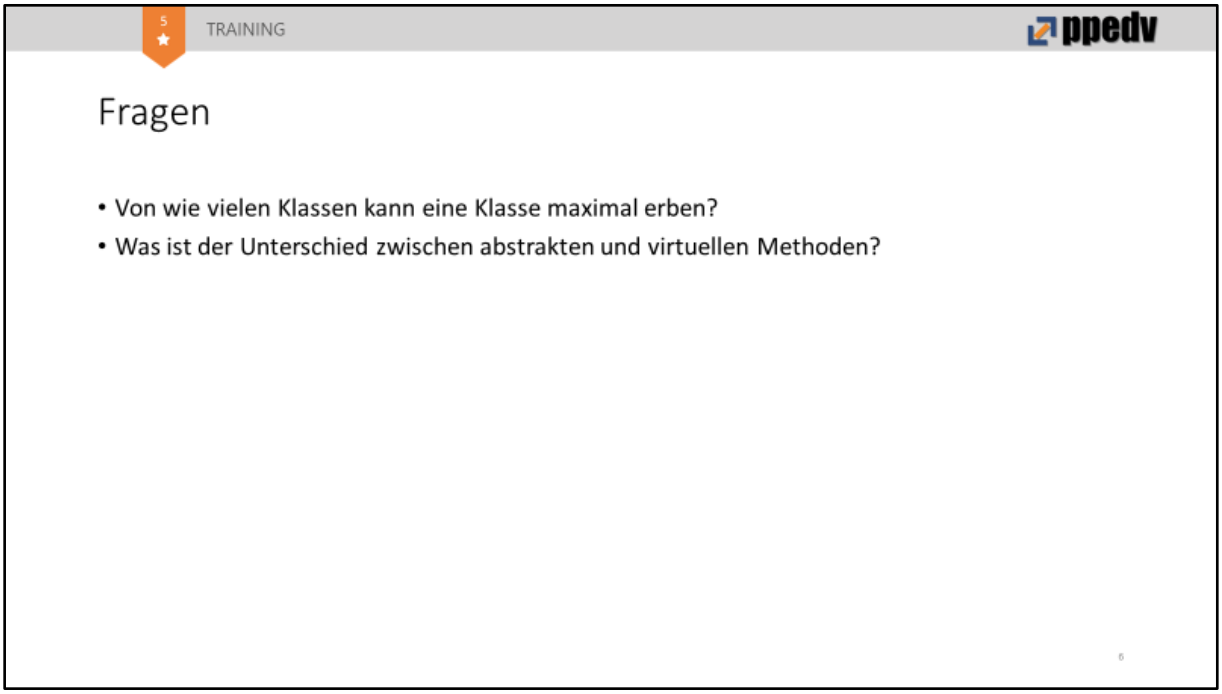
Abstrakte Methoden und Klassen



Wird eine Methode mittels des Modifiers ‚abstract' als abstrakt markiert, kann in ihr nur eine Signatur, aber kein Körper definiert werden. Kindklassen werden gezwungen die Methode zu implementieren und einen Körper zu definieren.

Wenn eine Klasse eine abstrakte Methode enthält, muss die ganze Klasse auf ‚abstract' gesetzt werden, was zu Folge hat, dass von ihr keine Objekte mehr instanziiert werden können. Diese Klasse ist nun eine reine Vorlage für Kindklassen.

Fragen

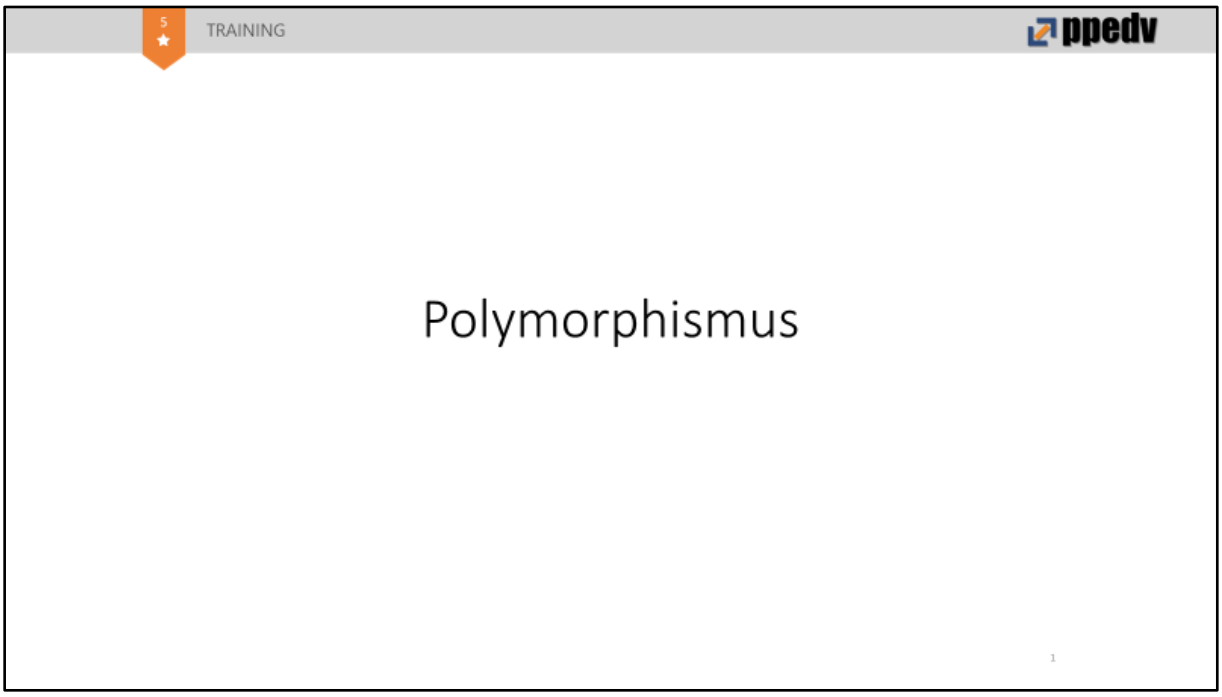


Antworten:

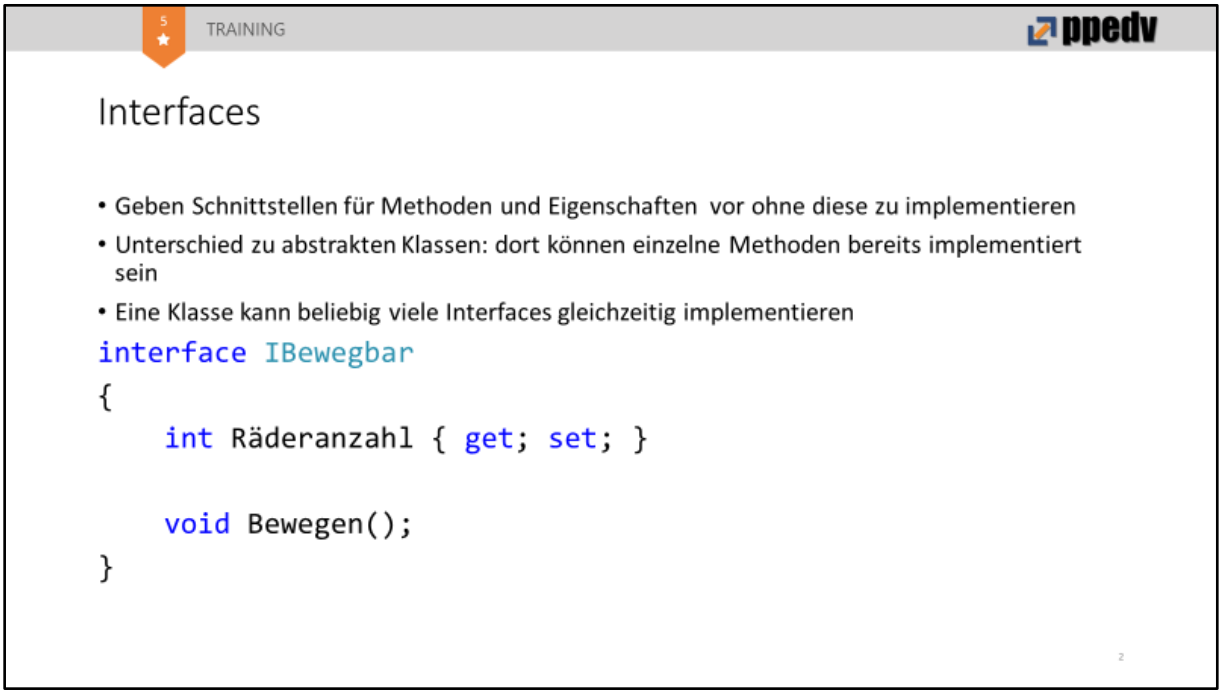
Jede Klasse kann maximal von einer anderen Klasse erben.

Während virtuelle Methoden einen Körper definieren und nur die Überschreibung durch Kindklassen erlauben, zwingen abstrakte Methoden, welche grundsätzlich keinen Körper definieren, die Kindklassen zu einer Implementierung.

Polymorphismus



Interfaces

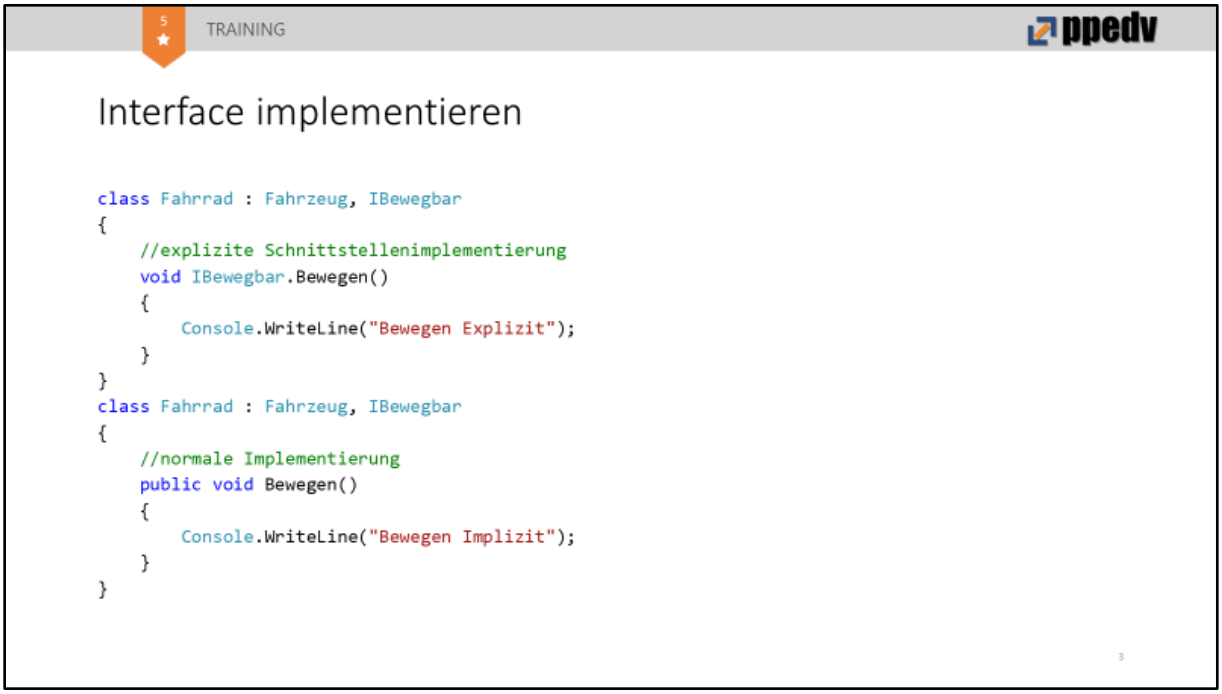


Interfaces (Schnittstellen) definieren Eigenschaften und Methoden ohne sie zu implementieren.

Klassen, welche das Interface einbinden, werden gezwungen die Eigenschaften und Methoden einzufügen und zu implementieren.

Eine Klasse kann beliebig viele Interfaces implementieren und Objekte dieser Klasse gelten zusätzlich auch als Objekte der Interfaces.

Interface implementieren

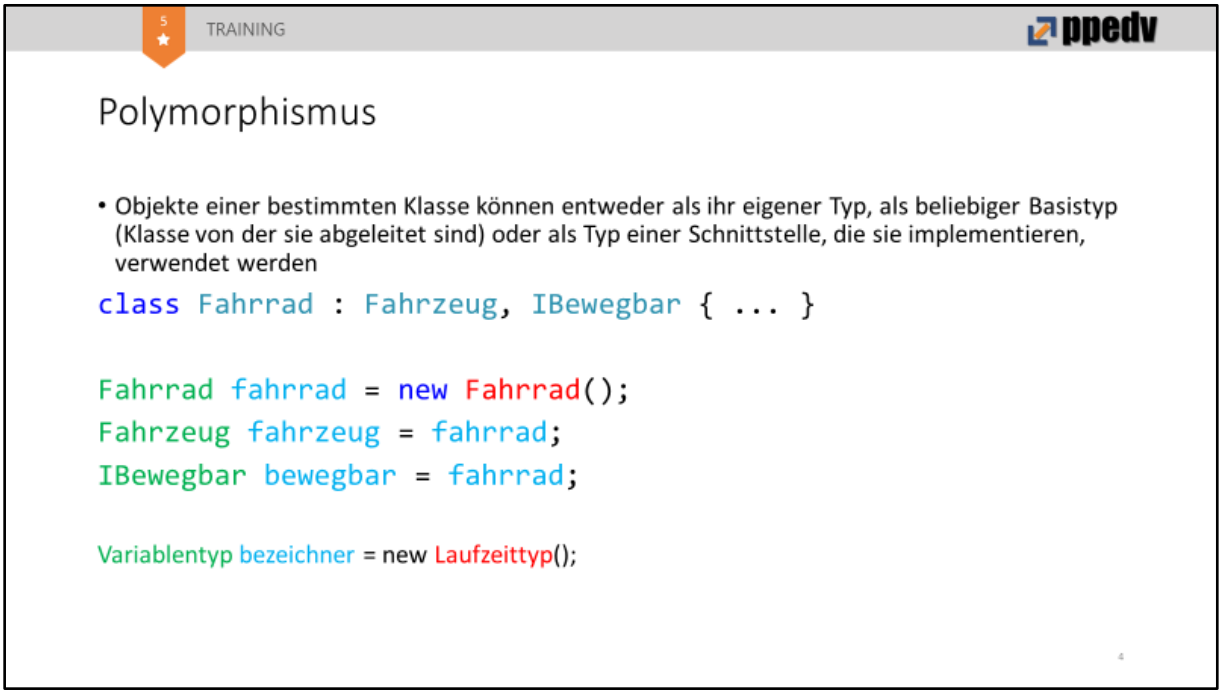


Die Implementierung von Interfaces in Klassen erfolgt über die gleiche Notation wie die Vererbung (Doppelpunkt in der Klassensignatur).

Es besteht die Möglichkeit die Methoden eines Interfaces auf zwei verschiedene Arten zu implementieren: Wird eine Methode ‚explizit' implementiert kann diese nur aufgerufen werden, wenn das Objekt zu einem Objekt vom Typ des Interfaces gecastet wird. Bei der ‚impliziten' Implementierung ist der Aufruf der Methode auch über das natürliche Objekt möglich.

Durch explizite Implementierung kann auch für den Fall eindeutig implementiert werden, dass gleichnamige Methoden aus unterschiedlichen Interfaces gleichzeitig implementiert werden sollen. Zudem ist es so möglich, mittels des gleichen Methodennamens zwei verschieden agierende Methoden zu implementieren.

Polymorphismus

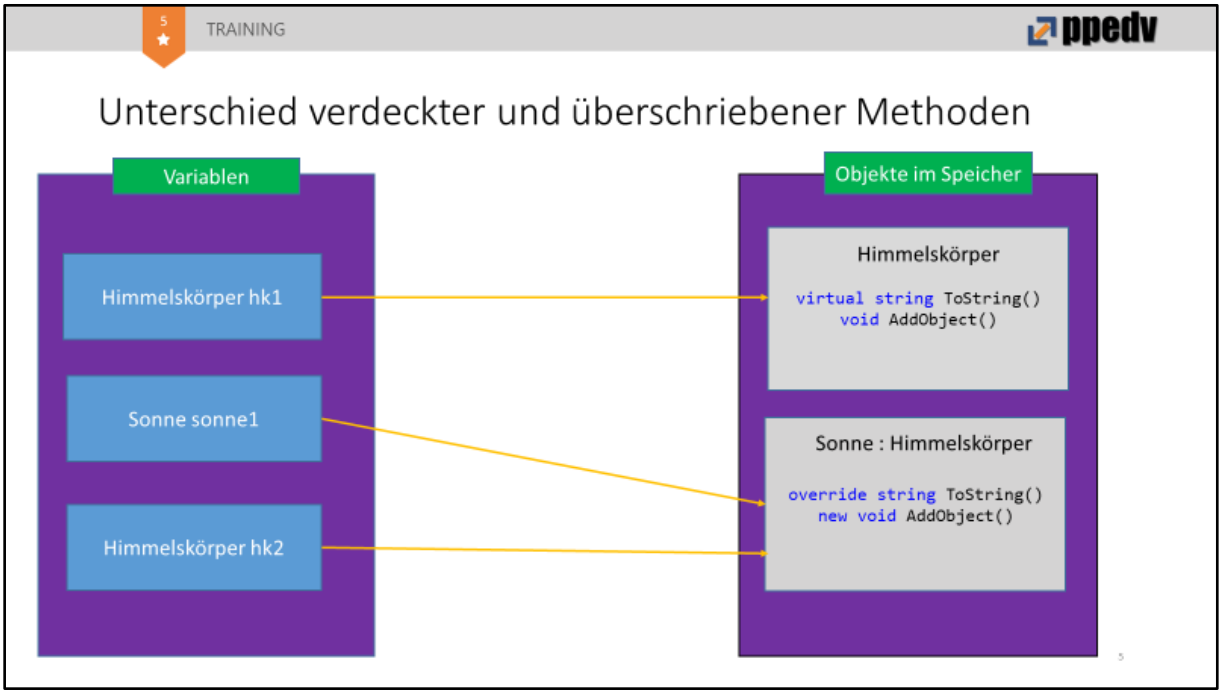


Unter Polymorphismus versteht man das Konzept, in dem ein Objekt grundsätzlich nicht nur als Objekt seiner Klasse, sondern auch als Objekt aller Mutterklassen und implementierten Interfaces betrachtet werden kann.

Dies bedeutet, dass jedes Objekt auch in Variablen vom Typ seiner Mutterklassen oder Interfaces gelegt werden kann. Der Variablentyp legt dabei fest, auf welche Methoden und Eigenschaften die erzeugten Laufzeittypen zugreifen können. So kennt das Objekt immer nur die Methoden und Eigenschaften, die der aktuelle Variablentyp besitzt. Befindet sich zum Beispiel ein Objekt in einer Variablen seiner Mutterklasse, so kennt das Objekt keine erst in der Kindklasse hinzugefügten Methoden.

Eine Ausnahme der zuletzt erwähnten Regel bilden die per ‚override' überschriebenen virtuellen Methoden: Hier werden nach wie vor die in der Kindklasse definierten Methoden ausgeführt.

Unterschied verdeckter und überschriebener Methoden



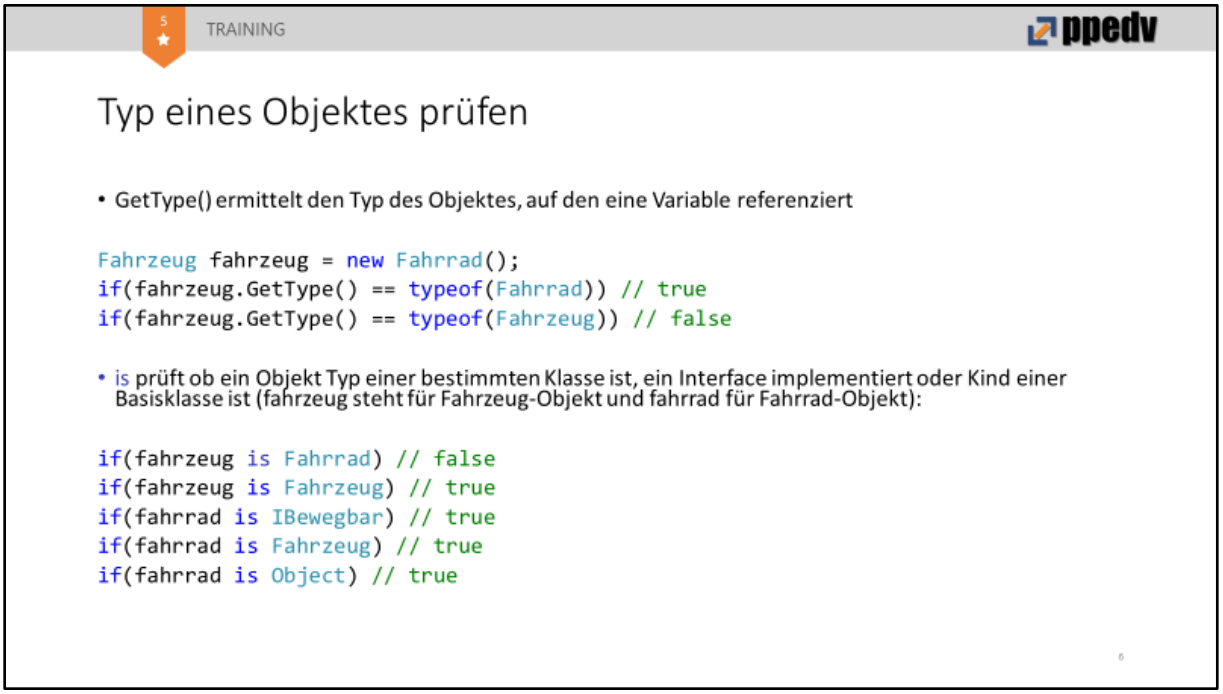
Neben dem Überschreiben von Methoden, bei welchem unabhängig vom Variablentyp immer die überschriebene Methode ausgeführt wird, können Methoden auch überladen werden. Dies geschieht, wenn man eine Methode in einer Kindklasse mit dem gleichen Namen und den gleichen Übergabeparametern wie eine Methode der Mutterklasse definiert.

Der Compiler akzeptiert diese Notation, empfiehlt jedoch die Verwendung des Stichworts ‚new' in der Methodensignatur zur besseren Erkennbarkeit der überlagerten Funktion.

Bei einer überlagerten Methode wird ein Objekt wieder auf die Implementierung der Variablen, in welcher sich das Objekt befindet, zurückgreifen. D.h. befindet sich das Objekt in einer Variablen der Mutterklasse, wird die Methode der Mutterklasse ausgeführt. In einer Variablen der Kindklasse wird deren Implementierung ausgeführt.

Im obigen Beispiel wird das Sonnen-Objekt in der Variablen sonne1 sowohl die überschriebene ToString()- als auch die überlagerte AddObject()-Methode der Sonnen-Klasse ausführen - das Sonnen-Objekt in der Himmelskörper-Variablen.

Typ eines Objektes prüfen

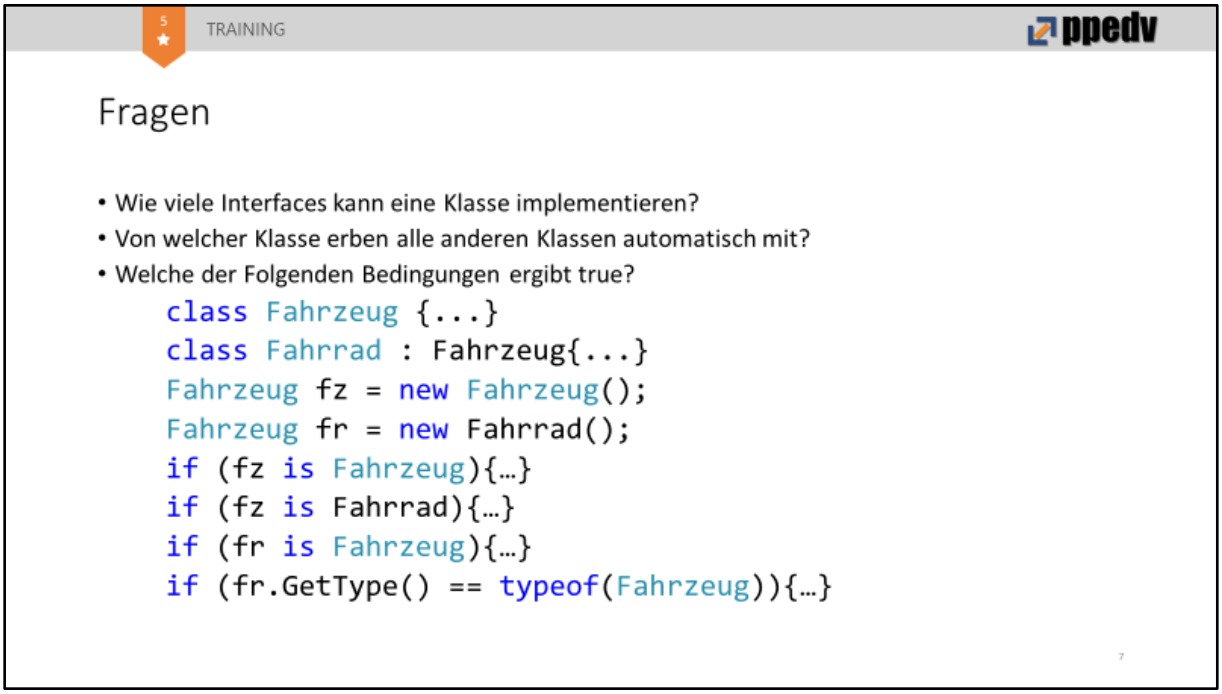


Da in einer Variable nicht nur Objekte des Variablentyps, sondern auch Objekte der Kindklassen stecken können, ist es notwendig, dass man den Datentyp von Objekten erfragen kann. Hierfür gibt es, unter anderem, die folgenden zwei Möglichkeiten:

Mittels der Methode GetType(), welche von der Object-Klasse an alle anderen Klassen vererbt wird, kann der genaue Typ eines Objekts erfragt werden. Nun kann ein Vergleich mit dem ‚typeof()'-Stichwort Aufschluss darüber geben, ob ein Objekt einen bestimmten Datentyp hat.

Mittels des ‚is'-Stichworts kann ein Objekt auf sämtliche Datentypen seiner Erbungslinie sowie auf implementierte Interfaces geprüft werden. Kurz gesagt wird dieses Stichwort immer ein ‚true' zurückgeben, wenn man ein Objekt in den erfragten Datentypen casten bzw. wenn man es in eine Variable dieses Typs legen kann.

Fragen



Antworten:

Jede Klasse kann eine beliebige Anzahl von Interfaces implementieren.

Alle Klassen erben von der Object-Klasse, welche gewisse Basis-Methoden, wie z.B. die ToString()- und die GetType()-Methoden.

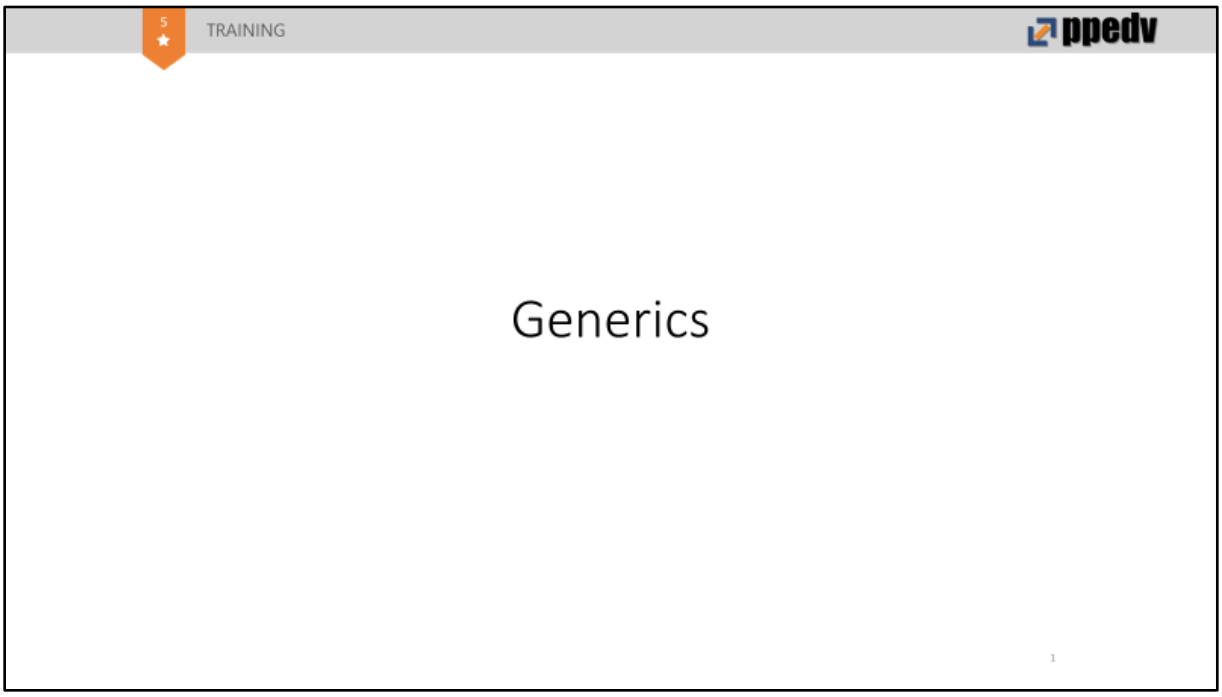
(fz ist Fahrzeug) -> true

(fz is Fahrrad) -> false

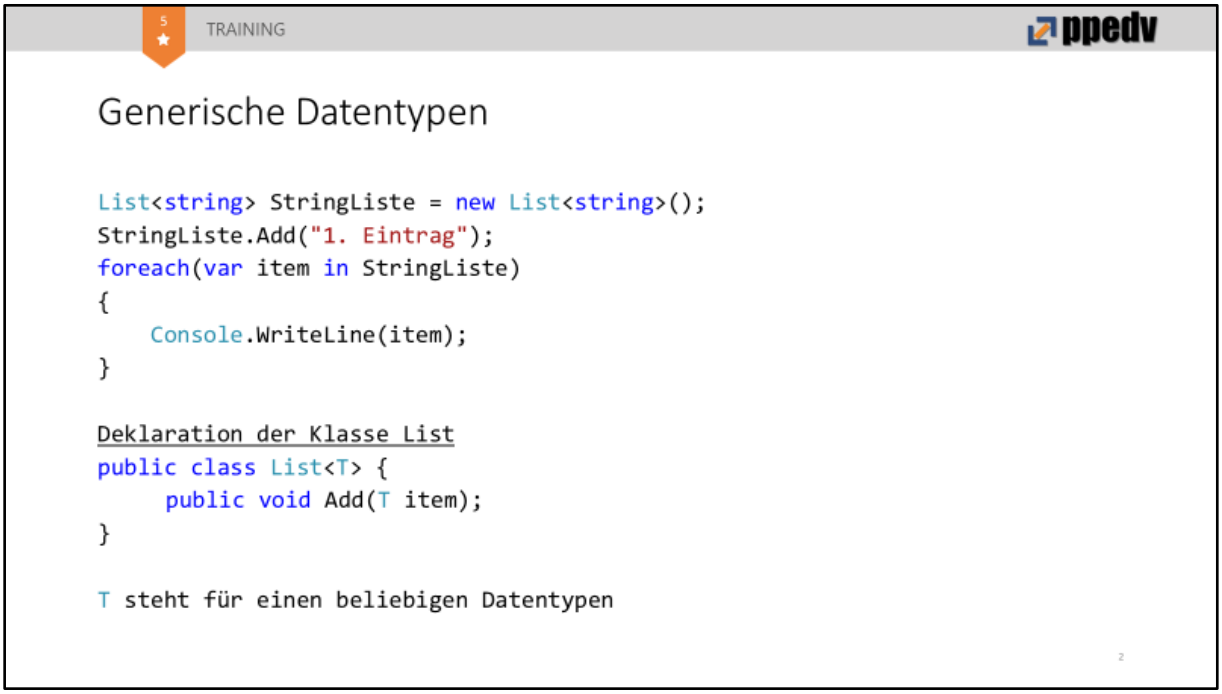
(fr is Fahrzeug) -> true

(fr.GetType() == typeof(Fahrzeug) -> false

Generics



Generische Datentypen

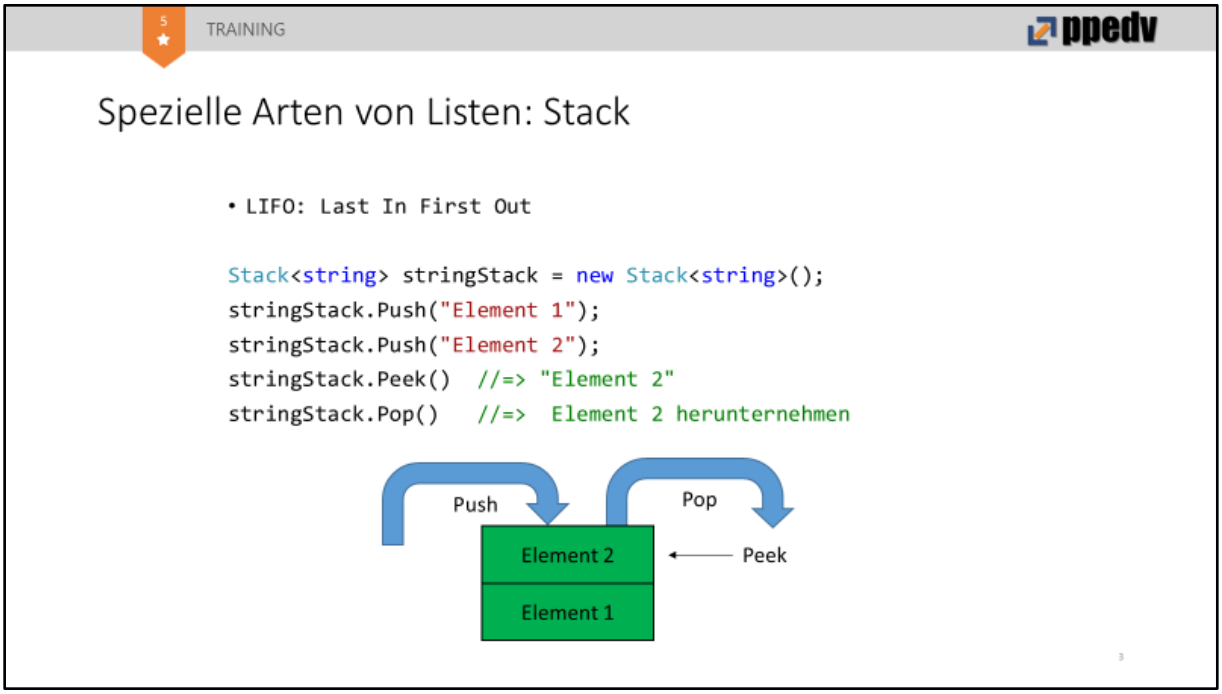


Generische Datentypen ermöglichen die variable Gestaltung von Membern (Methoden/Eigenschaften). Objekte von generischen Klassen können stark typisiert werden, d.h. sie können sich variabel auf einen Datentypen spezialisieren, welchen sie unterstützen. Der Platzhalter ‚T' steht in diesem Fall für den vom Nutzer der Klasse festgelegten Datentyp. Innerhalb der Klasse können dann Objekte des Datentyps T erzeugt werden.

Oftmals sind die generischen Datentypen Collections (Sammlungen), welche bestimmte Verhalten besitzen und über die spitzen Klammern (<>) bei Deklaration auf einen Datentyp, welchen sie sammeln können, festgelegt werden.

Im obigen Beispiel wird die Collection ‚List' vorgestellt, welche eine einfache Liste mit Elementen des Datentyps T beinhalten kann. Die Anzahl der Elemente kann, anders als im Array, zu Laufzeit variabel verändert werden. Zudem bietet List diverse Funktionen, z.B. zum Sortieren und Durchsuchen, an, welche Arrays nicht zur Verfügung stehen. Über die Add()-Methode werden der Liste neue Elemente hinzugefügt.

Spezielle Arten von Listen: Stack

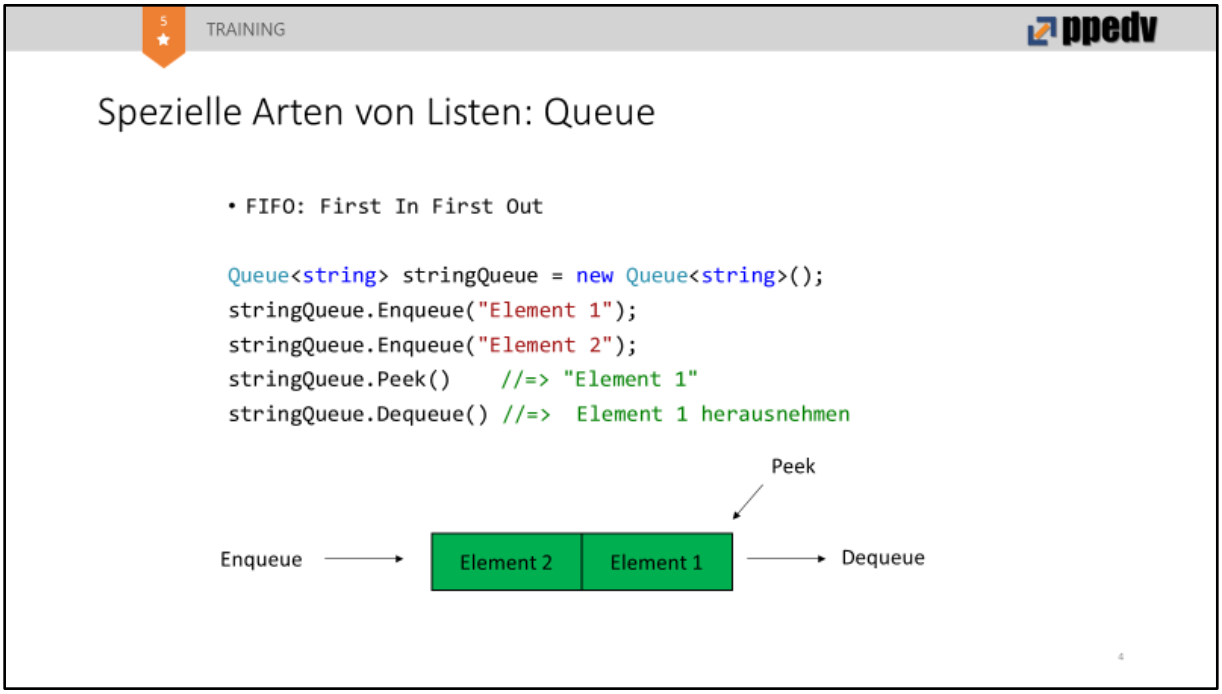


Die Klasse Stack implementiert eine Collection, welche nach dem Prinzip ‚Last In – First Out' funktioniert.

Dies bedeutet, dass das zuletzt hinzugefügte Element ‚an der Spitze' des Stacks liegt. Neue Elemente werden mit der Funktion Push() hinzugefügt. In einem Stack ist immer nur das oberste Element sichtbar, alle anderen Elemente befinden sich unsichtbar ‚unter' dem ersten Element.

Mittels der Funktion Peek() kann das oberste Element betrachtet werden, ohne das es vom Stack entfernt wird, während mit der Funktion Pop() das oberste Element betrachtet und von Stack entfernt wird.

Spezielle Arten von Listen: Queue



Die Klasse Queue funktioniert ähnlich wie der Stack, verwendet aber das Prinzip ‚First In – First Out'.

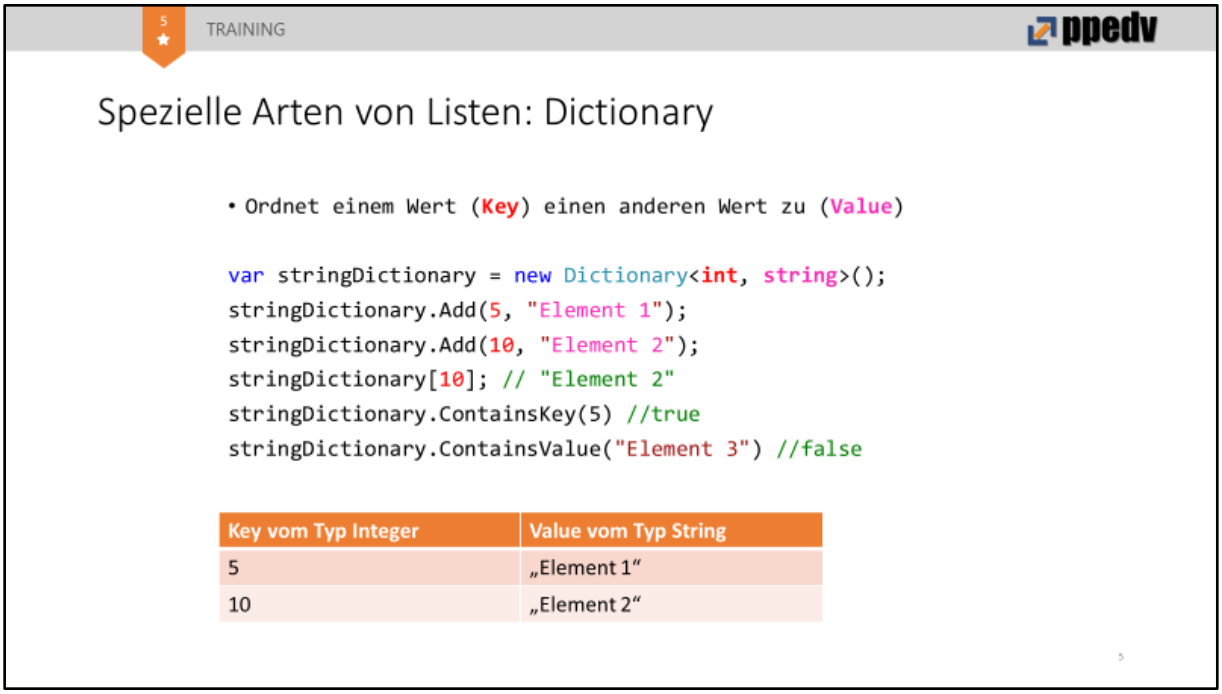
Dies bedeutet, dass per Enqueue()-Funktion hinzugefügte Elemente an das ‚Ende' der Queue angehängt wird.

Auch in einer Queue sind alle Elemente, mit Ausnahme des Ersten, unsichtbar.

Auch hier funktioniert die Peek()-Funktion, welche das selbe Verhalten wie im Stack besitzt.

Mittels der Dequeue()-Funktion wird das erste Objekt betrachtet und aus der Queue entfernt.

Spezielle Arten von Listen: Dictionary



Die Klasse Dictionary<T1,T2> weist ein ähnliches Verhalten wie List auf, speichert in seinen Einträgen allerdings immer ein Paar von zwei Objekten. Diese beiden Objekte können verschiedenen Datentyps sein, allerdings muss jedes Paar aus den gleichen Datentypen bestehen. Wie auch bei den anderen generischen Datentypen werden die zu beinhaltenden Typen bei der Deklaration in die spitzen Klammern geschrieben.

Der erste Datentyp beschreibt den sogenannten ‚Key', über welchen ein Zugriff auf die Paare erfolgt, der zweite Datentyp beschreibt den ‚Value' des Paars.

Der Zugriff erfolgt, wie in Arrays, über eine Positionsangabe in eckigen Klammern, welche auf den Variablennamen folgt. Die Positionsangabe ergibt sich allerdings nicht aus einem Index, sondern aus dem entsprechenden ‚Key', welcher in die Klammern eingefügt wird. Die Ausgabe ist dann der ‚Value'. Damit dieser Zugriff eindeutig ist, darf kein Key mehr als einmal im Dictionary vorhanden sein.

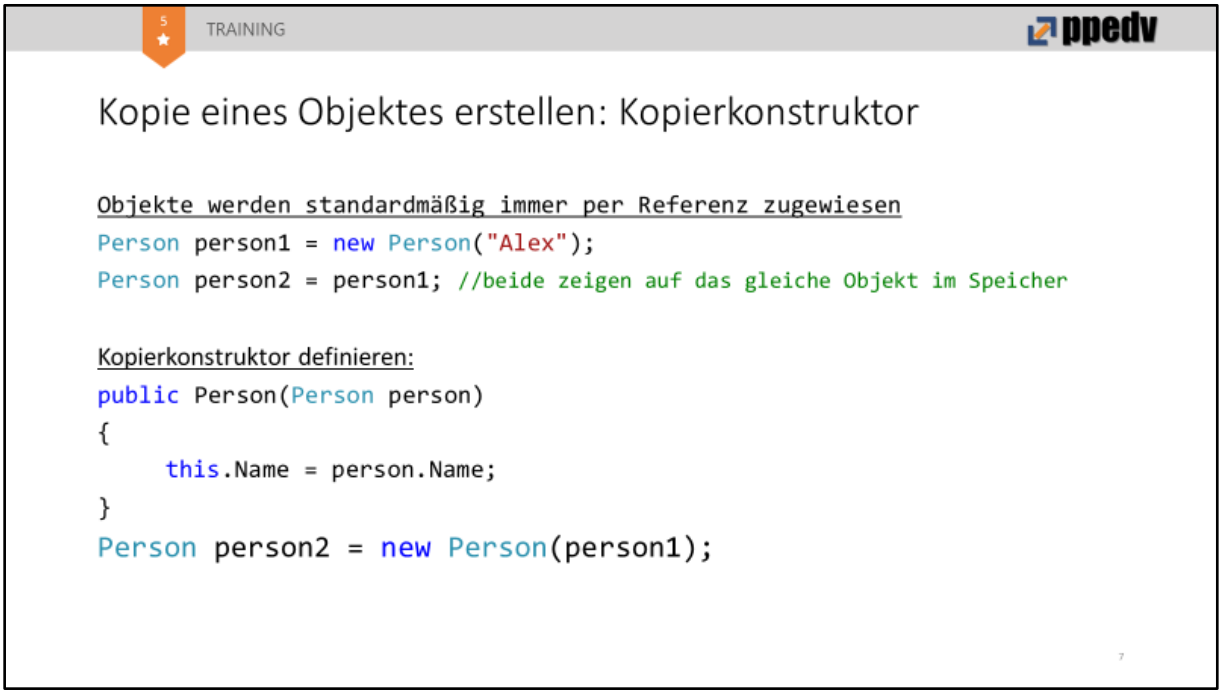
Dementsprechend ist das Dictionary eine zweispaltige Tabelle.

Spezielle Arten von Listen: Hashtable

Auch der Hashtable speichert Paare, welche aus zwei Werten bestehen. Im Unterschied zum Dictionary können hier in jeden Paar zwei beliebige Datentypen gespeichert werden. Auch hier erfolgt der Zugriff über den Key eines Paares, welcher auch hier nur einmal im Hashtable vorhanden sein darf.

Einträge der Hashtable werden intern als Nummer (HashCode) gespeichert. Der HashCode eines Items ist über GetHashCode() abrufbar.

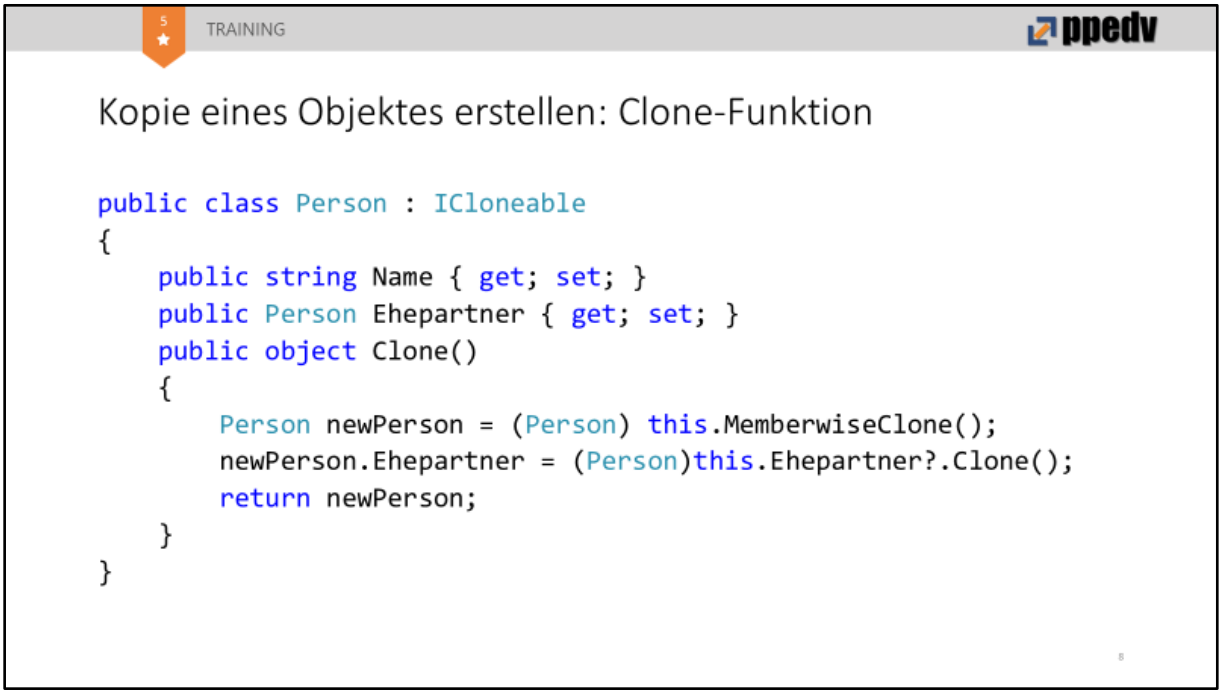
Kopie eines Objektes erstellen: Kopierkonstruktor



Möchte man von einem Objekt eine Kopie erstellen, reicht es nicht ein Objekt einer weiteren Variablen zuzuweisen. Anders als bei den Basisdatentypen werden komplexe Objekte nicht als Werte (d.h. als Kopien) übergeben, sondern es wird eine Referenz auf das bestehende Objekt übergeben.

Um ein Objekt zu kopieren, kann man einen Kopierkonstruktor erstellen, der als Parameter das zu kopierende Objekt erhält. Im Körper kann man dann alle Werttyp-Felder kopieren. Zu beachten ist, dass wenn die Objekte komplexe Datentypen als Felder haben auch hier keine direkte Kopie möglich ist.

Kopie eines Objektes erstellen: Clone-Funktion

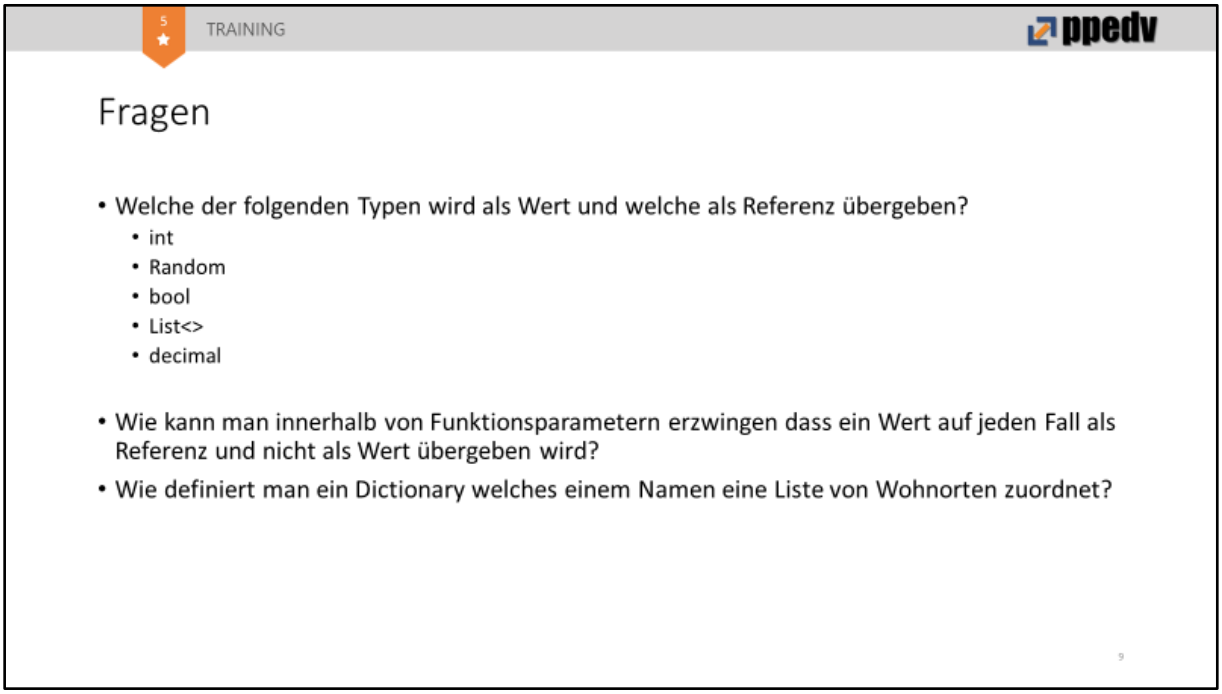


Eine weitere Möglichkeit ein Objekt zu kopieren ist die Implementierung des ICloneable-Interfaces. Dieses Interface verlangt die Implementierung einer Clone()-Funktion, welche ein neues Objekt zurückgeben soll.

Mittels der von der object-Klasse vererbten Funktion MemberwiseClone() kann eine sogenannte flache Kopie eines Objekts erzeugt werden. Dies beinhaltet Kopien aller Wertetypen, allerdings keine Referenztypen.

Diese müssen explizit einzeln z.B. mittels Clone()-Funktion kopiert werden. Ohne diese zusätzliche Zuweisung würden sowohl das Originalobjekt als auch die Kopie auf dasselbe Memberobjekt verweisen.

Fragen



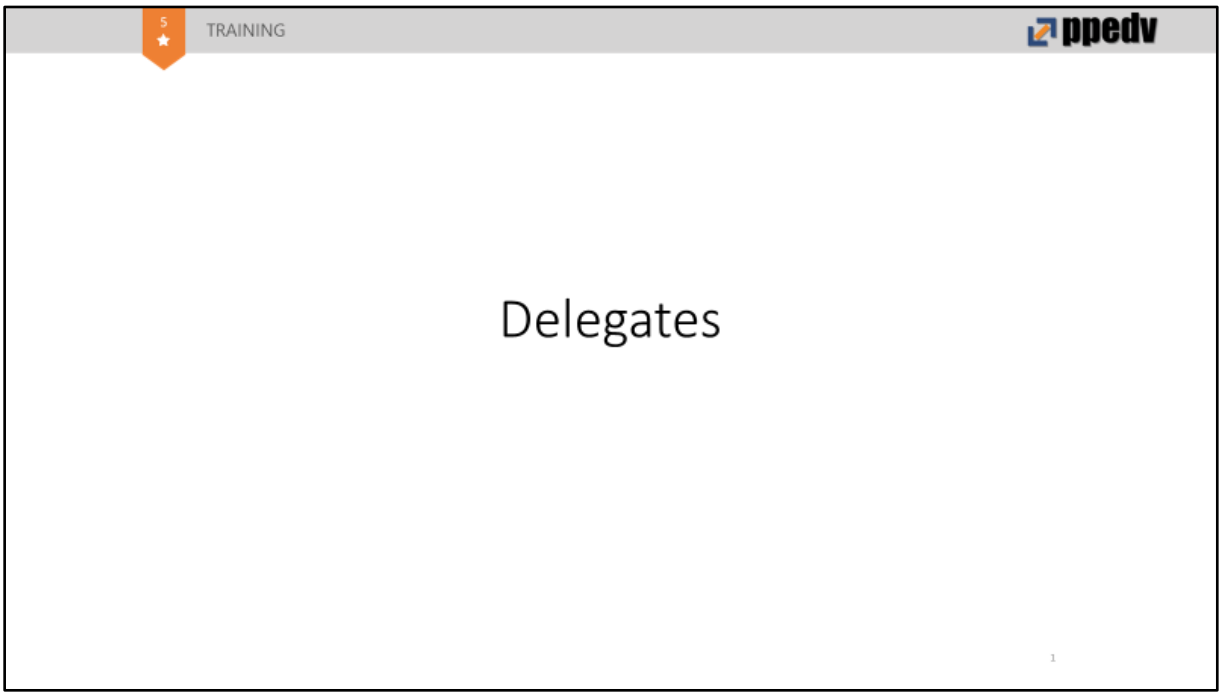
Antworten:

Werttypen:	int, bool, decimal
Referenztypen:	Random, List<>

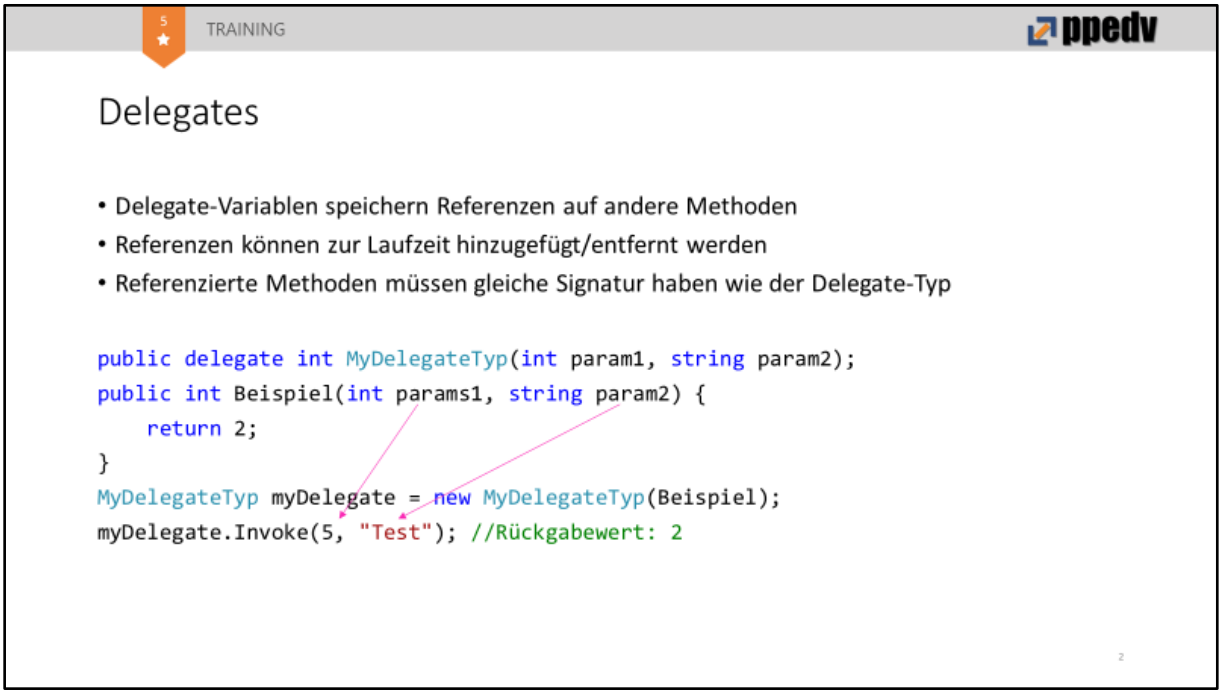
Mittels des Stichworts ‚ref' in den Methodenparametern kann ein Werttyp als Referenz an eine Methode übergeben werden.

```
Dictionary<string, List> dict = new Dictionary<string, List>;
```

Delegates



Delegates



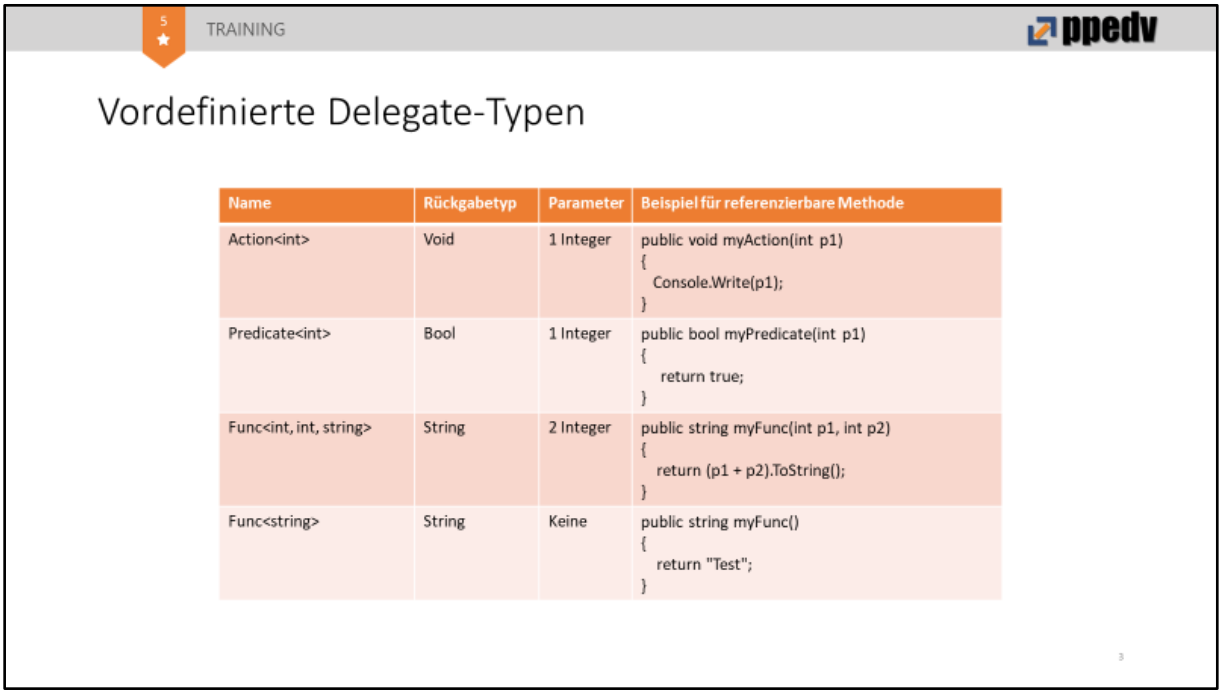
Delegates (Delegaten) können als Variablen verstanden werden, welche jeweils bestimmte Arten von Funktionen ‚beinhalten' können.

Um einen Delegate verwenden zu können, muss dieser erst definiert werden. Diese Definition ähnelt einer Methodensignatur, d.h. sie besitzt einen Rückgabetypen sowie gegebenenfalls Übergabeparameter, welche zusätzlich mit dem Stichwort ‚delegate' versehen ist. Variablen dieses Typs können nun alle Methoden, welche exakt diesen Rückgabetyp und diese Übergabeparameter besitzen, zugewiesen werden.

Jedem Delegate können über den +=-Operator beliebig viele Methoden mit gleicher Signatur zugewiesen werden. Mittels -= können Methoden wieder entfernt werden. Über die GetInvocationList()-Funktion kann eine Liste der Methoden, welche momentan einem Delegate zugwiesen sind, abgerufen werden.

Wenn die Delegate-Variable aufgerufen wird, werden die beinhaltenden Methoden in der Zuweisungsreihenfolge mit den der Variablen übergebenen Parametern ausgeführt. Wenn die Methoden Rückgabewerte besitzen, wird nur der Rückgabewert der zuletzt ausgeführten/zugewiesenen Methode an den Aufrufer zurückgegeben. Alle anderen Rückgabewerte verfallen.

Vordefinierte Delegate-Typen

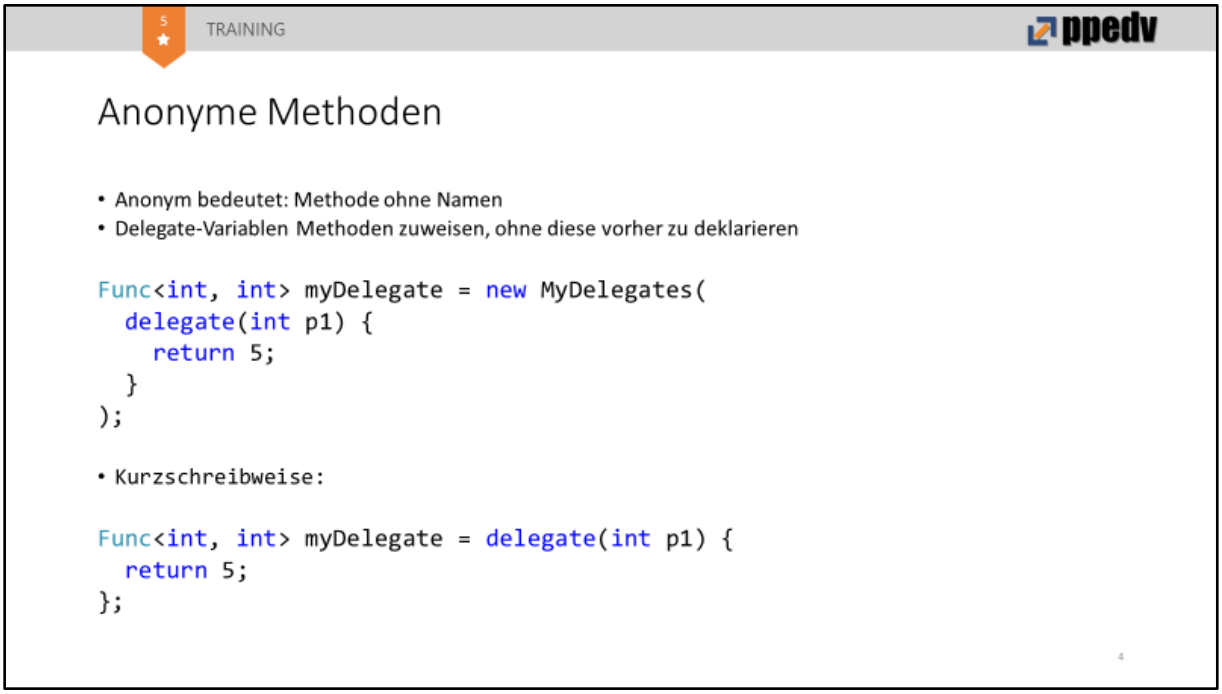


Vom .NET-Framework werden vordefinierte Delegat-Typen zur Verfügung gestellt, welche die meisten Anwendungsfälle abdecken. Diese Typen sind generische Datentypen, welchen die Übergabeparameter/Rückgabewerte in den spitzen Klammern übergeben werden.

Action<> stellt Delegates ohne Rückgabetypen (mit void) zur Verfügung. Es können bis zu 16 Parametertypen definiert werden. Predicate<> stellt Delegates mit dem Rückgabetyp bool zur Verfügung. Es kann ein Übergabeparameter definiert werden. Func<> stellt Delegates mit einem selbst zu definierenden Rückgabetyp zur Verfügung. Hier können bis zu 15 Parameter definiert werden, während der letzte der angegebenen Datentypen immer den Rückgabetyp bestimmt.

Dieser darf nicht void sein.

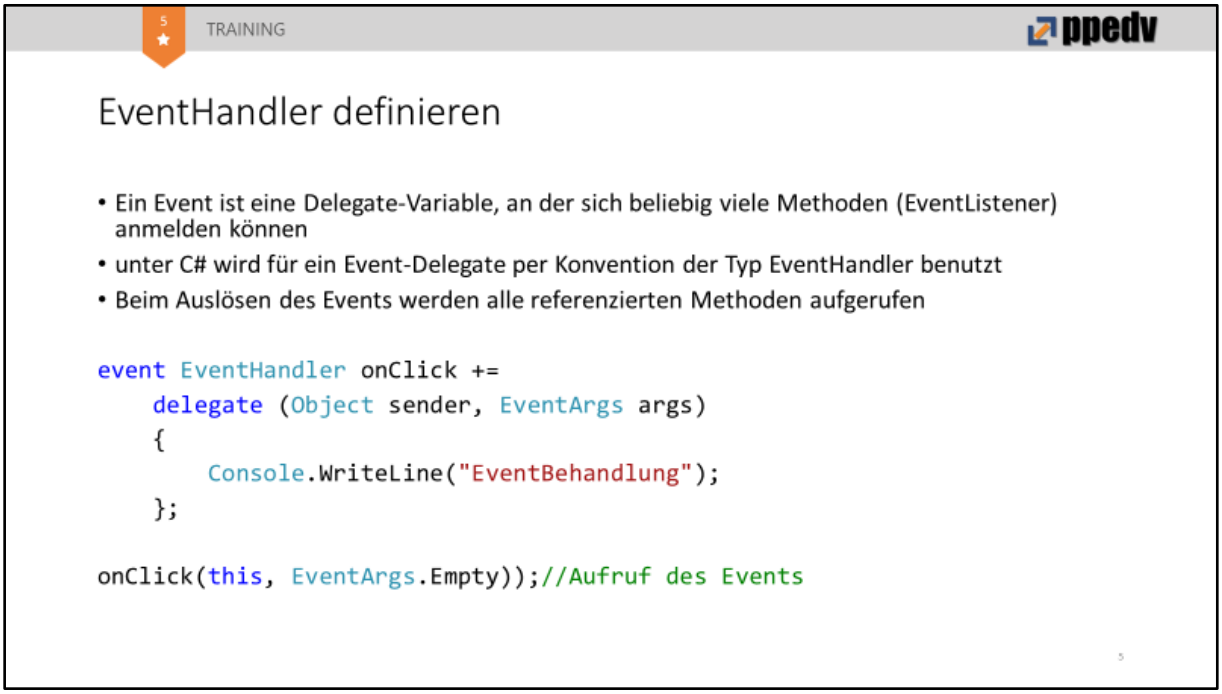
Anonyme Methoden



Anonyme Methoden sind Methoden, welche nicht fest im Code definiert sind. Sie werden nur zu Definition von Methoden verwendet, welche man in Delegate legen möchte. Dabei wird dem ‚delegate'-Stichwort ein Übergabeparameter in runden Klammern nachgestellt, worauf der in geschweifte Klammern gestellte Mehodenkörper folgt.

HINWEIS: Mittels den unten beschriebenen Lambda-Ausdrücken (vgl. ‚Fortgeschrittene Themen') können anonyme Methoden noch komprimierter formuliert werden.

EventHandler definieren

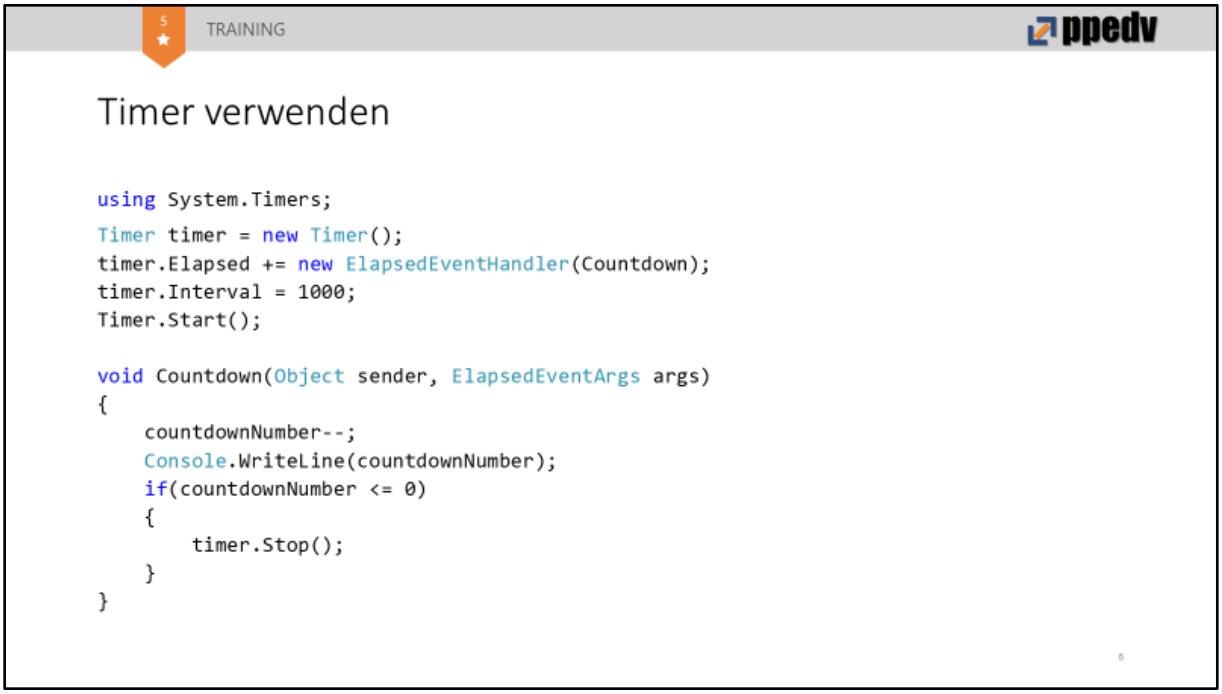


Mittels des Stichworts ‚event' wird ein spezieller Delegate, ein Event, definiert.

Diesem speziellen Delegate, welches meist vom Typ ‚EventHandler' ist, können von außerhalb der Klasse Methoden nur hinzugefügt (+=) oder entzogen (-=) werden können. Dies nennet man das Abonnieren bzw. Deabonnieren von Events. Eine Zuweisung von ‚null' oder das Setzen einer einzelnen Methode (=) ist nicht zulässig.

EventHandler-Delegates verlangen Methoden ohne Rückgabetyp (void) und mit den beiden Übergabeparametern ‚object' sender (welches den Aufrufer des Events definiert) sowie ‚EventArgs' args (welches je nach Event verschiedene Informationen transportiert). ‚EventArgs' hat standardmäßig keine Eigenschaften zur Übergabe. Um den EventListener-Methoden weitere Informationen zum Event geben zu können, muss eine eigene Klasse von EventArgs abgeleitet werden. Dem EventHandler-Typ wird diese neue Klasse als generischer Typ mitgegeben: EventHandler onClick = delegate(Object sender, CustomEventArgs args){…}

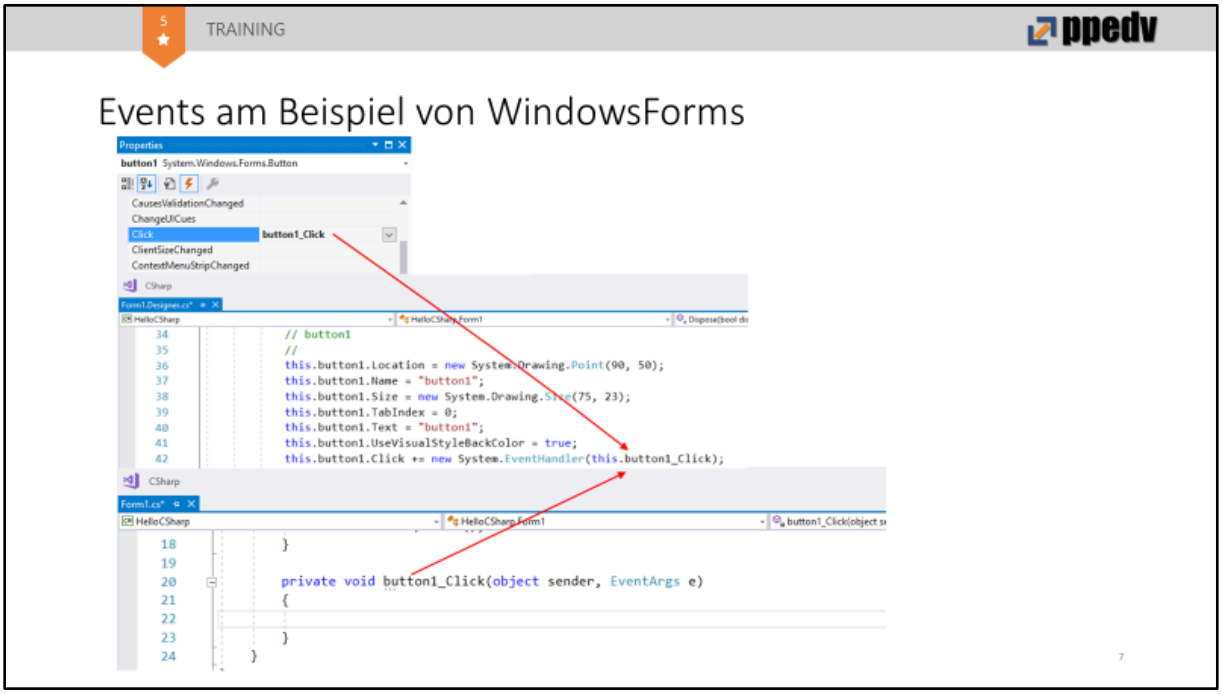
Timer verwenden



‚Timer' ruft in einem festgelegten Intervall (angegeben in Millisekunden) das Event Elapsed auf. Diesem Event kann nun eine Methode mit der zuvor erläuterten Signatur zugewiesen werden, welche immer am Ende eines Intervalls abgerufen wird.

Timer werden in einem vom Hauptthread separierten Thread ausgeführt.

Events am Beispiel von WindowsForms



Durch Doppelklick ins Click-Feld der Button1-Events wird automatisch eine Methode *button1_Click* mit der EventHandler-Signatur in Form1.cs angelegt und diese Methode in *Form1.Designer.cs* dem Click-Delegate vom Objekt Button1 hinzugefügt.

Eine manuelle Erzeugung sowie Zuweisung ist z.B. über den Konstruktor des Forms möglich.

Partielle Klassen am Beispiel WindowsForms

TRAINING

ppedv

Partielle Klassen am Beispiel WindowsForms

- jedes Formular in einer WindowsForms-Anwendung ist ein Objekt einer von der Klasse Form abgeleiteten Klasse, z.B. Form1
- die Klasse Form1 wird in 2 verschiedenen Dateien deklariert, dies ist möglich durch das Schlüsselwort „partial“

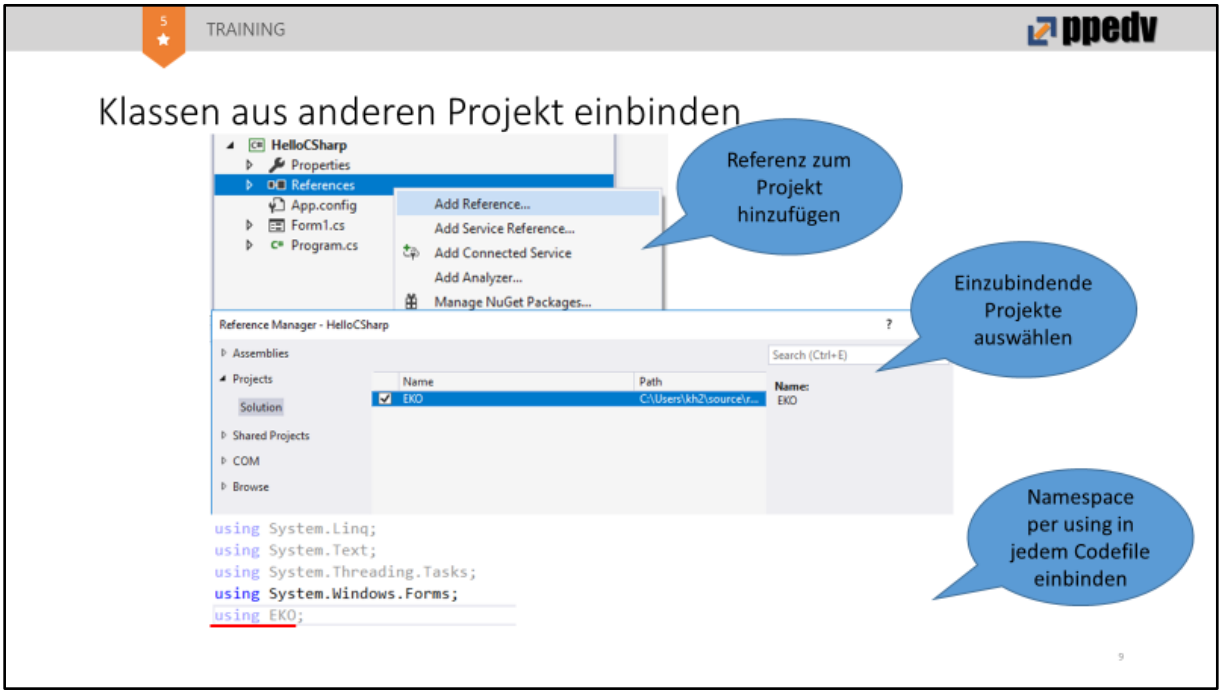
CSharp — Form1.cs* — HelloCSharp

```
10
11  namespace HelloCSharp
12  {
13      public partial class Form1 : F
14      {
15          public Form1()
16          {
17              InitializeComponent();
18          }
```

CSharp — Form1.Designer.cs* — HelloCSharp

```
1   namespace HelloCSharp
2   {
3       partial class Form1
4       {
5           /// <summary>
6           /// Required designer variabl
7           /// </summary>
8           private System.ComponentModel
9
```

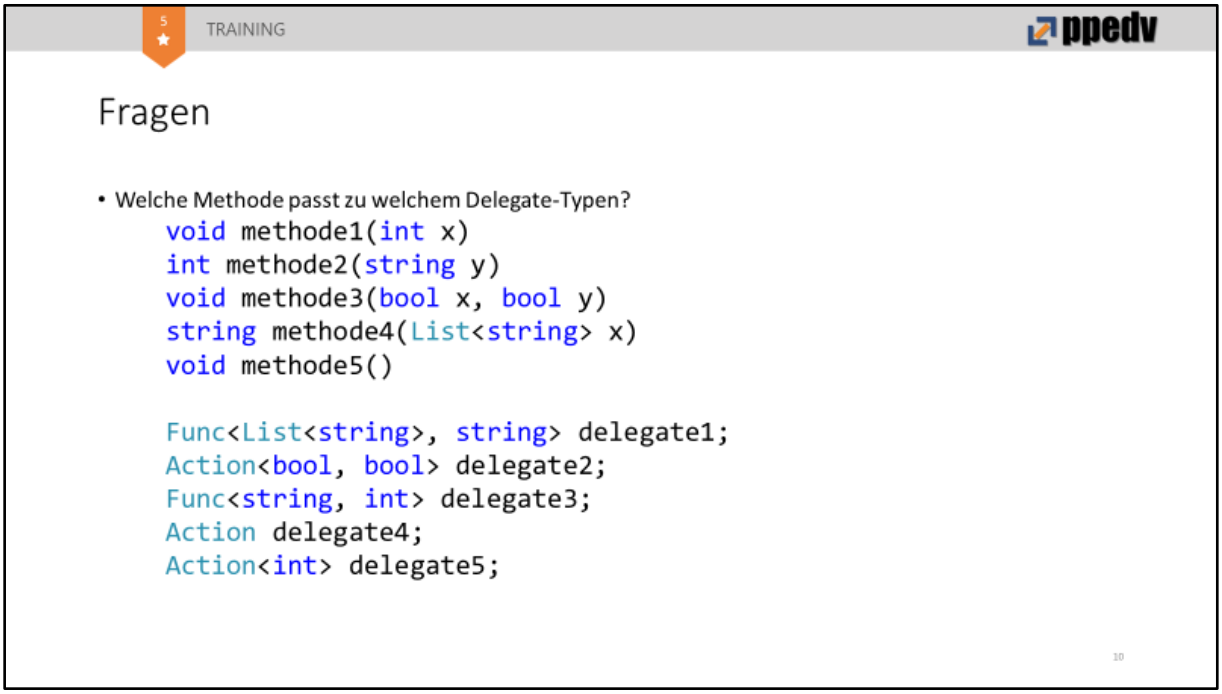
Klassen aus anderen Projekt einbinden



Per Rechtsklick auf die References des Projekts im Solution Explorer (standardmäßig am rechten Rand) kann über die Schaltfläche ‚Add Reference' ein anderes Projekt / Programm / Klassenbibliothek dem Projekt hinzugefügt werden. Nun kann auf alle als public deklarierten Elemente in dem zusätzlichen Projekt zugegriffen werden.

Will man von der verkürzten Zugriffsweise Gebrauch machen (d.h. man muss nicht immer den vollständigen Pfad der Elemente angeben), muss das neue Projekt per ‚using'-Anweisung in den einzelnen Formularen eingebunden werden.

Fragen



Antworten:

methode1 -> delegate5

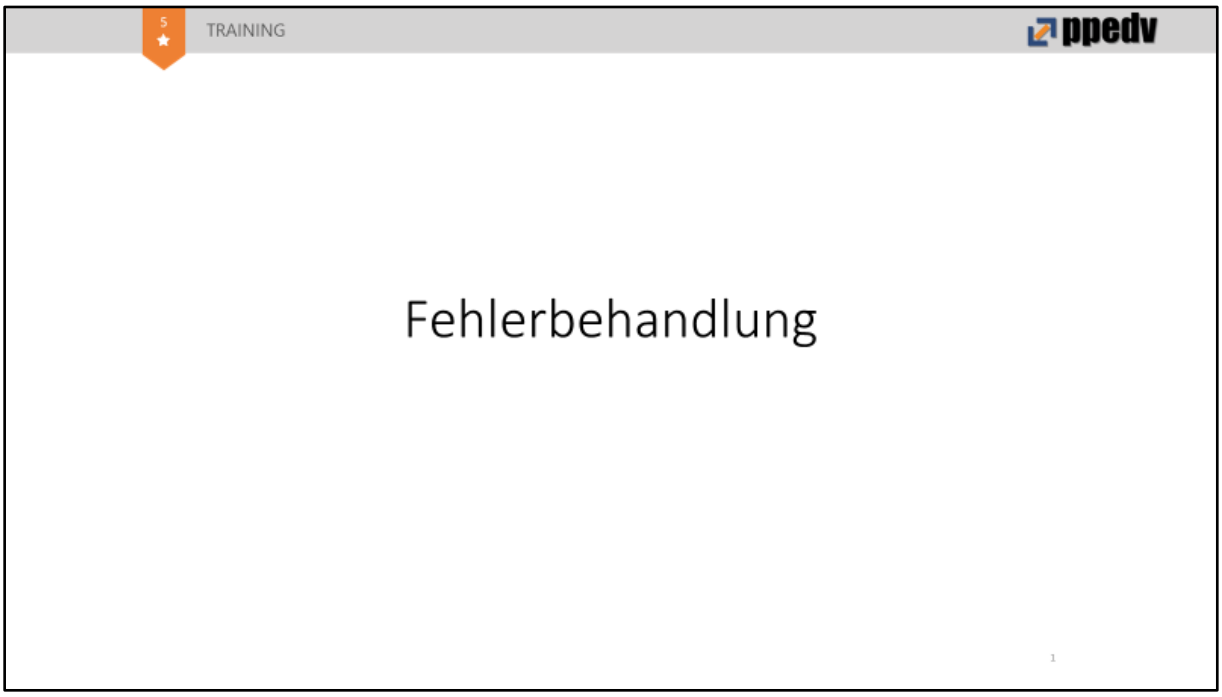
methode2 -> delegate3

methode3 -> delegate2

methode4 -> delegate1

methode5 -> delegate4

Fehlerbehandlung



Exception-Handling

Exception-Handling ist eine Möglichkeit fehlerhafte Benutzereingaben, falsche Übergabeparameter und andere Laufzeitfehler abzufangen und zu bearbeiten. Dazu werden sogenannte TRY-CATCH-Blöcke verwendet.

In den TRY-Block werden jene Anweisungen geschrieben, welche möglicherweise einen Fehler produzieren können. Würde so ein Fehler außerhalb eines TRY-Blocks auftreten, wird die Ausführung des Programms beendet, was in einem umgangssprachlichen Absturz des Programms endet.

Wenn im TRY-Block ein Fehler auftritt (man spricht von einer geworfenen Exception), springt die Programmausführung in den ersten CATCH-Block, der in seiner Signatur eine derartige Exception definiert und führt den Code aus, welcher in diesem Block steht. Dabei ist die Reihenfolge der CATCH-Blöcke wichtig: Es wird der erste CATCH-Block ausgeführt, der zum Fehler passt, d.h. wenn das geworfene Exception-Objekt vom gleichen Typ ist wie der erwartete Parameter bzw. eine Ableitung davon. Der letzte CATCH-Block im obigen Beispiel wird in jedem Fall ausgeführt, da jedes geworfene Exception-Objekt von der Klasse Exception abgeleitet ist.

Der optionale FINALLY-Block wird in jedem Fall ausgeführt, selbst wenn innerhalb eines Blockes mittels ‚return' die Methode abgebrochen wird, weshalb innerhalb dieses Blocks Anweisungen stehen sollten, die in jedem Fall ausgeführt werden sollen.

Nach Ausführung des Codes (entweder nach der fehlerfreien Ausführung des TRY-Blocks oder der Ausführung eines CATCH-Blocks und eventuell der Ausführung eines FINALLY-Blocks) wird der ‚normale' Code unter dem TRY-CATCH-Block weiter ausgeführt.

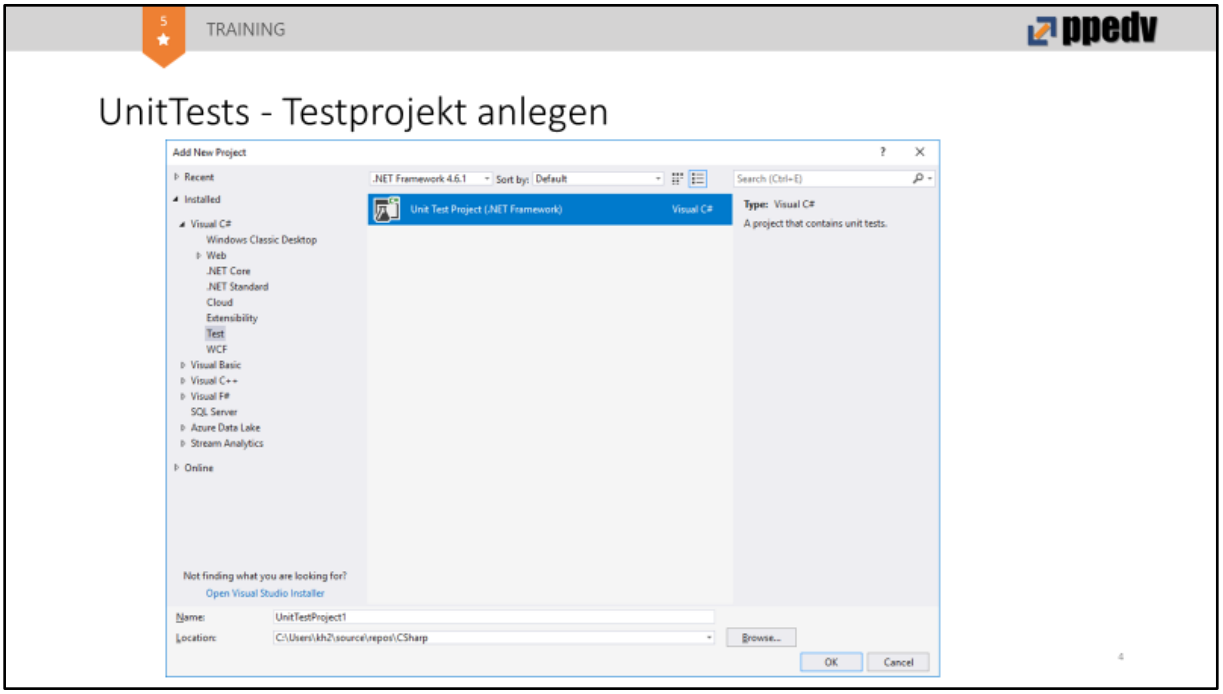
Eigene Exception-Klasse mit Fehlermeldung

Neben den vordefinierten Exceptions können auch eigene Exception-Klassen geschrieben werden. Diese müssen von der Klasse ‚Exception' erben.

Das manuelle Werfen von Exceptions erfolgt über das Stichwort ‚throw', wobei auch diese Exceptions von einem TRY-CATCH-Block abgefangen werden müssen.

Mittels ‚throw' können abgefangene Fehler in einem CATCH-Block „weitergeworfen" werden, d.h. die Exception wird auf eine höhere Befehlsebene (z.B. zum Aufrufer der aktuellen Methode) geworfen. Dort muss diese dann in einem weiteren TRY-CATCH-Block erneut abgefangen werden.

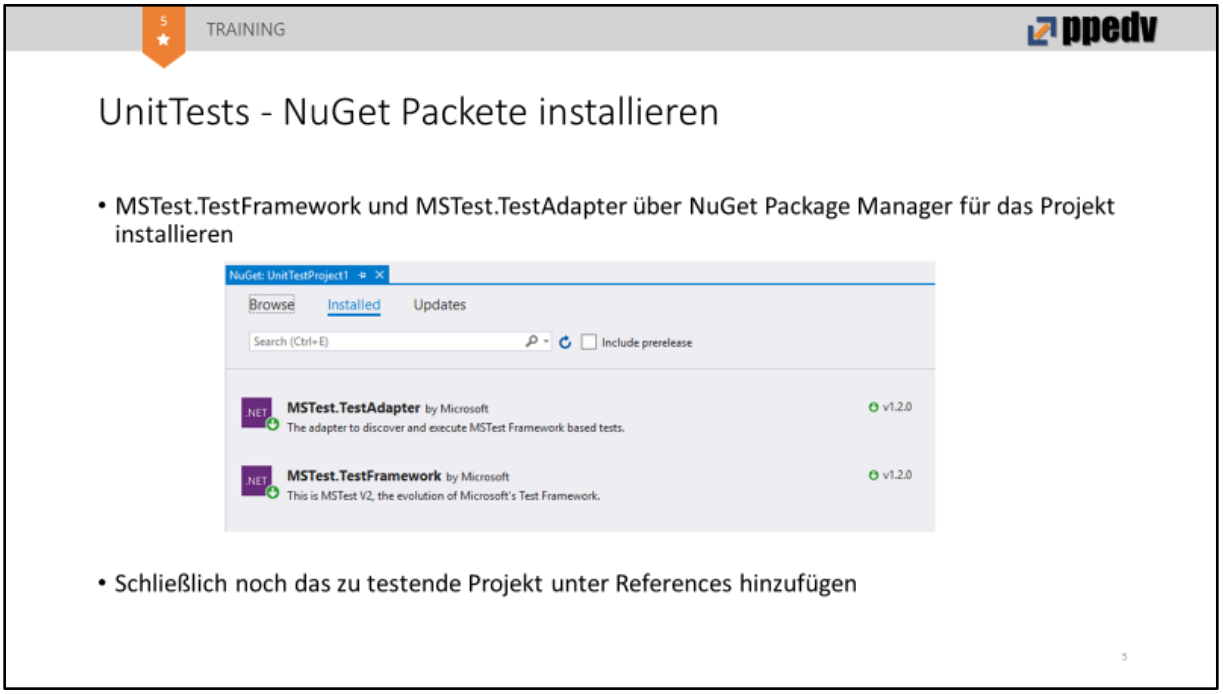
UnitTests - Testprojekt anlegen



UnitTests sind eine Möglichkeit einzelne Funktionen des Programmes auf ihre korrekte Funktionsweise hin zu überprüfen.

Ein neues UnitTest-Projekt wird ebenso wie jedes andere Projekt der Projektmappe hinzugefügt.

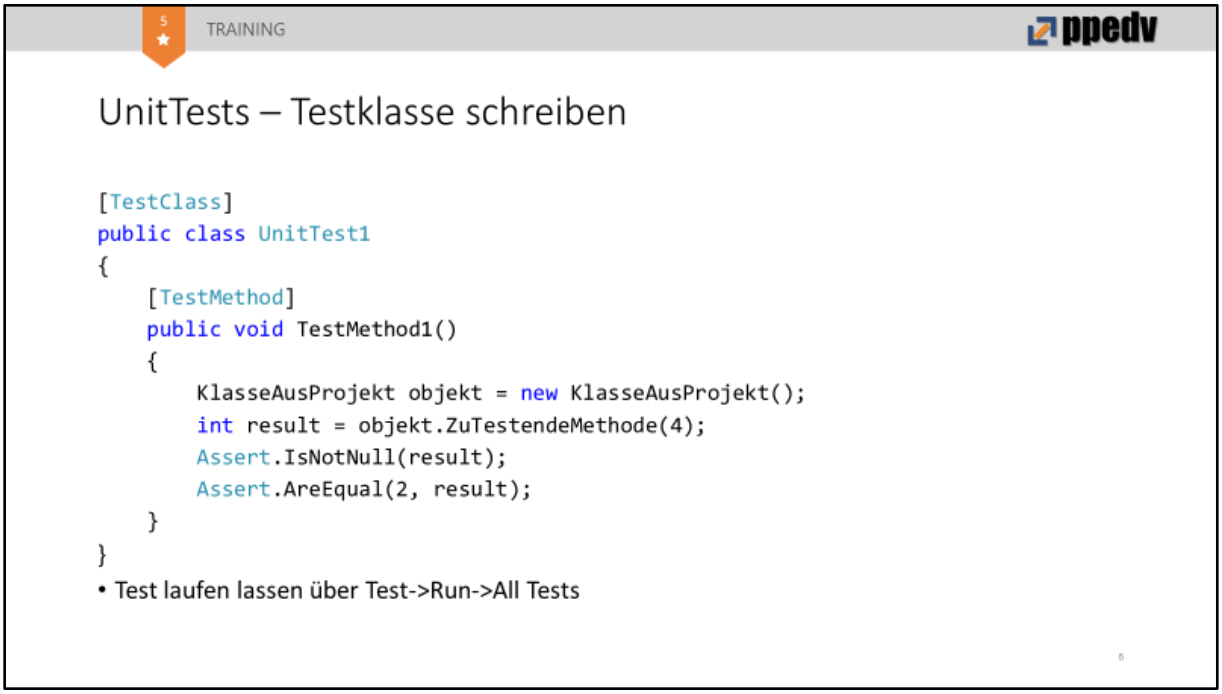
UnitTests - NuGet Packete installieren



Damit die UnitTests funktionieren müssen über die NuGet-Paketeverwaltung des UnitTest-Projekts die Pakete ‚MSTest.TestFramework' und ‚MSTest.TestAdapter' installiert werden.

Zudem muss den Referenzen des UnitTest-Projekts noch das zu testende Projekt hinzugefügt werden.

UnitTests – Testklasse schreiben



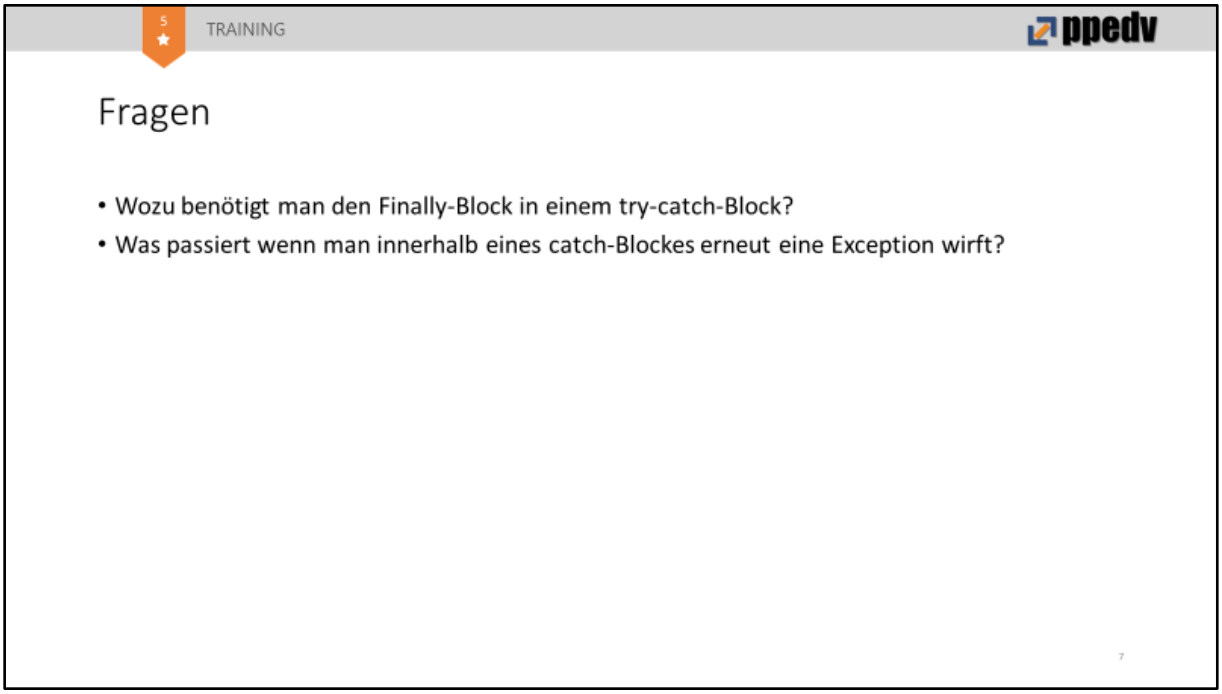
Innerhalb der TestMethod() kann nun auf die Funktionen des zu testenden Projekts zugegriffen werden. Mittels der für UnitTests zur Verfügung gestellten Testklasse ‚Assert' und deren Methoden können nun Laufzeitwerte des zu testenden Projekts überprüft und verglichen werden.

Dabei ist zu beachten, dass der UnitTest als neues Projekt nur auf als ‚puplic' markierte Elemente zugreifen kann.

Den Test kann man nun über den Menüpunkt ‚Test' von VisualStudio starten.

Auf folgende zusätzliche Einschränkungen sollte man ebenfalls achten: Nur mit [TestClass] und [TestMethod] annotierte Klassen und Methoden werden als Test ausgeführt. Hilfsmethoden bzw. Hilfsklassen innerhalb des UnitTest-Projekts, welche nur bei der Erstellung verschiedenen Test-Szenarien genutzt werden, sollten nicht annotiert werden. Zudem muss jede Test-Methode mindestens einen ‚Assert'-Befehl beinhalten.

Fragen



Antworten:

Der FINALLY-Block wird unabhängig von den Befehlen, welche im TRY- bzw. CATCH-Block ausgeführt werden, immer ausgeführt. D.h. hier sollten Befehle notiert werden, auf deren Ausführung unter keinen Umständen verzichtet werden kann (z.B. das Schließen von Ressourcen-Streams (vgl. Modul 10).

Wenn der ‚throw'-Befehl innerhalb eines CATCH-Blocks verwendet wird, wird die Exception auf die nächsthöhere Programmebene geworfen, wo sie erneut abgefangen werden muss.

Dateien schreiben/lesen

Auf externe Ressourcen wird über sogenannte Streams zugegriffen.

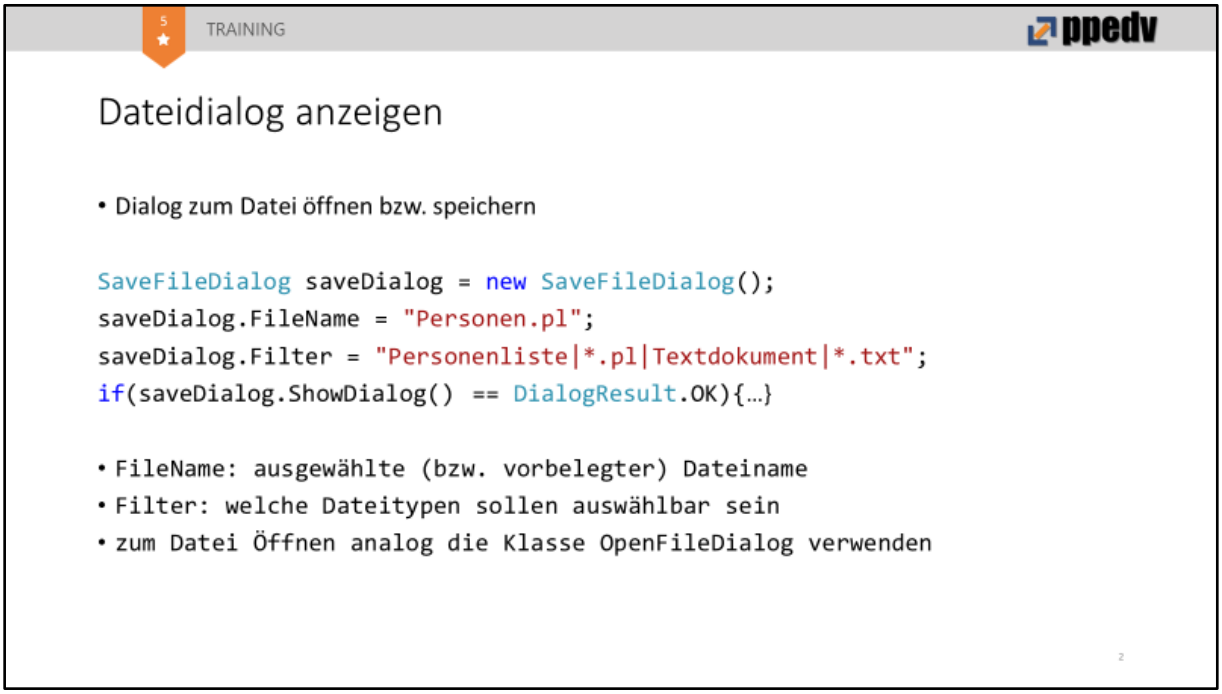
Für den Lese-/Schreibzugriff auf Dateien, welche z.B. auf der Festplatte liegen, kann man die Klassen ‚StreamReader' bzw. ‚StreamWriter' verwenden. Bei deren Initialisierung wird der Ort als String angegeben, an welchem sich die zu lesende/beschreibende Datei befindet. Dies kann ein relativer Pfad, ausgehend vom Speicherort des Programms oder ein absoluter Pfad auf dem System sein.

Unter anderem mittels der Methoden WriteLine() des StreamWriters und ReadLine() des StreamReaders können nun Zeilen in die Datei geschrieben bzw. aus der Datei gelesen werden.

Nach dem Zugriff müssen die Stream in jedem Fall wieder geschlossen werden, da sonst der der Zugriff durch andere Programme auf die Datei nicht möglich ist.

Bei den hier vorgestellten Klassen ist zu beachten, dass auf diese Weise nur Strings abgespeichert bzw. gelesen werden können.

Dateidialog anzeigen

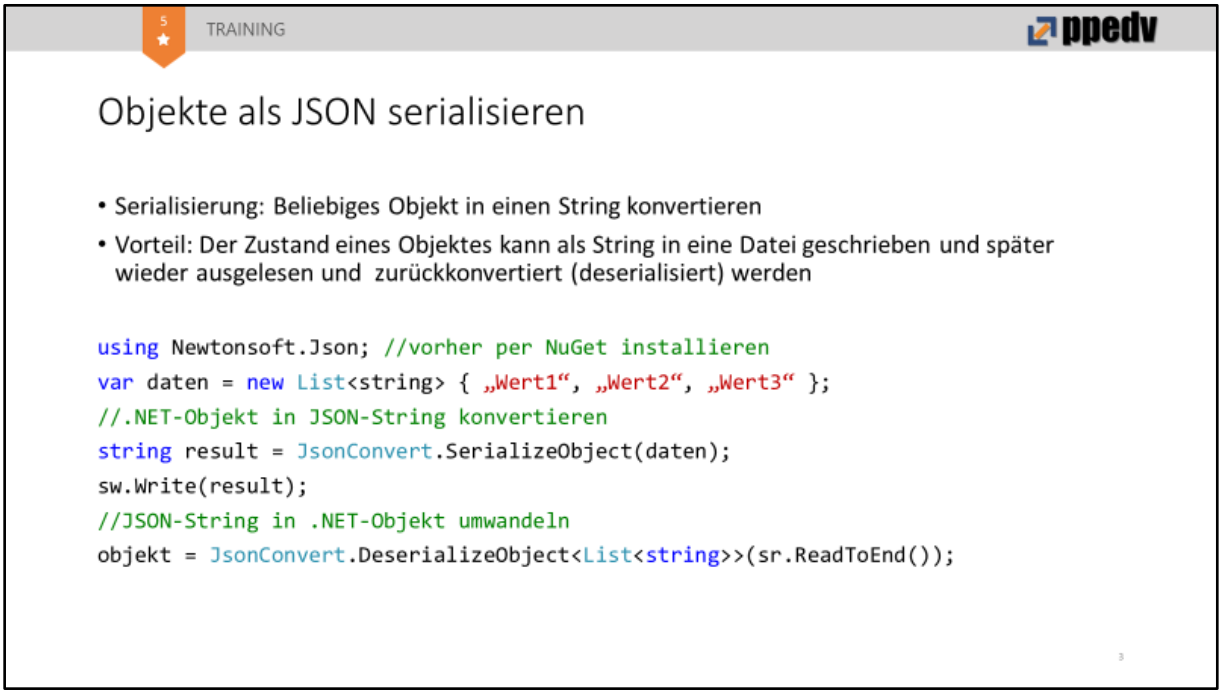


Um dem User eine komfortable Methode zum Laden / Speichern von Dateien zur Verfügung zu stellen, existieren die Klassen ‚SaveFileDialog' und ‚OpenFileDialog'. Diese beinhalten die aus Windows-Programmen bekannten Dialoge zum Speichern bzw. Laden.

Die Dialogklassen bieten diverse Eigenschaften, durch welche sich zum Beispiel ein Standart-Dateiname (FileName), ein Standart-Pfad (InitialDirectory) oder der Dateityp-Filter (Filter) definieren lassen. Bei letzterem werden die einzelnen Beschreibungen und die Dateiendungen, getrennt durch ein , | ' als String übergeben (vgl. obiges Beispiel).

Mittels der Methode ShowDialog() wird der Dialog dann aufgerufen. Diese Methode gibt ein Objekt vom Enumeratortyp DialogResult zurück, welcher Aufschluss über die Auswahl des Users gibt. Nach Aufruf des Dialogs befindet sich in der Eigenschaft FileName der Pfad, an welchem sich die zu lesende/schreibende Zieldatei befindet.

Objekte als JSON serialisieren



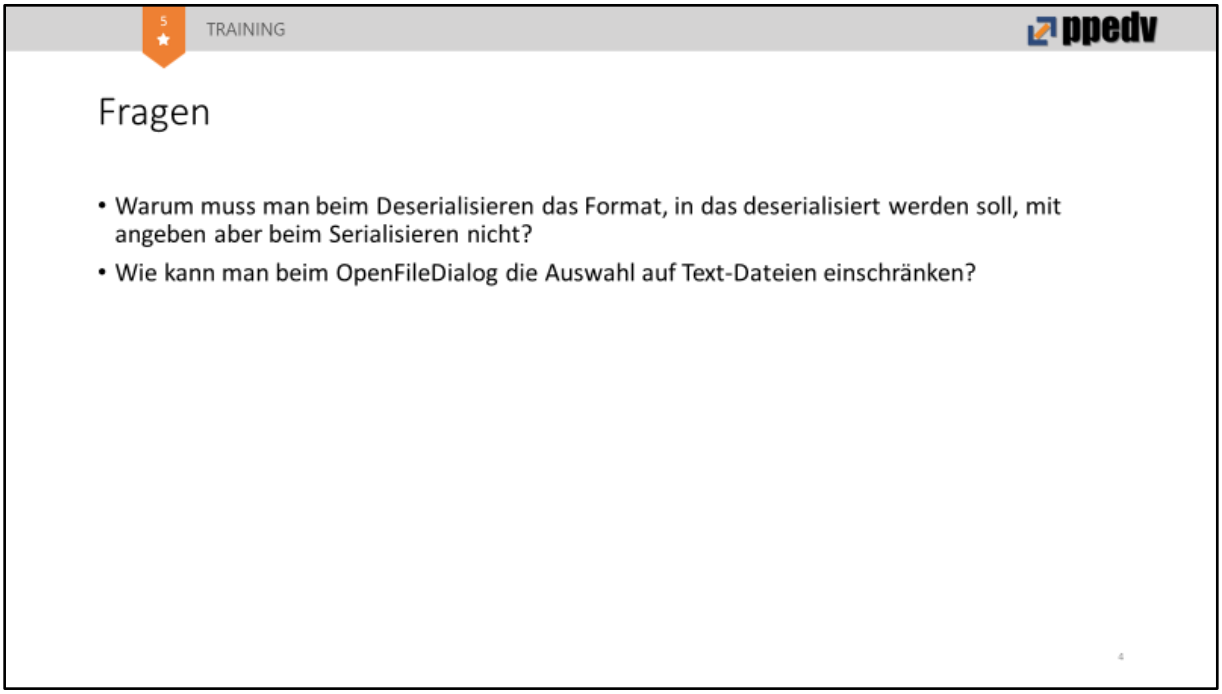
Mittels Serialisierung können auch komplexe Objekte gespeichert werden. Dazu werden diese vor dem Schreiben in eine speicherbare Form umgewandelt (serialisiert).

Die hier vorgestellte JSON-Serialisierung wandelt Objekte in Strings um, welche dann mit den vorher beschriebenen Methoden abgespeichert werden können. Dazu muss per NuGet zunächst das ‚Newtonsoft.Json‘-Paket installiert werden. Danach kann man per SerializeObject()-Methode der JsonConvert-Klasse ein komplexes Objekt in einen String umgewandelt werden.

Umgekehrt kann durch die JsonConvert.DeserializeObject<>()-Methode ein String zurückgewandelt werden. Allerdings muss hierzu in den spitzen Klammern der Methode der Zieltyp des Objekts angegeben werden.

Zudem ist zu beachten, dass die JsonConvert-Klasse nicht den Konstruktor der Klassen sondern die Eigenschaften verwendet. D.h., dass darauf geachtet werden muss, dass die set-Zugriffsmodifier der Eigenschaften zugänglich genug sind.

Fragen

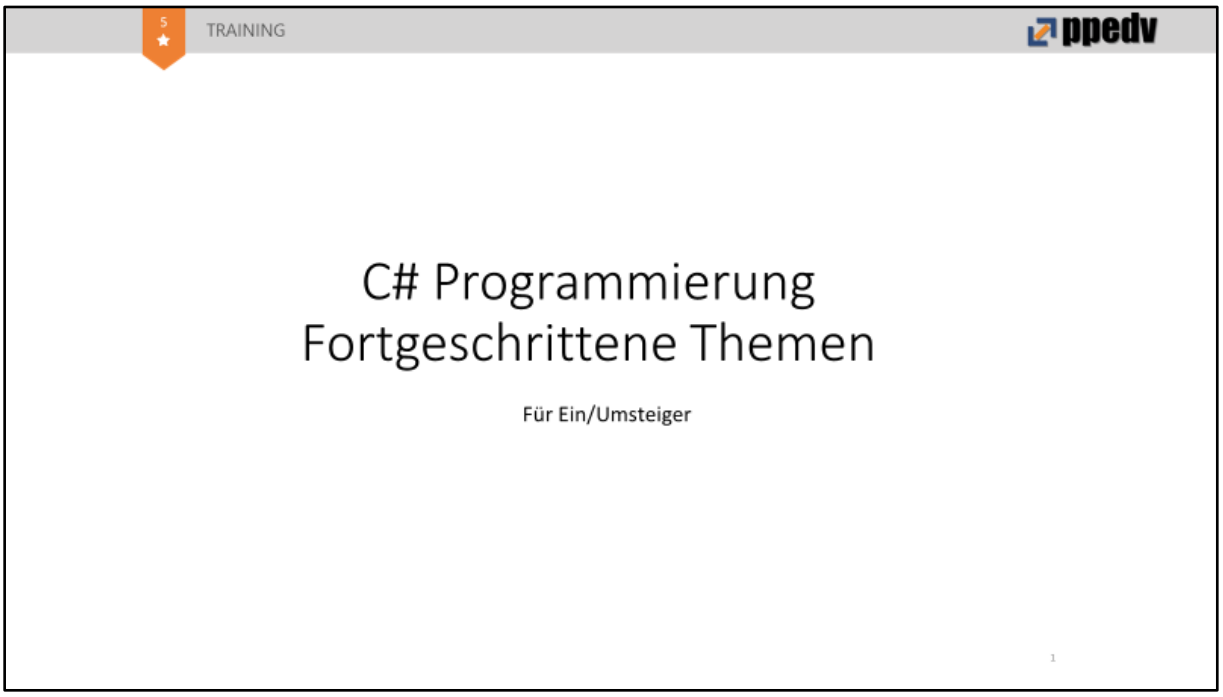


Antworten:

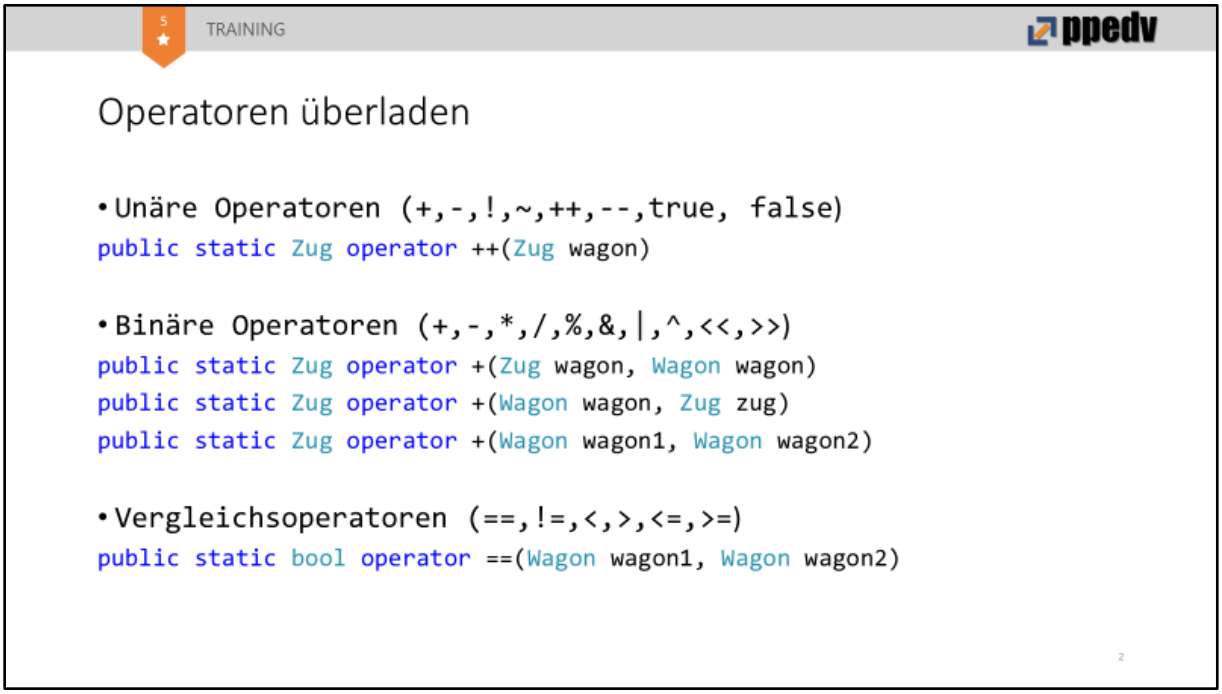
Bei der Serialisierung ist es für den Konverter nicht nötig die Struktur des Objekts zu kennen. Er schreibt einfach alle Felder-Werte als String hintereinander. Erst bei der Deserialisierung benötigt der Konverter eine Klassenangabe, damit er weiß, wie er die Werte wieder zu strukturieren hat.

Eine Einschränkung erfolgt über die Filter-Eigenschaft des Dialogobjekts. Dazu wird folgende Zuweisung verwendet: dialog.Filter = „Textdateien|*.txt“;

C# Programmierung Fortgeschrittene Themen



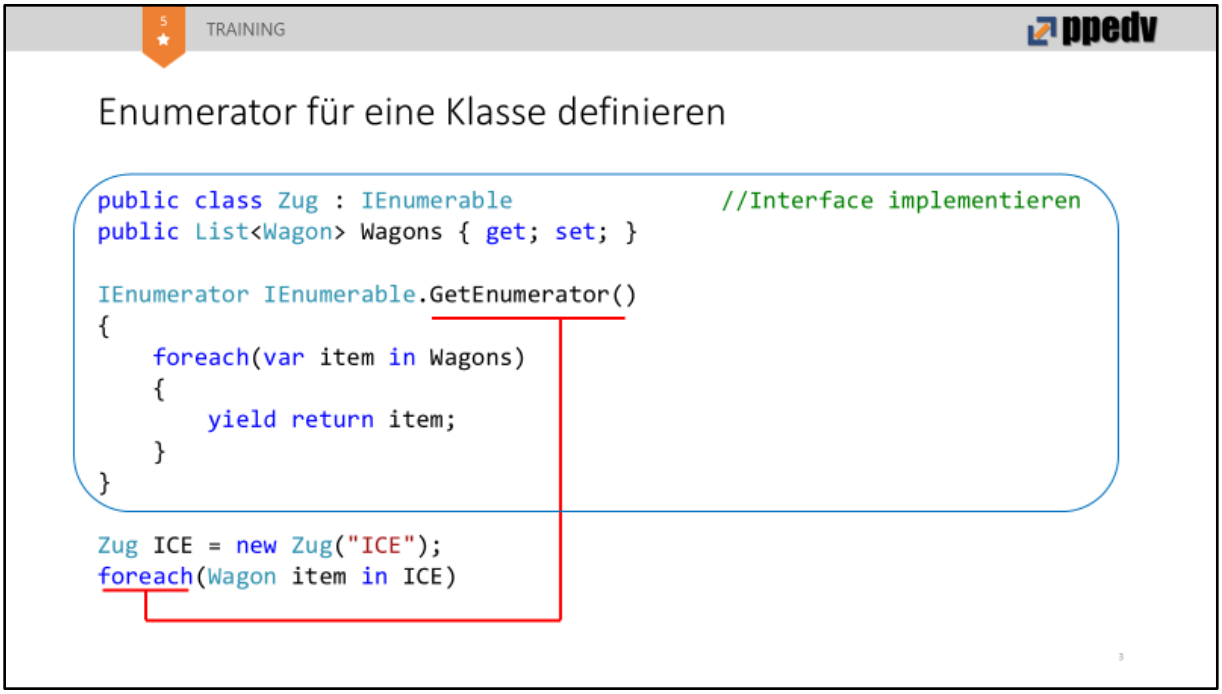
Operatoren überladen



Mittels des Stichworts ‚operator' können die Standartoperatoren für eigene Klassen überladen werden. Dazu werden diese (der obigen Notation folgend) als statische Methoden in der Klasse definiert. Dabei muss mindestens einer der Parameter vom Typ der Klasse sein, für welcher der Operator definiert wird. Zudem ist die Reihenfolge bei binären Operatoren wichtig: Der erste Parameter ist jener, der vor den Operator geschrieben wird, der Zweite jener, der nach dem Operator steht.

Zu beachten ist, dass bei der Definition bestimmter Operatoren (z.B. des Vergleichsoperators ==) auch die Gegenrichtung (!=) definiert werden muss.

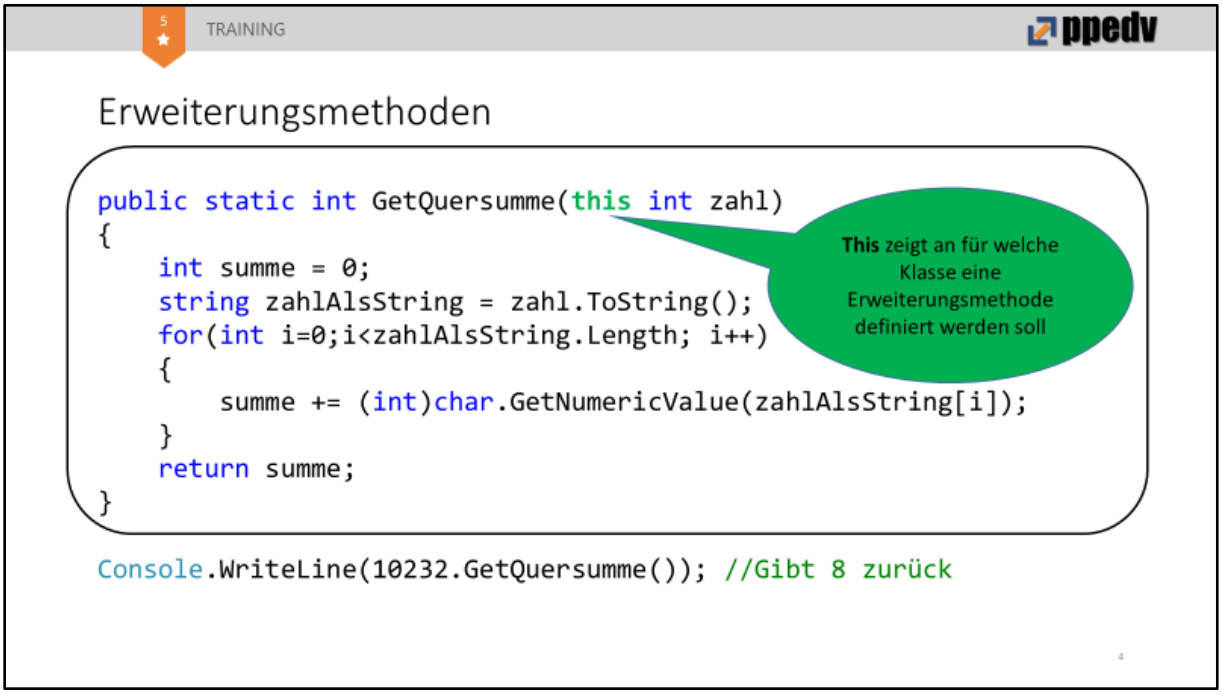
Enumerator für eine Klasse definieren



Mittels des Interfaces IEnumerable kann die ‚foreach'-Schleife für eigene Klassen definiert werden. Die Schleife ruft die durch das Interface verlangte Methode GetEnumerator() auf. Hier kann definiert werden, was in jedem Schleifendurchlauf zurückgegeben werden soll.

Dazu wird das Stichwort ‚yield' verwendet, welches zusammen mit dem ‚return'-Stichwort einen Rückgabewert pro Schleifendurchlauf definiert. Beim nächsten Schleifendurchlauf wird die Methode an der Stelle der zuletzt ausgeführten yield-return-Anweisung fortgesetzt. Mit der Anweisung ‚yield break' wird der Iterator-Prozess beendet.

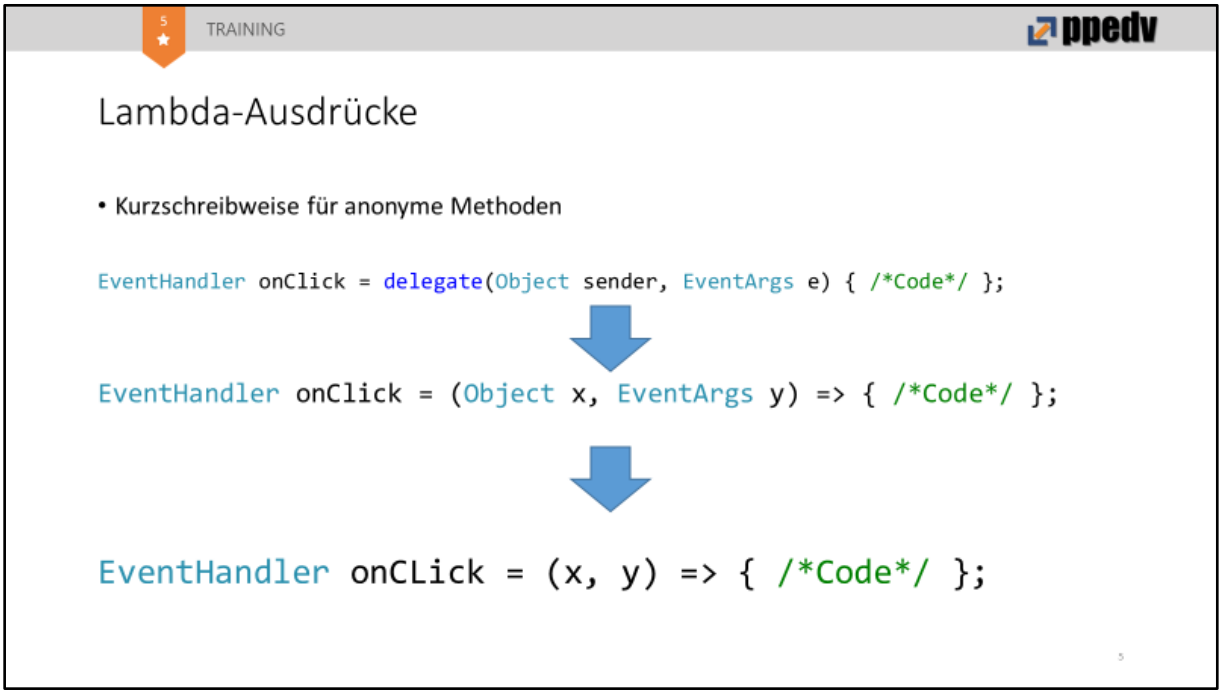
Erweiterungsmethoden



Mit Erweiterungsmethoden kann für eine bestehende Klasse eine neue Methode definiert werden, ohne das die Klassendefinition selbst verändert werden muss. Die Klasse muss nicht einmal sichtbar sein. Diese Methoden müssen innerhalb einer statischen Klasse definiert werden, welche dem aufrufenden Programm bekannt sein muss.

Der erste Parameter der Erweiterungsmethode muss auf die zu erweiternde Klasse verweisen. Es können anschließend noch weitere Parameter definiert werden. Beim Aufruf muss der erste Parameter nicht übergeben werden, da sich dieser auf das Objekt bezieht, welches die Erweiterungsmethode aufruft.

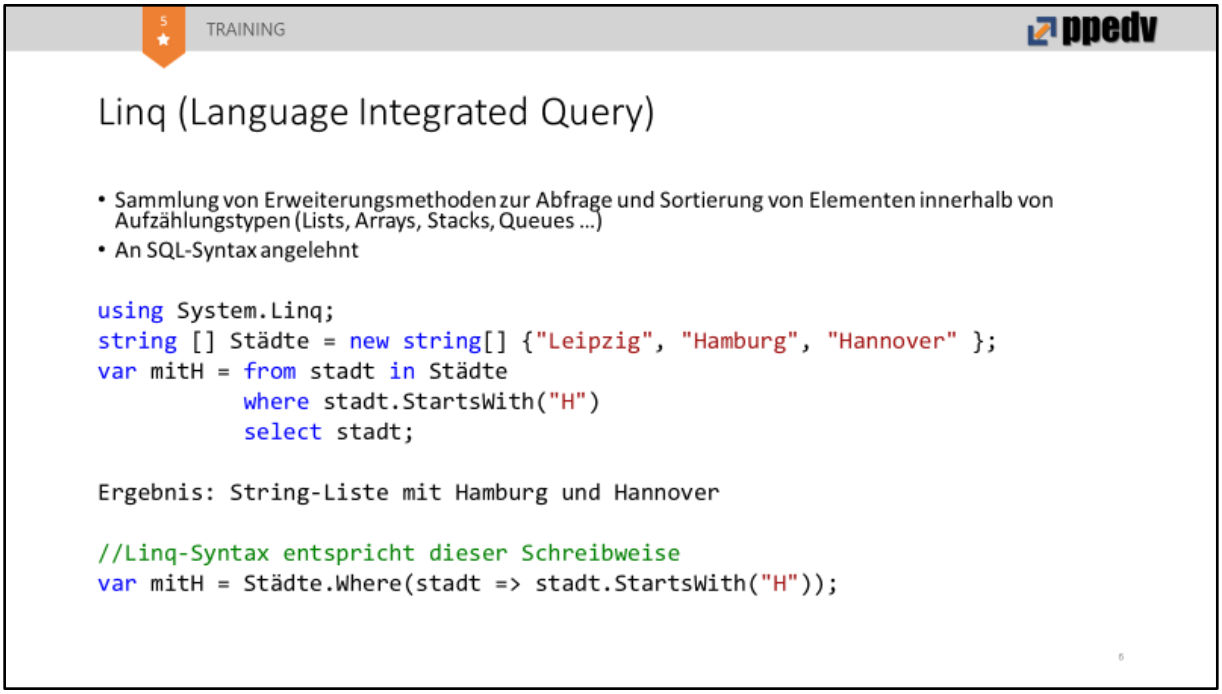
Lambda-Ausdrücke



Lambda-Ausdrücke sind eine Möglichkeit anonyme Methoden in stark verkürzter Form zu definieren. Dabei kann auf Rückgabe- und Übergabeparametertyp verzichtet werden, da sich diese bereits eindeutig aus dem Delegate-Typ ergeben.

Demnach werden für die Parameter nur Variablennamen definiert, welche dann mittels =>-Operator auf einen Codeblock verweisen, welche den Methodenkörper beinhalten. Diese Ausdrücke werden ausschließlich bei Delegate-Zuweisungen genutzt.

Linq (Language Integrated Query)

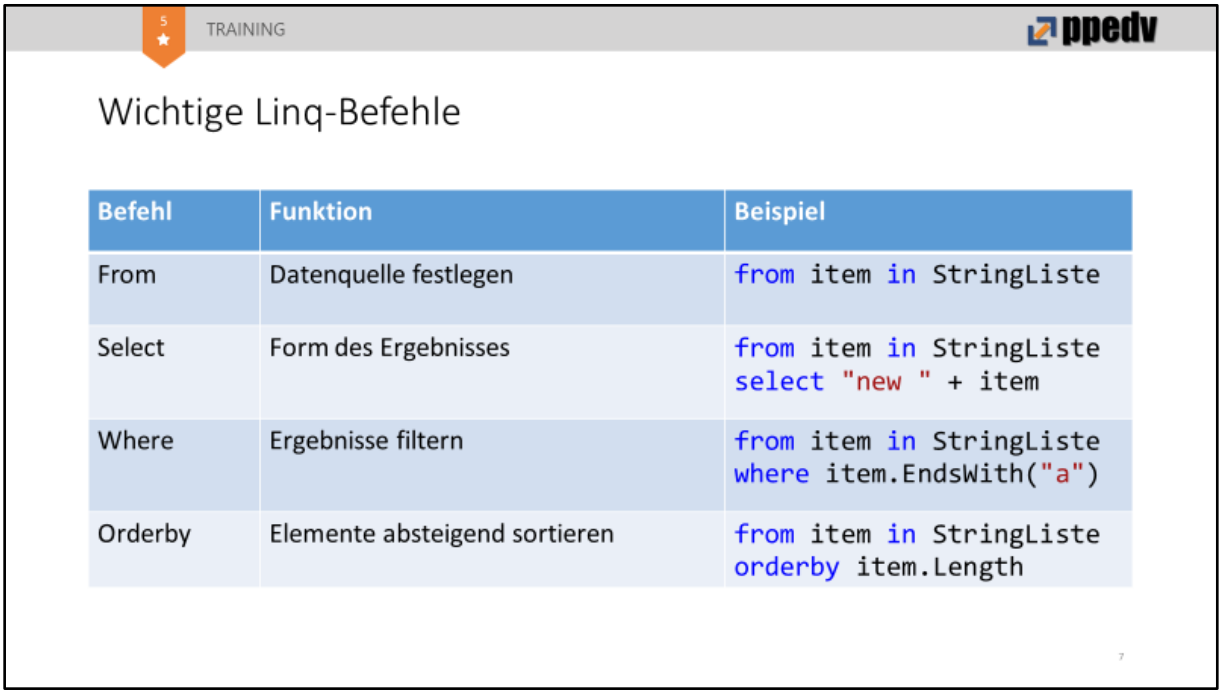


LINQ ist eine an die Syntax von SQL angelehnte Schreibweise für spezielle Methoden, welche für das Durchsuchen und Verwalten von Sammlungen verwendet werden.

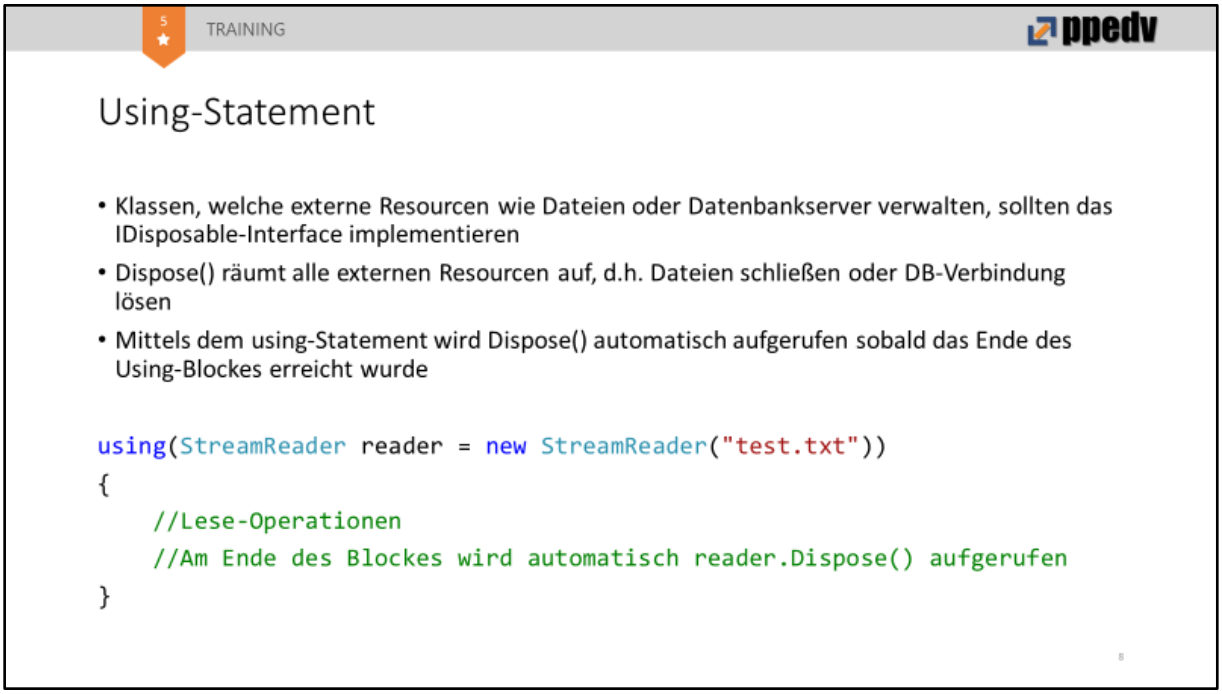
LINQ-Methoden geben entweder einzelne Objekte oder Ergebnislisten vom Typ IEnumerable zurück, die mit foreach durchlaufen werden können.

Der Compiler konvertiert die from-in-where-select-Syntax automisch in eine Kette von Erweiterungsmethodenaufrufe.

Wichtige Linq-Befehle



Using-Statement



Es können auch mehrere Variablen gleichzeitig im Using-Statement definiert werden, welche am Ende des Blocks automatisch ihre Dispose-Funktion ausführen.

Diese Variablen sind nur innerhalb des jeweiligen Using-Blocks bekannt. Beispiele für Klassen in .NET, welche IDisposable implementieren:

- StreamReader
- StreamWriter
- SqlConnection