5 C#-Scripte für Unity programmieren

In den vergangenen Kapiteln hast du bereits erfahren, dass Unity für sehr viele Standardfunktionen fertige Components mitliefert. Du musst also nicht jede Kleinigkeit selber programmieren. Trotzdem kommst du in Unity aber nicht darum herum, auch selber ein paar Zeilen Code schreiben zu müssen. Deshalb werde ich dir in diesem Kapitel die Grundlagen zum Erstellen eigener *Scripte* (auch *Behaviours* oder *Components* genannt) beibringen.

Ein *Script* ist zunächst eine Textdatei, die Anweisungen in einer Programmiersprache enthält, welche sowohl von Menschen als auch von Programmen gelesen und verstanden werden kann. Programmiersprachen sind in der Regel darauf ausgelegt, dass man sie als Mensch gut versteht, und nicht darauf, dass eine Maschine das Geschriebene schnell abarbeiten kann. Da es bei der Ausführung von Scripten jedoch um Millisekunden geht, werden sie vor dem Ausführen in eine Maschinensprache übersetzt, welche der Computer schneller interpretieren kann, für Menschen aber schwer zu verstehen ist. Dieses Übersetzen des Scripts in Maschinensprache nennt sich *Kompilieren* und das Programm, das dein Script übersetzt, nennt sich *Compiler*.

Sobald ein Script in deinem Projektordner liegt, startet Unity automatisch den *Compiler*, welcher das Script dann übersetzt. Die ursprüngliche Script-Datei bleibt dabei jedoch unangetastet, da die kompilierten Varianten in einem anderen Bereich zwischengespeichert werden. Du kannst also jederzeit noch Änderungen vornehmen und Unity aktualisiert dann automatisch die kompilierte Version.

Sollte dein Script einen Fehler enthalten, findest du die sogenannten *Compiler-Fehler* in der *Console*. Unity startet diesen Kompilierungsvorgang vollkommen automatisch jedes Mal, wenn du eine Änderung an einem Script vorgenommen oder ein neues Script angelegt hast, ohne dass du es per Hand auslösen musst.

Mit *Scripten* kannst du Unitys Funktionen nahezu endlos erweitern. In deinen Scripten hast du zudem Zugriff auf alle GameObjects und Components in der Scene und kannst sie nach deinen Wünschen ändern. Auf diese Weise kannst du zum Beispiel ein *Light*-Component auf Tastendruck ein- oder ausschalten.

Unity unterstützt zwei Programmiersprachen, *C#* und *JavaScript*. Während in älteren Unity-Versionen noch viel mit JavaScript gearbeitet und teilweise auch offizielle Beispiele in dieser Sprache veröffentlicht wurden, hat sich *C#* mittlerweile als die Standardsprache etabliert. Sowohl *Unity Technologies* als auch die meisten Entwickler verwenden nur noch C#. Deshalb werden wir in diesem Buch ebenfalls C# verwenden.

5.1 C#, .Net und Mono

C# (gesprochen „C-Sharp“) ist eine objektorientierte Programmiersprache, die von Microsoft entwickelt wurde. Die Sprache orientiert sich an Konzepten der Programmiersprachen Java und C++, wenn du bereits Erfahrungen mit diesen hast, wird dir der Umstieg wahrscheinlich leichtfallen.

Die Sprache an sich ist plattformunabhängig, allerdings gibt es unterschiedliche Implementierungen, die nicht immer plattformunabhängig sind. Zur Erklärung: Die Sprache legt zunächst nur fest, welche Wörter es gibt und was sie bedeuten, also *was* passieren soll. Die Programmiersprache legt nicht fest, *wie* der Computer dieses Ziel erreichen soll, das macht die *Implementierung*. Unter Windows verwendet man in der Regel Microsofts Implementierung, welche Teil des *.Net Frameworks* ist. Diese Implementierung der Sprache ist allerdings nicht plattformunabhängig und funktioniert nur unter Windows.

Alternativ dazu gibt es zum Beispiel *Mono*. Mono ist eine quelloffene Implementierung der Programmiersprache. Sie wird seit 2011 entwickelt und erlaubt es, C# auch auf iOS, Android, Unix und anderen Plattformen zu verwenden. Diese Implementierung stammt nicht von Microsoft selber, sondern ist ein Gemeinschaftsprojekt mehrerer Entwickler. Anfänglich wurde das Projekt hauptsächlich durch Mitarbeiter der Firma *Novell* vorangetrieben und später durch das Unternehmen *Xamarin*. 2016 wurde *Xamarin* aus San Francisco durch Microsoft übernommen und ist jetzt eine Tochtergesellschaft von Microsoft, wodurch Microsoft Hauptsponsor des Mono-Projektes wurde.

Unity verwendet Mono, um deine Spiele unabhängig von der Plattform zu erstellen. Die Mono-Implementierung hängt der offiziellen Version meistens ein wenig hinterher. In Unity 2017.1 wird standardmäßig eine Mono-Version verwendet, die dem Stand von *.Net 3.5* entspricht. Optional kannst du auch eine Mono-Version aktivieren, die dem Stand von .Net 4.6 entspricht, jedoch ist diese Unterstützung nur experimentell. Das bedeutet, du solltest es nur aktivieren, wenn du gute Gründe dafür hast, da sie fehleranfälliger sein kann.

5.2 Grundlagen

In diesem Kapitel beginnen wir ganz am Anfang und sehen uns zunächst einmal an, wie C# aufgebaut ist und wie man es verwendet.

5.2.1 C#-Syntax

Wie schon erwähnt, wird jedes Script von der Programmiersprache in eine Sprache übersetzt, die der Computer schneller verarbeiten kann. Damit dieses Übersetzen funktioniert, muss man sich beim Schreiben von Quellcode an eine bestimmte Syntax (vergleichbar mit Grammatiken) halten, die von der Programmiersprache bestimmt wird. Auf diese Weise weiß der Compiler dann, wo ein Abschnitt anfängt, wo ein Abschnitt endet und welche

Anweisungen wann ausgeführt werden sollen. Die genaue Syntax für die einzelnen Funktionen schauen wir uns in den jeweiligen Kapiteln genauer an. Nachfolgend findest du allgemeine Eigenschaften der C#-Syntax, die für jede Funktion wichtig sind:

- In C# markiert ein Semikolon **;** das Ende einer Anweisung.
- Zusammengehörige Codeblöcke werden in C# mit den geschweiften Klammern **{** und **}** umfasst.
- Zu jeder geöffneten Klammer **{**, **(**, **[** muss eine geschlossene Klammer in umgekehrter Reihenfolge vorhanden sein **]**, **)**, **}**.

Listing 5.1 Ein Beispielcode in C#-Syntax

```
public class Example
{
    string message = "";

    public void SayHelloMethod()
    {
        HelloMethod();
    }
    public void HelloMethod()
    {
        message = "";
        message = message + "Hello";
        message = message +
                          " World";
    }
}
```

Vergisst du das Semikolon am Ende einer Anweisung, wird der *Compiler* denken, dass die nächste Zeile zur selben Anweisung gehört. In C# können beliebig viele Zeilenumbrüche oder Leerzeichen zwischen zwei Wörtern stehen, da sie keine eigene Funktion haben, sondern nur der übersichtlicheren Darstellung im Code-Editor dienen. In Listing 5.1 ist zum Beispiel die letzte Anweisung mit der `message`-Variablen über zwei Zeilen verteilt. Für den *Compiler* ist das aber egal und er verarbeitet sie genauso wie die Anweisung darüber. Diese Eigenschaft ist in erster Linie praktisch, damit du deinen Code übersichtlich formatieren kannst. Lange Anweisungen würden sonst schnell unübersichtlich werden, wenn sie in einer Zeile stehen müssten.

Außerdem siehst du in dem Listing, dass auf jede öffnende Klammer eine schließende Klammer in umgekehrter Reihenfolge folgt. Das ist wichtig, weil eine Klammer immer einen Block oder eine Gruppe markiert, welche jeweils abgeschlossen sein müssen.

5.2.2 Kommentare

Der Compiler hält zunächst jede Zeile in deinem Script für eine wichtige Zeile, die er übersetzen muss. Aber was ist, wenn du eine Notiz in deinen Code schreiben möchtest? Gerade jetzt am Anfang möchtest du eventuell an manche Anweisungen schreiben, was sie bewirken, da du die Funktionen noch nicht komplett auswendig kennst. Für diese Fälle ist in C#

der *Kommentar* vorgesehen. Kommentare können auch genutzt werden, um Anweisungen vorübergehend zu deaktivieren, ohne sie aus dem Script löschen zu müssen.

C# besitzt für diese Zwecke zwei Varianten, einen Abschnitt als Kommentar zu markieren: den einzeiligen und den mehrzeiligen Kommentar.

Fügst du in einer Zeile einen *Doppel-Slash* „//“ ein, markierst du alles, was in derselben Zeile *hinter* dem Doppel-Slash steht, als Kommentar. In Listing 5.2 siehst du verschiedene Varianten dieses einzeiligen Kommentars. Für den *Compiler* wäre alles bis auf die Anweisung `int energy = 1;` nicht relevant, da alles andere als Kommentar markiert wurde.

Listing 5.2 Einzeiliger Kommentar

```
// Das ist ein einzeiliger Kommentar
int energy = 1; //Kommentare können auch hinter Anweisungen stehen
// Es können auch Anweisungen auskommentiert werden:
//int enemyEnergy = 2;
```

Wenn du viele Zeilen als Kommentar markieren möchtest, kannst du das grundsätzlich auch mit dem einzeiligen Kommentar machen, indem du den *Doppel-Slash* am Anfang jeder Zeile wiederholst. Bei längeren Abschnitten kann das jedoch auch lästig werden. Deswegen bietet *C#* dir die Möglichkeit, beliebig lange Abschnitte als Kommentar zu markieren: Alles, was zwischen dem *Kommentar-Start-Zeichen* „/*“ und dem *Kommentar-Stopp-Zeichen* „*/“ steht, ist ein Kommentar. In Listing 5.3 siehst du ein Beispiel für einen mehrzeiligen Kommentar. Der Compiler würde nur die Anweisung `int enemyEnergy = 2;` sehen, da alles andere *auskommentiert* ist.

Listing 5.3 Mehrzeiliger Kommentar

```
/* Hier beginnt der Kommentar
int energy = 1;
Das hier ist immer noch auskommentiert.
Hier endet der Kommentar */
int enemyEnergy = 2;
```

In Visual Studio, aber auch in anderen Code-Editoren, werden Kommentare farblich hervorgehoben. In Visual Studio werden Kommentare standardmäßig dunkelgrün dargestellt. Dadurch kannst du schnell erkennen, welche Anweisungen auskommentiert sind und welche vom Compiler übersetzt werden.

5.2.3 Variablen

Variablen sind Platzhalter, in denen Werte, zum Beispiel Zahlen oder Texte, gespeichert werden können.

Der Name einer Variablen muss eindeutig sein, damit der *Compiler* stets weiß, auf welche Variable du dich beziehst. Außerdem dürfen in Variablennamen keine Leerzeichen, Sonderzeichen (Ausnahme hier ist der Unterstrich „_“) und Umlaute enthalten sein. Bis auf diese Einschränkungen kannst du die Namen technisch gesehen frei wählen. Damit dein Code gut lesbar bleibt, gibt es allerdings noch ein paar Konventionen für das Vergeben von Variablennamen, an die du dich halten solltest:

- Der Variablenname sollte den Inhalt der Variablen aussagekräftig beschreiben.

  z. B. „*currentPosition*“ statt „cp“ oder „aVariable“
- Es ist empfohlen, englische Begriffe zu verwenden. Damit umgehst du die Umlautproblematik und dein Code passt gut zu den restlichen Ausdrücken von C# und Unity.

  z. B. „*health*“ statt „lebensenergie“
- Variablennamen sollten *Camel Case* verwenden. Das bedeutet, sie sollten mit einem Kleinbuchstaben beginnen; besteht der Name aus mehreren Wörtern, schreibst du den ersten Buchstaben von jedem weiteren Wort mit einem Großbuchstaben

  z. B. „*playerPoints*“ statt „playerpoints“, „player_points“, „PlayerPoints“ oder „PLAYERPOINTS“.
- Variablennamen sollten keine Zahlen enthalten. Wenn du das Gefühl hast, es ist sinnvoll, deine Variablen durchzunummerieren, solltest du sie ggf. in einem *Array* (Kapitel 5.2.3.5) oder einer *Liste* (Kapitel 5.4.4) zusammenfassen.

  z. B. „*playerHandLeft*“ und „*playerHandRight*“ statt „playerHand1“ und „playerHand2“.

5.2.3.1 Datentypen

In *C#* ist eine einzelne Variable kein universaler Platzhalter, sondern kann immer nur einen bestimmten *Datentyp* aufnehmen, den du zuvor angeben musst. Das Zuordnen eines Datentyps zu einem bestimmten Variablennamen wird als *Deklaration* bezeichnet. Das *erste* Zuweisen eines Wertes nennt man wiederum *Initialisierung*. Bevor du eine Variable frei verwenden kannst, muss sie *deklariert* und *initialisiert* sein.

Um eine Variable deklarieren zu können, musst du die Datentypen kennen. Datentypen unterscheidet man dabei in sogenannte *primitive Datentypen* (das sind zum Beispiel Zahlen und Texte) und *dynamische Datentypen*. Dynamische Datentypen können zum Beispiel Verweise zu anderen Klassen oder auch zu Assets (z. B. *Materials*) sein.

Zunächst schauen wir uns die primitiven Datentypen an. Was bei den Farben die „Grundfarben“ sind, sind die primitiven Datentypen bei den Datentypen; denn am Ende basiert alles auf ihnen. Die folgende Tabelle zeigt dir die meistverwendeten primitiven Datentypen und beschreibt sie ein wenig näher. Es gibt noch weitere, mit denen wirst du aber im Umgang mit Unity in der Regel nicht in Berührung kommen.

Tabelle 5.1 Die meistverwendeten primitiven Datentypen in Unity

Datentyp	Beschreibung
int	*„Integer“* genannt, stellt Ganzzahlen dar. Beispiele: 1, 5, 36, 122
float	*„Floating Point Number“* (dt. *„Fließkommazahl“*) wird üblicherweise kurz als „Float“ bezeichnet, stellt Zahlen inklusive Nachkommastellen dar. Werte werden durch ein „f“ als Float markiert. Beispiele: 1.5f, 1.0f, 1.63156f, 3.1415926f
bool	„Boolean“ genannt, stellt einen booleschen Wert dar. Beispiele: *true* oder *false* (also „wahr“ oder „falsch“ bzw. „ein“ oder „aus“)
string	Zeichenketten (Texte) Beispiel: Hallo, das ist eine Zeichenkette.

5.2.3.2 Deklaration und Initialisierung

Für die Deklaration und Initialisierung von Variablen stehen dir unterschiedliche Varianten zur Verfügung, weswegen wir uns hier ein paar Beispiele ansehen werden.

Listing 5.4 zeigt dir beispielhaft, wie du vier verschiedene Variablen deklarierst.

Listing 5.4 Variablendeklaration

```
int enemyCount;
string playerName;
float playerHeight;
bool isFlashlightOn;
```

Möchtest du den Variablen anschließend Werte zuweisen, kannst du das wie in Listing 5.5 machen. Dieses erste Zuweisen von Werten nach der Deklaration nennt man *Initialisieren*. Versuchst du eine Variable zu lesen, bevor sie *deklariert und initialisiert* wurde, resultiert das in einem Fehler. Von der Vorgehensweise her unterscheidet sich das Initialisieren nicht von dem Zuweisen von Werten an anderen Stellen.

Listing 5.5 Initialisieren (Werte zuweisen)

```
enemyCount = 5;
playerName = "Daniel Korgel";
playerHeight = 1.78f;
isFlashlightOn = true;
```

Wenn du einer `string`-Variable einen Wert zuweist, musst du den Wert, wie in Listing 5.5, in Anführungsstrichen schreiben. Damit sagst du dem Compiler, wo der Text anfängt und wo er aufhört.

Wie am Computer üblich, musst du bei `float` für das „Komma“ die amerikanische Schreibweise, also den **Punkt .**, verwenden (z. B. „*1.5f*“ statt „1,5f“). Weist du einer *float*-Variable einen Wert zu, muss hinter der Kommazahl ein `f` für *float* stehen. *C#* unterstützt nämlich noch einen anderen primitiven Datentyp für Kommazahlen, den sogenannten *double*. Er unterscheidet sich vom *float* darin, dass er Zahlen mit einer höheren Genauigkeit speichern kann. Die Verwendung von *doubles* ist in Unity jedoch unüblich, weil *floats* in der Regel genau genug sind und weniger Speicher verbrauchen. Lässt du hinter einer Kommazahl das `f` weg, bedeutet dies für den Compiler, es handelt sich dabei um einen *double*. Weil es nicht möglich ist, einer *float*-Variable einen *double*-Wert zuzuweisen, resultiert das in einem Compiler-Fehler.

Variablen mit dem Typ `bool` kannst du nur einen von zwei Werten zuweisen: `true` oder `false`. Diese Werte stehen für „wahr“ oder „1“ und „falsch“ oder „0“. Auch wenn es so aussieht, weist du der Variable keinen Text zu: Bei *true* und *false* handelt es sich um sogenannte *Schlüsselwörter*, die innerhalb der Programmiersprache eine definierte Bedeutung haben. Diesen Datentyp benutzt du für gewöhnlich, um einfache Ein-/Aus-Zustände zu speichern (zum Beispiel ob eine Taschenlampe derzeit eingeschaltet ist).

Es ist auch möglich, Deklaration und Initialisierung in einer einzigen Zeile vorzunehmen, das sähe dann zum Beispiel aus wie in Listing 5.6. Um Codezeilen zu sparen, ist das die Art und Weise, die in der Praxis am meisten verwendet wird.

Listing 5.6 Deklaration und sofortige Initialisierung

```
int enemyCount = 5;
string playerName = "Daniel Korgel";
float playerHeight = 1.78f;
bool isFlashlightOn = true;
```

5.2.3.3 Datentypen konvertieren

Grundsätzlich lassen sich auch bestimmte Datentypen in andere Datentypen konvertieren. Es ist also durchaus zum Beispiel möglich, einen `int`-Wert einer `float`-Variable zuzuweisen. Dieses Umwandeln von Werten nennt sich *Casten* und wird in zwei Varianten unterschieden: *impliziter Cast* und *expliziter Cast*.

Ein impliziter Cast ist immer dann möglich, wenn mit der Umwandlung kein Datenverlust verbunden ist. Das ist unter anderem immer dann der Fall, wenn der Zieltyp den Ursprungstyp ohne Weiteres darstellen kann. Listing 5.7 zeigt als Beispiel einen impliziten Cast von einem Integer zu einem Float. Da eine Ganzzahl auch als eine Kommazahl dargestellt werden kann, ist das kein Problem.

Listing 5.7 Der implizite Cast funktioniert, da floats auch ganze Zahlen darstellen können.

```
int enemyCount = 5;
float enemyCoutFloat = enemyCount;
```

Andersherum wäre ein expliziter Cast notwendig, da beim Umwandeln von `float` nach `int` ein Datenverlust stattfindet, da Integer keine Kommazahlen darstellen können und diese einfach bei der Typumwandlung verloren gehen Bei einem expliziten Cast von einem *Float* zu einem *Integer* wird die Nachkommastelle einfach abgeschnitten, *es wird nicht gerundet*. Einen expliziten Cast führst du aus, indem du, wie in Listing 5.9, den Zieldatentyp in runden Klammern vor der Variable mit dem Ursprungsdatentyp schreibst. Versuchst du diese Zuweisung ohne einen expliziten Cast, erhältst du einen Compiler-Fehler.

Listing 5.8 Bei Datenverlust ist ein expliziter Cast notwendig.

```
float playerHeight = 1.78f;
int playerHeightInteger = (int)playerHeight; // hat danach den Wert 1
```

Möchtest du einen Wert runden, kannst du die *RoundToInt*-Methode der *Mathf*-Klasse verwenden. Diese Methode übernimmt das Runden und das Casten für dich. In Listing 5.9 siehst du, wie du die Methode verwendest.

Listing 5.9 Umwandlung mit Runden der Werte

```
float playerHeight = 1.78f;
int playerHeightInteger = Mathf.RoundToInt(playerHeight); // hat danach den Wert 2
```

Ein besonderer Fall ist noch das Umwandeln eines Wertes in eine Zeichenkette (`string`). In diesem Fall musst du die *ToString()*-Methode der jeweiligen Variable verwenden. Bei `floats` ist nach einer String-Umwandlung das „*f*“ nicht sichtbar.

Listing 5.10 Einen primitiven Typ in eine Zeichenkette umwandeln

```
float playerHeight = 1.78f;
string playerHeightText = playerHeight.ToString(); // hat danach den Wert "1.78"
```

5.2.3.4 Rechen- und String-Operatoren

Mit Variablen können in C# diverse Rechenoperationen ausgeführt werden. Neben den Grundrechenarten gibt es auch noch ein paar besondere Rechen-Operatoren.

In der folgenden Tabelle siehst du die am meisten verwendeten Operatoren und ihre Funktion:

Tabelle 5.2 Rechenoperationen in C#

Operator	Funktion
+	Addieren
-	Subtrahieren
/	Dividieren
*	Multiplizieren
%	Modulo (Rest einer Ganzzahldivision)
++	Wert um 1 erhöhen
--	Wert um 1 verringern

Wie du in Listing 5.11 siehst, funktioniert das Rechnen so, wie man es sich vorstellt:

Listing 5.11 Ein paar Rechnungen

```
int arrows = 5 - 2; // hat nach der Rechnung den Wert 3
int swords = 22;
int totalItems = arrows + swords; // hat nach der Rechnung den Wert 25
```

Wenn du eine Variable basierend auf ihrem aktuellen Wert verändern möchtest, hast du zwei Möglichkeiten, dies zu tun, welche zum gleichen Ergebnis führen. Listing 5.12 zeigt dir sowohl die klassische als auch die Kurzschreibweise. Die kurze Schreibweise kannst du, bis auf für die „++“- und „--“-Operatoren, für alle Operatoren aus der Tabelle verwenden.

Listing 5.12 Beispiel für die Kurzschreibweise

```
swords = swords + 5; // Lange Schreibweise
swords += 5;         // Kurzform
```

Zu den beiden verbleibenden Operatoren kommen wir jetzt. Möchtest du eine Variable um 1 erhöhen oder verringern, geht es noch kürzer als mit der Kurzform. Listing 5.13 zeigt dir, wie du den „--“- und den „++“-Operator verwendest.

Listing 5.13 Beispiel für die ++- und --Operatoren

```
arrows--; // verringert arrows um 1
swords++; // erhöht swords um 1
```

`arrows++` ist also äquivalent zu `arrows += 1` und auch zu `arrows = arrows + 1`.

Es können natürlich auch kompliziertere Rechnungen mit Klammern und gemischten Operationen durchgeführt werden. In C# gilt wie sonst auch: Punkt- vor Strich-Rechnung.

Listing 5.14 Beispiel für eine komplexe Rechnung

```
float result = (5 + 3) * 4 / (1 + 1) + 1; // Ergebnis ist 17
```

Dividieren mit Integern

Wenn du mit *Integern* (`int`) rechnest, gibt es jedoch eine Besonderheit, wenn du dividieren möchtest. Da Integer nur Ganzzahlen unterstützen, wird auch die Rechnung in Ganzzahlen ausgeführt, wenn sowohl Dividend als auch Divisor *Integer* sind. Das geschieht, selbst wenn deine Zielvariable vom Typ `float` sein sollte, weil die Rechnung vor der Zuweisung ausgeführt wird. Es muss mindestens einer der beiden Operanden ein *Float* sein, damit die Rechenoperation inklusiv Dezimalstellen ausgeführt wird.

In Listing 5.15 findest du ein paar Rechenbeispiele mit Lösung, die dich vielleicht überraschen.

Listing 5.15 Beispiel zum Dividieren mit Integern

```
int quotientA  = 5  / 2;   // Ergebnis ist 2, Rest 1 wird verworfen
float quotientB = 5  / 2;   // Ergebnis ist 2, Rest 1 wird verworfen
float quotientC = 5f / 2;   // Ergebnis ist 2.5f
float quotientD = 5  / 2f;  // Ergebnis ist 2.5f
int quotientE  = 5f / 2;   // Fehler (siehe unten)
```

Bei der letzten Rechnung in Listing 5.15 (*quotientE*) würdest du eine Fehlermeldung erhalten, da das Ergebnis der Rechenoperation ein *Float* ist und es nicht möglich ist, einen *Float* implizit in einen *Integer* zu casten.

In diesem Zusammenhang ist der Modulo-Operator dann auch interessant, da er den Rest einer Division mit ganzen Zahlen berechnen kann.

Listing 5.16 Beispiel für den Modulo-Operator

```
int result = 5 / 2; // Ergebnis ist 2, Rest 1 wird verworfen
int rest   = 5 % 2; // Ergebnis ist 1, da 5 / 2, 1 Rest hat
```

Der Modulo-Operator wird sehr häufig verwendet, um zu prüfen, ob eine bestimmte Zahl gerade oder ungerade ist. Dabei prüft man einfach, ob das Ergebnis einer ganzzahligen Division der jeweiligen Zahl durch 2 einen Rest hat oder nicht. Listing 5.17 zeigt dir, wie das aussehen könnte. Dazu wird auch ein Vergleichsoperator verwendet, der das Ergebnis der Modulo-Rechnung mit *0* vergleicht und einen *Boolean* zum Ergebnis hat (`true` oder `false`).

Listing 5.17 Prüfen, ob eine Zahl gerade oder ungerade ist mit Modulo:

```
bool isEvenA   = (2016 % 2)==0; // Zahl ist gerade da Ergebnis 0
bool isEvenB   = (5697 % 2)==0;  // Zahl ungerade da Ergebnis 1 (und 1!=0 ist).
```

String-Operatoren

Die zuvor erklärten Operatoren funktionieren nur mit Datentypen, welche Zahlen repräsentieren, allerdings nicht mit Zeichenketten (`string`). Der +-Operator doubelt jedoch auch als *Konkatenations-Operator*. Das bedeutet, mit ihm kannst du mehrere *Strings* miteinander verbinden.

Die Verwendung ist denkbar einfach, wie in Listing 5.18 zeigt.

Listing 5.18 Strings miteinander verbinden

```
string aString = "Hello" + " World"; // aString hat danach den Wert "Hello World"
string aDifferentString = "!!!!!";
string result = "My Message is" +" "+ aString + " "+ aDifferentString;
```

Die Variable `result` hätte nach dem Zuweisen den Wert: *„My Message is Hello World !!!!!“*

5.2.3.5 Arrays

Arrays sind sogenannte Datenfelder, eine sehr einfache Variante einer Liste. Arrays sind dann praktisch, wenn man mehrere Variablen eines Typs benötigt, ohne dass man mit vielen einzelnen Namen umgehen möchte.

Im Prinzip sind *Arrays* Listen, die eine bestimmte Anzahl von Variablen eines bestimmten Datentyps speichern können. Nach der Initialisierung des Arrays kann seine *Länge* nicht mehr geändert werden. Die einzelnen *Werte* in dem Array können allerdings weiterhin ausgetauscht werden.

Wenn du einen Array *deklarieren* möchtest, musst du hinter dem *Datentyp* ein Paar *eckige Klammern* schreiben, wie du in Listing 5.19 siehst.

Listing 5.19 Array-Deklaration

```
int[] itemSlots;
```

Die *Initialisierung* eines Arrays erfolgt über das `new`-Schlüsselwort. Hinter das „new“ schreibst du dann anschließend nochmals den Datentyp und die gewünschte Größe der Liste in eckigen Klammern. Listing 5.20 zeigt, wie das für den Array aus dem Listing 5.19 aussehen könnte.

Listing 5.20 Initialisierung eines Arrays mit fünf Feldern

```
itemSlots = new int[5];
```

Bei dieser Art der Initialisierung werden alle Felder mit einem *Standardwert* initialisiert. Bei Zahlen ist das „0“, bei Strings ist das eine leere Zeichenkette „“.

Du kannst bei der Initialisierung aber auch direkt eigene Werte angeben, die für die einzelnen Einträge verwendet werden sollen. Ein Beispiel dafür siehst du in Listing 5.21.

Listing 5.21 Initialisierung eines Arrays mit fünf Feldern und eigenen Werten

```
int[] itemSlots = new int[5]{2,50,400,60,32};
```

Wenn du auf ein Element eines Arrays zugreifen möchtest, musst du einfach die entsprechende Array-Variable mit dem gewünschten Index ansprechen. Der Index beginnt, wie in der Softwareentwicklung üblich, mit *0*.

Listing 5.22 Auf Array-Elemente zugreifen

```
int[] itemSlots = new int[5]{2,50,400,60,32};
itemSlots[0] = 4; // Element 0 war vorher 2, ist jetzt 4
itemSlots[3] = itemSlots[4]; // Element 3, war vorher 400,
                             // ist jetzt 60 (Wert von Element 4)
```

Bei komplexeren Projekten wirst du auch mit Arrays zu tun haben, deren Länge du nicht kennst; zum Beispiel weil die Array-Länge abhängig von bestimmten Eingaben des Spielers ist. Für diese Fälle bieten dir Arrays eine Eigenschaft mit dem Namen `Length`. Auf Eigenschaften greifst du zu, indem du hinter den Variablennamen einen Punkt `.` schreibst, gefolgt von dem Namen der Eigenschaft; also zum Beispiel `variabelName.Eigenschaft`. Listing 5.23 zeigt, wie du die Länge des itemSlots-Arrays aus den vorherigen Beispielen auslesen kannst. Der Wert, den du von der Eigenschaft erhältst, ist ein *Integer*.

Listing 5.23 Array-Größe auslesen

```
int lengthOfItemSlots = itemSlots.Length;
```

Hierbei musst du beachten, dass die Länge des Arrays immer um eins höher ist als der höchste Index in dem Array. Das liegt daran, dass das erste Element im Array den Index „*0*“ hat. Hast du zum Beispiel ein Array mit fünf Einträgen, gibt dir die `Length`-Eigenschaft den Wert *5* zurück. Der höchste Index im Array ist aber *4* (der Array hätte die fünf Einträge: 0, 1, 2, 3, 4). Demnach erhältst du den letzten Eintrag, ohne die Länge des Arrays zu kennen, immer über `Length-1`.

Listing 5.24 Das letzte Element eines beliebigen Arrays erhalten

```
int value = itemSlots[itemSlots.Length-1];
```

5.2.4 Methoden

Wenn du ein Script schreibst, kommt es ständig vor, dass du dieselben Zeilen Code immer wieder an verschiedenen Stellen in deinem Script benötigst. Methoden erlauben dir, diese immer wiederkehrenden Codezeilen in einer Gruppe zusammenzufassen und mit einem Namen zu versehen. Wenn du die Codezeilen jetzt verwenden möchtest, rufst du einfach die Methode über ihren Namen auf und musst nicht mehr die Codezeilen in dein Script kopieren. Neben dem reduzierten Speicherbedarf hat dieses Vorgehen auch den Vorteil, dass man eventuelle Änderungen nur an einer zentralen Stelle vornehmen muss und nicht überall durch das Script verteilt.

Methoden bestehen aus einem *Methodenkopf*, welcher den *Rückgabetyp*, ihren *Namen* und ihre Parameter *deklariert*. Darunter befindet sich der *Methodenrumpf*, welcher die Logik der der Methode *implementiert*.

In Listing 5.25 hat die Methoden den Rückgabetyp `void`, den Namen *sayHello* und keine Parameter. Der Datentyp `void` gibt an, dass diese Methode *keinen* Wert zurückgibt.

Listing 5.25 Beispiel für eine Methode

```
string message = "";

void SayHello(){  //Methodenkopf
    message = "hello";  //Methodenrumpf
}
```

Immer wenn diese Methode aufgerufen wird, wird der Variablen `message` der Wert *hello* zugewiesen. Der Aufruf dieser Methode würde wie in Listing 5.26 aussehen.

Listing 5.26 Beispielhafter Aufruf einer Methode

```
    SayHello();
```

Parameter

Als Nächstes schauen wir uns eine Methode mit Parametern an. Die Parameter werden mit Datentyp und einem eindeutigen Variablennamen in den runden Klammern des *Methodenkopfes* deklariert. Wenn eine Methode mehrere Parameter hat, werden diese durch ein *Komma* **,** getrennt. Listing 5.27 zeigt, wie eine Methode mit zwei Parametern aussehen könnte. Die Parameter können innerhalb der Methode wie normale Variablen verwendet werden.

Listing 5.27 Beispiel für eine Methode mit Parametern

```
float result = 0;
void AddValues(float a, int b){
    result = a + b;
}
```

Listing 5.28 zeigt, wie der Aufruf der Methode in Listing 5.27 aussehen könnte.

Listing 5.28 Beispiel Methodenaufruf mit Rückgabewert

```
    AddValues(5, 3); // result hat danach den Wert 8
```

Rückgabetypen

Zuletzt schauen wir uns eine Methode mit einem Rückgabewert an. Gibst du im Methodenkopf einen anderen Rückgabetyp als `void` an, musst du mit dem Schlüsselwort `return` am Ende der Methode einen passenden Wert zurückgeben. Für die `Add Values` könnte das aussehen wie in Listing 5.29.

Listing 5.29 Beispiel für eine Methode mit Parameter und Rückgabewert

```
float AddValues(float a, int b){
    float calculation = a + b;
    return calculation;
}
```

Im Beispiel in Listing 5.29 werden die beiden Werte von *a* und *b* addiert und in der Variable *calculation* gespeichert. Zum Schluss wird der Inhalt der Variable *calculation* zurückgegeben. Wenn du die Methode jetzt aufrufst, musst du den Rückgabewert einer Variable zuweisen, um ihn zu speichern. Listing 5.30 zeigt dir, wie das aussehen würde.

Listing 5.30 So rufst du die AddValues-Methode auf und speicherst das Ergebnis.

```
float result = AddValues(5, 3);
```

Beim Aufrufen der Methode kannst du sowohl konkrete Werte als auch Variablen als Parameter anwenden. Bei primitiven Datentypen wird dann immer der Inhalt der Variable als Parameter übergeben. Listing 5.31 zeigt, wie ein Aufruf mit verschiedenen Variablen aussehen würde.

Listing 5.31 Beispielaufrufe der addValues-Methode mit Variablen

```
float result = 0;
int first = 1;
int second = 2;

result = AddValues(first, 3); //result hat danach den Wert 4 (1+3)
result = AddValues(result, second); //result hat danach den Wert 6 (4+2)
```

Methodennamen

Wie bei Variablen dürfen in Methodennamen keine Leerzeichen, Sonderzeichen (Ausnahme auch hier der Unterstrich „_“) und Umlaute enthalten sein. Zusätzlich existieren folgende Konventionen zur Benennung einer Methode, an die du dich halten solltest:

- Der Methodenname sollte die Funktion der Methode aussagekräftig beschreiben und mit einem *Verb* beginnen

  z. B. „*GetComponent*“, statt „GC“ oder „Component“
- Methodennamen sollten in *Pascal Case* geschrieben werden. Das bedeutet, sie sollten mit einem *Großbuchstaben* beginnen; besteht der Name aus mehreren Wörtern, schreibst du den ersten Buchstaben jedes weiteren Worts ebenfalls mit einem Großbuchstaben

  z. B. „*CalculateComplexMathematics*“ statt „*calculateComplexMathematics*“, „*Calculate_Complex_Mathematics*“ oder „*calculatecomplexmathematics*“
- Methodennamen sollten keine Zahlen enthalten

  z. B. „*MultiplayByThree*“ statt „*MultiplayBy3*“.

Methodennamen müssen zusammen mit ihrer Parameter-Kombination eindeutig sein. Das bedeutet, du kannst zwei Methoden mit dem gleichen Namen haben, wenn die Datentypen der Parameter unterschiedlich sind. Man spricht dabei von einer *Überlagerung*. Rufst du eine überlagerte Methode auf, wird basierend auf den von dir angegebenen Parametern entschieden, welche Variante ausgeführt wird.

Listing 5.32 Beispiel für eine überlagerte Methode

```
int Combine(int a, int b) { // Addiert a und b
    return a + b;
}
```

```
string Combine(string a, string b) { // Fügt a und b mit einem Leerzeichen zusammen
    return a + " " + b;
}
```

Rufst du die `Combine`-Methode mit *String*-Parametern auf, wird automatisch die String-Variante ausgeführt. Gibst du als Parameter *Integer* an, wird automatisch die Integer-Variante ausgewählt. Listing 5.33 zeigt dir verschiedene, korrekte Aufrufe der überlagerten Methode.

Listing 5.33 Aufrufen einer überlagerten Methode

```
int resultA = Combine(5, 3); // Ergebnis ist 8
string resultB = Combine("Hello", "World"); // Ergebnis ist "Hello World"
string resultC = Combine("5", "3"); // Ergebnis ist "5 3"
```

5.2.5 Klassen

Bei objektorientierten Programmiersprachen fasst man Variablen und Eigenschaften thematisch passend in sogenannten *Klassen* zusammen. Das bedeutet im Prinzip nur, dass beispielweise alle *Variablen* und *Methoden*, die zum Beispiel Gegner betreffen, sich in einer Klasse *Enemy* befinden.

Klassennamen sollten nach Konvention immer mit einem Großbuchstaben beginnen (Pascal Case) und müssen ebenfalls eindeutig sein.

Listing 5.34 Eine Beispielklasse „Enemy“

```
class Enemy {
    int health = 100;
    int strength = 5;

    void TakeDamage(int damage){
        health -= damage;
    }
}
```

Die Klasse in Listing 5.34 beschreibt nur, welche Variablen und Methoden ein Gegner hat, und stellt noch keinen konkreten Gegner dar. Dazu musst du zunächst ein *Objekt* der Klasse erzeugen. Das funktioniert wie mit dem *new*-Schlüsselwort:

Listing 5.35 Ein Objekt der Enemy-Klasse erzeugen

```
Enemy enemyKlaus = new Enemy();
```

Die Variable `enemyKlaus` enthält jetzt ein *Objekt* der *Klasse* `Enemy`.

Klassen sind so gesehen Bauanleitungen, wie etwas auszusehen hat. Mit dem *new*-Schlüsselwort wird dann nach dieser Anleitung ein konkretes Objekt gebaut. Du kannst natürlich auch mehrere Objekte nach dieser Anleitung bauen lassen, deren Eigenschaften dann unabhängig voneinander sind.

Listing 5.36 Erzeuge zwei voneinander unabhängige Objekte der Enemy Klasse

```
Enemy enemyKlaus = new Enemy();
Enemy enemyFred = new Enemy();

enemyKlaus.TakeDamage(20); // enemyKlaus.health ist jetzt 80
enemyFred.TakeDamage(50); // enemyFred.health ist jetzt 50
```

Obwohl beide Variablen Objekte der Klasse Enemy sind, agieren die beiden vollkommen unabhängig voneinander, sie wurden lediglich nach derselben Bauanleitung angelegt. Nimmt *enemyKlaus* Schaden, hat dies keine Auswirkung auf *enemyFred* und umgekehrt.

5.2.6 Dynamische Datentypen

Variablen, die für Objekte von Klassen vorgesehen sind, verwenden sogenannte *dynamische Datentypen*. Man bezeichnet sie als dynamisch, weil der Datentyp (die Klasse) ebenfalls als Quellcode definiert ist, während *primitive Datentypen* zur Programmiersprache gehören. Dynamische Datentypen unterscheiden sich darin von den primitiven Datentypen, dass sie in C# standardmäßig als *Verweis* übergeben werden. Was das bedeutet, zeigt das Beispiel in Listing 5.37:

Listing 5.37 Beispiel für Besonderheiten von dynamischen Datentypen

```
Enemy enemyKlaus = new Enemy();
Enemy enemyFred = enemyKlaus;
enemyKlaus.TakeDamage(20); // enemyKlaus.health und enemyFred.health sind jetzt 80
enemyFred.TakeDamage(50);  // enemyKlaus.health und enemyFred.health sind jetzt 30
```

Wenn du, wie im Beispiel, *enemyFred* den Wert von *enemyKlaus* zuweist, zeigen *enemyFred* und *enemyKlaus* danach auf **dasselbe Objekt**. Es wird nicht, wie man erwarten könnte, eine Kopie von *enemyKlaus* angelegt und *enemyFred* zugewiesen.

Basierend auf diesem Wissen kannst du Methoden schreiben, welche Objekte verändern. Denn auch wenn du Objekte als Parameter übergibst, wird ein *Verweis* auf das Objekt und keine *Kopie* übergeben:

Listing 5.38 Beispiel für eine Methode mit einem dynamischen Datentyp als Parameter

```
void AttackEnemy(Enemy target, int damage){
    target.TakeDamage(damage);
}
```

Führt man die Methode aus Listing 5.38 aus, wird die `TakeDamage`-Methode des übergebenen Enemy-Objekts ausgeführt. Listing 5.39 demonstriert, wie die Methode verwendet werden könnte.

Listing 5.39 Beispiel für Aufrufe der Methode AttackEnemy

```
Enemy enemyKlaus = new Enemy();

AttackEnemy(enemyKlaus, 20);
//enemyKlaus.health ist jetzt 80
```

```
AttackEnemy(enemyKlaus, 25);
//enemyKlaus.health ist jetzt 55
```

Wenn du also Variablen mit einem dynamischen Datentyp übergibst, übergibst du immer einen Verweis auf genau das Objekt und alle Änderungen wirken sich deswegen auch auf das „ursprüngliche“ Objekt aus.

Primitive und dynamische Datentypen erkennen:

Primitive Datentypen beginnen immer mit einem Kleinbuchstaben, außerdem sind es einfache Typen, die nur eine Zahl oder einen Text speichern können (zum Beispiel: *float, int, string bool etc.*).

Dynamische Datentypen beginnen mit einem Großbuchstaben, da per Konvention Klassennamen mit einem Großbuchstaben beginnen sollten. Außerdem beschreiben sie meist komplexe Objekte, die sich aus mehreren primitiven Datentypen und Methoden zusammensetzen (zum Beispiel: *Enemy, Material, List, Player* etc.).

5.2.6.1 Ein Verweis

Da du in Unity ständig mit Verweisen (auch „Referenzen“ genannt) arbeiten wirst, möchte ich genauer darauf eingehen. Technisch gesehen übergibst du bei einem Verweis eine Speicheradresse, die auf die Stelle zeigt, wo das jeweilige Objekt liegt. Praktisch kannst du es dir so ähnlich vorstellen: Durch einen Verweis erkennt ein Script, wo es ein bestimmtes Objekt findet, und kann damit arbeiten. Verändert das Script das Objekt, müssen alle anderen ebenfalls mit den Änderungen zurechtkommen. Übergibst du einen Verweis an eine Methode, spricht man davon, dass das Objekt *By Reference* übergeben wurde. Die Alternative, die bei primitiven Datentypen standardmäßig verwendet wird, nennt sich *By Value*.

By Reference Beispiel: Bob fragt Anna nach einem Stift. Anna sagt zu Bob, dass in ihrer Schublade ein Bleistift liegt, den er verwenden kann (Anna übergibt Bob einen Verweis auf ihren Bleistift). Bob verwendet den Bleistift und dabei bricht die Spitze ab. Wenn Anna jetzt ihren Bleistift verwenden will, muss sie ihn zuerst anspitzen.

By Value Beispiel: Bob fragt Anna nach einem Stift. Anna besitzt einen Bleistift, weil sie aber Angst hat, dass Bob ihn kaputt macht, fertigt sie einen neuen, identischen Bleistift an und gibt ihn Bob. Wenn jetzt einer der beiden Bleistifte kaputtgeht, bleibt der jeweils andere davon unangetastet.

Für beide Arten gibt es Fälle, in denen sie jeweils Sinn machen. Für den Anfang verwenden wir alle Datentypen aber so, wie es von C# vorgesehen ist: primitive Datentypen mittels „By Value“ und dynamische Datentypen mittels „By Reference“.

5.2.6.2 Der Wert „null“

Bei dynamischen Datentypen gibt es keinen sinnvollen Standardwert, der eine deklarierte Variable hat, bevor ihr erstmals ein Wert zugewiesen wird. Zu diesem Zeitpunkt ist also kein Verweis vorhanden und der Wert der Variablen ist *leer*. In der Programmierung wird

dieser leere Zustand mit dem Schlüsselwort `null` bezeichnet. Umgangssprachlich sagt man, die Variable hat den Wert `null`.

Versuchst du, auf eine Variable zuzugreifen, deren Wert `null` ist, erzeugt das einen Fehler und in deiner *Console* wirst du eine Fehlermeldung mit dem Namen *Null Reference Exception* lesen können.

5.2.7 Statische Methoden und statische Variablen

Statische Methoden und Variablen sind besondere Elemente der objektorientierten Programmierung, welche sich nicht auf ein bestimmtes Objekt, sondern auf die Klasse selbst beziehen. Wenn eine Variable statisch ist, teilen sich alle Objekte der Klasse diese Variable. Weist Objekt `A` der Variablen einen neuen Wert zu, ändert sich dieser Wert so gesehen auch in allen anderen Objekten der Klasse. Das liegt daran, dass alle Objekte dieselbe Variable verwenden.

„Static" im Unity Inspector

Statische Methoden und Variablen haben nichts mit der *Static*-Eigenschaft im Inspector zu tun!

Statische Variablen erkennst du an dem Schlüsselwort `static` vor dem Datentyp. Hat eine statische Variable wie in Listing 5.40 einen Initialisierungswert, wird dieser ausschließlich bei Programmstart gesetzt, nicht, wenn du ein neues Objekt der Klasse anlegst.

Listing 5.40 Beispiel für eine statische Variable

```
class Enemy {
    static int enemyCount = 0;
    void RiseCounter(){
        enemyCount+=1;
    }
}
```

Innerhalb der Klasse kannst du auf die statische Variable wie auf eine ganz normale Variable zugreifen. Wenn du von einer anderen Klasse aus auf die Variable zugreifen möchtest, machst du das nicht über ein Objekt der Klasse, sondern über den Klassennamen; zum Beispiel, wie in Listing 5.41, mit `Enemy.enemyCount`.

Listing 5.41 Ein Beispiel für die Verwendung der statischen Variablen

```
Enemy enemyMax = new Enemy();
enemyMax.RiseCounter(); // Enemy.enemyCount ist jetzt 1
Enemy enemySteve = new Enemy();
enemySteve.RiseCounter(); // Enemy.enemyCount ist jetzt 2
```

Ähnlich wie statische Variablen werden statische Methoden meist als Helfer verwendet, die sich nicht auf ein bestimmtes Objekt beziehen, sondern generelle Aufgaben erledigen.

Innerhalb von statischen Methoden kannst du ausschließlich auf ebenfalls statische Variablen zugreifen (das beinhaltet alle Variablen, die innerhalb der statischen Methode angelegt werden). Um auf Objekt-Variablen zugreifen zu können, benötigst du einen Verweis auf ein spezifisches Objekt.

Listing 5.42 Beispiel für eine statische Methode, welche neue Gegner erzeugt

```
class EnemyFactory {
    static int createdEnemies = 0;
    static Enemy BuildEnemy(){
        createdEnemies+=1;
        Enemy randomEnemy = new Enemy();
        // Code, der randomEnemy z.B. zufällige Stärke und Lebensenergie zuweist
        return randomEnemy;
    }
}
```

Listing 5.42 zeigt eine Gegner-Fabrik, welche eine statische Methode zum Erstellen neuer Gegner besitzt. Möchtest du einen neuen Gegner mit einer zufälligeren Stärke und Lebensenergie erstellen, musst du einfach nur noch `EnemyFactory.BuildEnemy()` aufrufen und erhältst als Rückgabewert ein fertig konfiguriertes Objekt vom Typ `Enemy`. Du musst vorher kein Objekt von der *EnemyFactory*-Klasse anlegen.

5.2.8 Vererbung

Ein wichtiges Konzept von objektorientierter Programmierung ist *Vererbung*. Das Konzept ist deswegen sehr wichtig, da es hilft, deine Klassen übersichtlich und gut wartbar zu halten. Bei dem Konzept der Vererbung wird versucht, doppelten Code zu verhindern, indem eine Eltern-Klasse all ihre Methoden und Variablen einer anderen Klasse als Basis zur Verfügung stellt.

Wenn du beispielweise ein Auto *Rennwagen* und ein Auto *Pickup Truck* hast, würdest du alles, was beide Autos gemeinsam haben, in eine eigene Klasse *Fahrzeug* schreiben. Die *Fahrzeug*-Klasse würde dann in diesem Beispiel sämtlichen Code zum Fahren des Fahrzeugs enthalten. Außerdem würde die Klasse alle Variablen enthalten, die alle Fahrzeuge gemeinsam haben, auch wenn die konkreten Werte je nach Fahrzeug variieren können. Listing 5.43 zeigt, wie diese Fahrzeug-Klasse (englisch „Vehicle“) aussehen könnte.

Listing 5.43 Beispiel für eine Vehicle-Klasse

```
public class Vehicle {
    int currentSpeed = 0;           // Aktuelle Geschwindigkeit
    int seats = 4;                  // Anzahl der Sitze
    public void Steer(){            // Lenken
         // Code
    }
    public void Accelerate(){      // Beschleunigen
        currentSpeed += 5;
    }
}
```

Die Klassen für den *Rennwagen* und den *PickupTruck* erben dann von der Klasse `Vehicle` und erhalten dadurch alle *Variablen* und *Methoden* dieser Klasse, die nicht als *private* markiert sind.[1] Die Eltern- und Kind-Klasse werden also kombiniert. Dadurch musst du nicht in jeder der Klassen die Methoden `Steer()` und `Accelerate()` neu schreiben, sondern hast sie an einer einzigen Stelle. Die Kind-Klassen können dann noch eigene Methoden und Variablen enthalten, die nur sie betreffen.

Wenn eine Klasse von einer anderen Klasse erben soll, schreibt man die Eltern-Klassen, getrennt durch einen Doppelpunkt „**:**“, hinter den Klassennamen.

Listing 5.44 Beispiel für eine Klasse *PickupTruck*, die von *Vehicle* erbt

```
public class PickupTruck : Vehicle {
    int itemsOnTruck = 0;
    public void LoadAnItem(){
        itemsOnTruck ++;
    }
}
```

Listing 5.44 zeigt eine Klasse `PickupTruck`, welche von der Klasse `Fahrzeug` erbt. Zusätzlich implementiert sie selbst noch eine Methode zum Beladen der Ladefläche. Insgesamt stellt dir die Klasse `PickupTruck` also drei Methoden (`Steer()`, `Accelerate()` und `LoadAnItem()`) und drei Variablen (`currentSpeed`, `seats`, `itemsOnTruck`) zur Verfügung, die du auf Objekten der Klasse *PickupTruck* ausführen kannst.

Über *implizite Casts* kannst du Verweise auf `PickupTruck`-Objekte auch in einer `Vehicle`-Variablen speichern, da jeder `PickupTruck` auch ein `Vehicle` ist. Andersherum ist das nur über einen expliziten Cast möglich, da nicht jedes Fahrzeug ein *PickupTruck* ist. Schlägt der Cast fehl, erhältst du zur Laufzeit eine Fehlermeldung in der *Console*.

Listing 5.45 Implizite und explizite Umwandlung in Eltern- bzw. Kind-Klasse

```
PickupTruck myPickup = new PickupTruck();
Vehicle myVehicle = myPickup;                    // Implizit möglich
Vehicle yourVehicle = new Vehicle();
PickupTruck yourPickup = (PickupTruck)yourVehicle;  // Nur explizit möglich
```

5.2.8.1 Methoden überschreiben

In manchen Fällen ist es notwendig, dass eine Kind-Klasse eine Methode aus der Eltern-Klasse überschreibt, weil das Kind die Methode anders implementieren soll. In unserem Beispiel soll die Klasse `Rennwagen` eine eigene `Accelerate()`-Methode implementieren, welche den Rennwagen schneller beschleunigen lässt als ein normales Fahrzeug.

Damit man eine Methode überschreiben kann, muss zu der entsprechenden Methodendeklaration in der Eltern-Klasse das Schlüsselwort `virtual` hinzugefügt werden, wie Listing 5.46 zeigt.

[1] Mehr zu den *Zugriffsmodifizierern* „private“ und „public“ findest du in Kapitel 5.2.11.

Listing 5.46 Bei dieser Variante der Vehicle-Klasse kann die Accelerate-Methode wenn gewünscht überschrieben werden.

```
public class Vehicle {
    public int currentSpeed = 0;
    int seats = 4;
    public void Steer(){
         // Code
    }
    public virtual void Accelerate(){      // Virtual, kann überschrieben werden
        currentSpeed += 5;
    }
}
```

In der Kind-Klasse kann jetzt die gleiche Methode deklariert werden, allerdings mit dem Schlüsselwort `override` anstelle von `virtual`, wie Listing 5.47 zeigt.

Listing 5.47 Die Klasse RaceCar mit einer eigenen Accelerate-Methode

```
public class RaceCar : Vehicle {
    public override void Accelerate (){
        currentSpeed += 25; // höhere Beschleunigung
    }
}
```

Wenn du jetzt die `Accelerate`-Methode aufrufst, wird die Implementierung aus der `Rennwagen`-Klasse, anstelle der aus der `Vehicle`-Klasse, aufgerufen. Alle anderen geerbten Methoden und Variablen bleiben unverändert.

5.2.8.2 Polymorphie

Polymorphie ist eine besondere Eigenschaft der *Vererbung* in objektorientierten Programmiersprachen. Lass dich von dem Begriff nicht abschrecken, im Moment bedeutet das nur, wie bereits erwähnt, dass jeder `PickupTruck` und jedes `RaceCar` auch ein `Vehicle` sind. Das bedeutet, dass du sowohl `PickupTruck`-Objekte als auch `RaceCar`-Objekte in einer `Vehicle`-Variablen speichern kannst. Dank der Polymorphie kannst du alle Methoden und Variablen, die in der Vehicle-Klasse deklariert sind, verwenden, ohne dass du weißt, ob es sich dabei um einen `PickupTruck` oder ein `RaceCar` handelt.

Listing 5.48 und Listing 5.49 zeigen dir beispielhaft, wie du auf diese Weise Fahrzeuge steuern kannst, ohne sie genau zu kennen, da sich Klassen, die von `Vehicle` erben, gleich steuern lassen.

Listing 5.48 Eine Methode, die alle Fahrzeuge beschleunigen kann

```
public void AccelerateVehicle(Vehicle vehicle){
    vehicle.Accelerate();
}
```

Listing 5.49 Verwendung der Methode AccelerateVehicle()

```
PickupTruck pickupTruck = new PickupTruck();
RaceCar raceCar = new RaceCar ();
AccelerateVehicle(pickupTruck);
AccelerateVehicle(raceCar);
```

5.2.9 Namespaces

Wie du nun schon weißt, müssen Klassennamen immer eindeutig sein. Zieht man in Betracht, dass *.NET* selbst schon Hunderte Klassen besitzt und durch *Unity* noch einige Hundert dazukommen, kann es schon schwierig werden, einen guten Namen zu finden, der noch frei ist. Vor allem ist das der Fall, wenn man seinen Code eventuell noch mit anderen Entwicklern teilen möchte, denn dann müssen die eigenen Klassennamen auch kompatibel zu deren Klassennamen sein. Um dieses Problem zu vereinfachen, gibt es *Namespaces*.

Mehrere Klassen können in einen *Namespace* zusammengefasst werden und müssen dann nur innerhalb ihres Namespace einen eindeutigen Namen besitzen. Im Prinzip kann man sich Namespaces wie Ordner vorstellen. Für besonders große Projekte ist es auch möglich, Namespaces innerhalb von Namespaces zu erstellen, um die Klassen noch feiner zu kategorisieren.

Alle Unity-Klassen befinden sich zum Beispiel in dem Namespace *UnityEngine*. Dieser Namespace enthält unter anderem den Namespace *VR*, worin sich alle Klassen befinden, die sich mit Unitys VR-Unterstützung beschäftigen.

Wenn du auf eine Klasse innerhalb eines Namespace zugreifen möchtest, machst du das mithilfe des *Punktes* „.“. Um beispielsweise auf die Klasse *VRSettings* zugreifen zu können, würdest du eingeben: *UnityEngine.VR.VRSettings*.

Diese Schreibweise nimmt in deinem Script jedoch viel Platz ein, weshalb sie nur in Ausnahmefällen verwendet wird. Das kann zum Beispiel sein, wenn du nur einmal in deinem ganzen Script auf eine Klasse des jeweiligen Namespace zugreifen musst. Häufiger werden Namespaces auf die Weise verwendet, die in dem folgenden Abschnitt beschrieben wird.

5.2.9.1 Verwendung von Namespaces vereinfachen

Wenn du in deinem Script Klassen aus bestimmten Namespaces sehr häufig verwendest, kann es lästig werden, dass du den entsprechenden Namespace jedes Mal vor den Klassennamen schreiben musst. Deswegen bietet dir C# sogenannte `using`-Anweisungen.

Mit diesen Anweisungen kannst du am Anfang deines Scripts festlegen, welche Namespaces du in dem *Script* verwenden möchtest. Dadurch weiß der Compiler dann, dass du dich auf Klassen aus diesen Namespaces beziehst, und du musst den jeweiligen Namespace nicht mehr vor den Klassennamen schreiben. Standardmäßig wirst du bei Unity-Scripten zum Beispiel immer den *UnityEngine*-Namespace verwenden, weswegen sich bei allen Scripten ganz oben die Anweisung aus Listing 5.50 finden lässt.

Listing 5.50 Beispiel für eine using-Anweisung

```
using UnityEngine;
```

Aber keine Angst, während du über das Konzept von Namespaces Bescheid wissen solltest, musst du in den meisten Fällen nicht wissen, welche Klasse sich in welchem Namespace befindet, um sie benutzen zu können. Wenn du eine Klasse verwendest, die in dem aktuellen Kontext noch nicht gefunden wurde, weil keiner der verwendeten Namespaces sie implementiert, schlägt Visual Studios dir automatisch alle Namespaces vor, in denen eine Klasse mit diesen Namen vorkommt.

Zunächst markiert Visual Studio den Klassennamen als einen Konflikt und unterstreicht ihn rot. Zeigst du dann mit deiner Maus auf das unterstrichene Wort oder klickst auf die Glühbirne links, öffnet sich die Konfliktlösung. In der Konfliktlösung werden dir alle Namespaces angezeigt, die eine Klasse mit dem verwendeten Namen enthalten. Wird kein passender Namespace gefunden, schlägt Visual Studio dir vor, die Klasse selbst zu erstellen. Wird die Klasse in einem Namespace gefunden, sieht es aus wie in Bild 5.1. Du kannst dann den Namespace als `using`-Anweisung hinzufügen oder den Namespace automatisch vor den Klassennamen einfügen lassen.

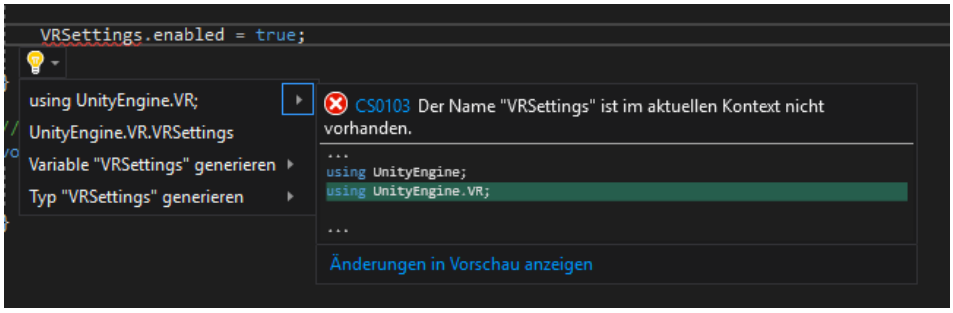

Bild 5.1 Visual Studio schlägt dir automatisch passende Namespaces vor.

5.2.10 Lebenszyklus von Variablen

Der Lebenszyklus einer Variable beschreibt, wo die Variable gültig ist, wo du auf sie zugreifen kannst und wo du keine Variablen mit dem gleichen Namen verwenden darfst.

Grundsätzlich kannst du dir merken, Variablen existieren immer in ihrem jeweiligen Codeblock, der durch Schweifklammern begrenzt ist. Schreibst du einen Codeblock innerhalb eines anderen Codeblocks, existieren auch die Variablen des übergeordneten Codeblocks, weshalb du verschachtelte Codeblöcke immer passend einrücken solltest, um die Übersicht zu behalten.

Klassen-Variablen können demnach in der gesamten Klasse verwendet werden. *Parameter* und *Variablen*, die in *Methoden* angelegt werden, existieren nur in den jeweiligen Methoden. Variablen, die in weiter verschachtelten Codeblöcken angelegt werden, existieren nur innerhalb der Codeblöcke.

Listing 5.51 zeigt eine Beispiel-Klasse, in der verschiedene Variablen angelegt werden. In den Kommentaren steht zusätzlich erneut, von wo bis wo die Variablen existieren.

Listing 5.51 Ein Beispiel für den Lebenszyklus von Variablen

```
public class ExampleClass
{
    // Klassen-Variablen sind in der gesamten Klasse gültig:
    int aClassVariable = 0;

    // Parameter existieren nur innerhalb der Methode
    void AMethod(int aParameter)
    {
```

```
        // Variablen, die innerhalb einer Methode definiert wurden,
        // existieren, wie Parameter, nur innerhalb der Methode
        int aVariableInAMethod = 5;

        if (aParameter == 4)
        {
            // Variablen, die in einem Codeblock, wie hier in der if-Abfrage,
            // definiert werden, existieren nur innerhalb des Codeblocks
            int variableInAnIf = 22;

            // Hier kann aber auch auf alle Variablen des übergeordneten Blocks
            // zugegriffen werden
            aClassVariable = aVariableInAMethod + aParameter
                        + variableInAnIf;

        } // variableInAnIf hört hier auf zu existieren

        aClassVariable += aVariableInAMethod + aParameter;

    } // aParameter und aVariableInAMethod hören hier auf zu existieren

} // aClassVariable hört hier auf zu existieren
```

5.2.11 Sichtbarkeiten

Bevor wir uns den Aufbau eines *Scripts* anschauen, möchte ich dir noch etwas zu *Sichtbarkeiten* sagen. Denn nicht jede Variable und jede Methode, die eine Klasse besitzt, kann auch von anderen Klassen aufgerufen werden. Manche sind nur für die Verwendung innerhalb der jeweiligen Klasse bestimmt. Um die Übersicht zu behalten, welche Methoden und Variablen für externe Klassen interessant sind, kannst du als Programmierer die Sichtbarkeiten von ihren Methoden und Klassen-Variablen bestimmen. Das machst du über die Schlüsselwörter `private` und `public`. Die beiden Schlüsselwörter gehören zu der Gruppe der sogenannten *Zugriffsmodifizierer*.

Es macht Sinn, Variablen und Methoden, die ausschließlich innerhalb der Klasse verwendet werden, als `private` zu markieren, damit diese nicht aus Versehen auch von anderen Klassen aufgerufen werden. Außerdem wird es anderen Entwicklern, welche die Klasse nicht kennen (oder auch dir mit ein paar Wochen Abstand), leichter fallen, die Klasse zu verwenden, wenn du weißt, dass du nur die paar öffentlichen Methoden zu verwenden brauchst. Die richtige Definition von Sichtbarkeiten erleichtert dir also das Entwickeln mit eigenen und auch fremden Klassen, weil du davon ausgehen kannst, dass alle sichtbaren Variablen und Methoden relevant sind. Methoden und Variablen ohne Zugriffsmodifizierer können nicht von einer anderen Klasse aufgerufen werden.

In dem Beispiel in Listing 5.52 soll die Enemy-Klasse nicht in der Lage sein, die Lebensenergie des Spielers direkt zu verändern, da der *Player* dann keine Kontrolle über die Schadenswerte hätte. Deswegen ist die Variable `health` als `private` deklariert. Stattdessen können andere Klassen die Methode `TakeDamage` verwenden, um dem Spieler Schaden zuzufügen. In dieser Methode wird von dem zugefügten Schaden zuvor noch der Rüstungswert abgezogen, den der Spieler besitzt. Dies wäre nicht einheitlich möglich, wenn der Gegner direkten Zugriff auf die Lebensenergie-Variable `health` hätte.

Listing 5.52 Beispiel für eine Klasse mit public- und private-Methoden

```
public class Player
{
    private int health = 100;
    private int armor = 15;
    public void TakeDamage(int damage)
    {
        // Der Schaden wird um den Wert der Rüstung (armor)
        // reduziert und von der Lebensenergie abgezogen
        int calculatedDamge = damage - armor;
        health -= calculatedDamge;
        // Hier könnte man noch prüfen, ob die Lebensenergie
        // auf 0 gefallen ist und der Spieler also Game Over ist.
    }
}
```

Zugriffsmodifizierer nur bei Klassen-Variablen

Bei Variablen, die innerhalb von Methoden deklariert werden, kann kein solcher Zugriffsmodifizierer angegeben werden, da diese ohnehin nicht von außerhalb der Klassen gesehen und verwendet werden können.

5.2.12 Ausnahmen – „Exceptions“

Beim Programmieren kommt es immer wieder vor, dass eine Situation eintritt, die so nicht passieren sollte. In der Regel handelt es sich dabei um Fehler in deinem Code, weil nicht alle möglichen Zustände berücksichtigt wurden. In dieser Situation treten *Exceptions* auf, die, wie Sicherungen in einem Gebäude, verhindern, dass durch den Fehler etwas Schlimmeres passiert. Basierend auf diesem Vergleich spricht man auch häufig davon, dass eine Exception wie eine Sicherung „fliegt“. Wenn eine Exception auftritt, wird sie im Unity-Editor in der *Console* als Fehler angezeigt. Dort siehst du auch, wo genau sie aufgetreten ist, damit du das Problem finden und lösen kannst.

Exceptions treten immer nur auf, während du dein Spiel gerade ausführst. Das kann passieren, wenn du zum Beispiel versuchst, auf eine nicht initialisierte Variable zuzugreifen, oder wenn du bei einem Array einen Index verwendest, der höher ist als die Länge des Arrays. Fliegt eine Exception, wird die betroffene Methode sofort verlassen. In Unity stürzt durch eine Exception nicht zwingend das gesamte Spiel ab, es kann gut sein, dass nur eine einzelne Funktion nicht funktioniert. Beim Testen deines Spiels solltest du die Console also stets im Auge haben, um Exceptions zu erkennen.

Exceptions können über einen *Try-Catch-Block* innerhalb des Codes abgefangen und weiterverarbeitet werden. In dem Problemlösungskapitel 5.6 findest du Beschreibungen zu gängigen *Exceptions* und Tipps, wie du sie lösen kannst.

5.3 Erster Test

Du hast jetzt einiges über den Aufbau und die Funktionsweisen von C# gelesen. Falls du die ganzen Kapitel noch nicht auswendig kennst, ist das vollkommen in Ordnung. Am besten lernst du die ganzen Sachen wahrscheinlich ohnehin, indem du sie anwendest. Du hast jetzt aber von den wichtigsten Eigenschaften der Sprache schon gelesen und weißt, wohin du zurückblättern musst, wenn du in einem späteren Kapitel nochmals darauf stößt.

Als Nächstes bereiten wir alles dafür vor, damit du dein erstes eigenes Code-Schnipsel schreiben kannst. Dafür schauen wir uns erst einmal die Entwicklungsumgebung *Visual Studio* an.

5.3.1 Microsoft Visual Studio 2017

Microsoft Visual Studio 2107 ist der Standard-Script-Editor von Unity unter Windows. Als Microsoft Hauptsponsor des *Mono-Projektes* wurde, hat Visual Studio den plattformunabhängigen Code-Editor *MonoDevelop* unter Windows abgelöst, welcher zuvor mit Unity ausgeliefert wurde.

Bei der Installation von Unity wird automatisch die *Visual Studio 2017 Community Edition* installiert, wenn noch keine andere unterstützte Visual Studio-Version auf deinem PC vorhanden ist. Die Community Edition ist eine kostenlose Variante von Visual Studio, welche du frei verwenden kannst, solange du sie nicht in einer Firmenumgebung verwendest.[2] Anders als man erwarten könnte, ist diese Version nicht stark eingeschränkt und enthält alle Funktionen, die du für deine Unity-Projekte benötigen wirst.

5.3.1.1 Visual Studio aus Unity starten

Wenn du Visual Studio mit Unity verwendest, startest du Visual Studio typischerweise nicht von Hand, sondern verwendest die Unity-Integration: Gehst du mittels Doppelklick auf ein *C#-Script* in deinem *Project Browser*, startet sich Visual Studio automatisch und zeigt dir das jeweilige Script an.

Damit du das ausprobieren kannst, musst du in deinem Projekt zunächst ein *Script* anlegen.

5.3.1.2 Ein neues Script anlegen

Unity starten und ein neues Projekt anlegen

Um die folgenden Beschreibungen direkt ausprobieren zu können, solltest du Unity starten und ein neues, leeres Projekt legen. (GearVR/GoogleVR: Dieses Projekt musst du *nicht* auf Android umstellen, da du die Code-Schnipsel am besten erst am PC testest.)

[2] Mehr Infos zu den verschiedenen Visual Studio-Versionen findest du hier: https://www.visualstudio.com/de/vs/compare/

Neue Scripte legst du nicht in *Visual Studio*, sondern im *Unity Editor* an. Als Erstes solltest du im Project Browser dafür einen neuen Ordner mit dem Namen „*Scripts*“ anlegen. Um darin jetzt ein neues Script anzulegen, klickst du mit der rechten Maustaste in den „Scripts“-Ordner. Es öffnet sich das Kontextmenü, indem du **Create/C# Script** auswählst. Es wird im *Project Browser* dann ein neues *C#-Script* angelegt und du kannst den Namen bestimmen. Achte hier stetig auf Tippfehler, da ein späteres Umbenennen relativ umständlich ist.

5.3.1.3 Der erste Start

Doppelklicke dein neues Script, um Visual Studio zu starten. Beim ersten Start von Visual Studio wirst du aufgefordert, dich mit deinem *Microsoft*-Account einzuloggen, um Zugriff auf alle Funktionen zu erhalten (u. a. Synchronisierung der Visual Studio-Einstellungen über mehrere PCs, private Git-Versionsverwaltung). Diese Funktionen benötigst du jedoch nicht, um Visual Studio mit Unity verwenden zu können. Aus diesem Grund steht es dir frei, ob du dich anmeldest oder auf **Jetzt nicht, vielleicht später** klickst, um die Anmeldung zu überspringen.

Nach dem Anmeldebildschirm wirst du aufgefordert, noch ein paar Einstellungen vorzunehmen. Wähle bei *Einstellungen* den Eintrag **Visual C#** aus, um das Layout von Visual Studio für C#-Entwicklung zu optimieren. Außerdem kannst du noch ein *Farbschema* auswählen, das dir gefällt (in den Bildern verwende ich „Dunkel“).

Aufgrund der Einrichtungs-Dialoge öffnet sich häufig beim ersten Start noch nicht das eigentlich gestartete Script, sondern nur ein leeres Visual Studio-Fenster, in dem das Unity-Projekt geladen wurde. Klicke in diesem Fall einfach erneut mit deinem Doppelklick auf das gewünschte Script im *Unity Editor* und es sollte sich wie in Bild 5.2 öffnen.

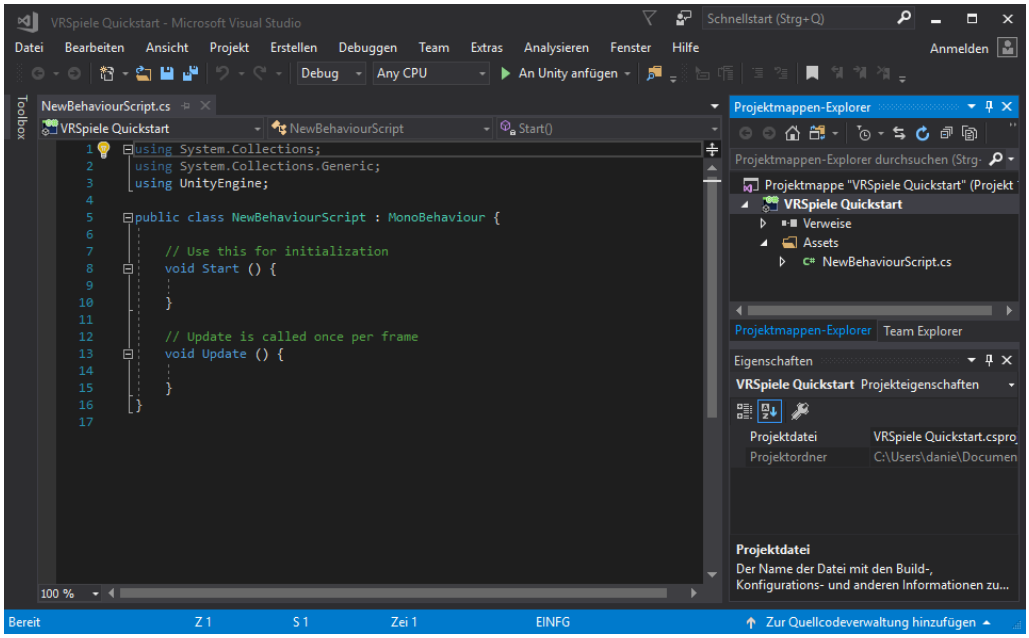

Bild 5.2 Links: der Code des derzeit offenen Scripts.
Rechts: die Projektmappe, darin sind alle C#-Scripte deines Unity-Projektes enthalten.

5.3.1.4 Häufig verwendete Funktionen

Als Nächstes stelle ich dir in einem kurzen Exkurs ein paar wichtige und praktische Funktionen von Visual Studio vor.

5.3.1.4.1 Automatische Vervollständigung

Visual Studio bietet eine Funktion, die du höchstwahrscheinlich sehr viel verwenden wirst, wenn du eigenen Code schreibst: die intelligente, automatische Vervollständigung. Wie in Bild 5.3 zu sehen, beginnt Visual Studio, sobald du anfängst zu tippen, dir Vorschläge zu machen, was du vielleicht meinen könntest. Anders als bei deinem Smartphone ist diese automatische Vervollständigung jedoch kein einfaches Wörterbuch, sondern schlägt nur Dinge vor, die du in dem jeweiligen Kontext auch verwenden kannst. Methoden und Variablen, die wegen ihrer *Sichtbarkeit* nicht verwendet werden können, werden dir also erst gar nicht vorgeschlagen. Mit der Maus und mit den Pfeiltasten deiner Tastatur kannst du die Vorschläge durchblättern. Zu dem ausgewählten Vorschlag wird zudem eine Hilfe aus der *Unity-Code-Dokumentation* angezeigt, wenn sie vorhanden ist. Drückst du die ENTER- oder RETURN-Taste auf deiner Tastatur, bestätigst du die aktuelle Auswahl und sie wird in dein *Script* übernommen. Alternativ kannst du auch einen Eintrag in der Liste mit einem Doppelklick übernehmen.

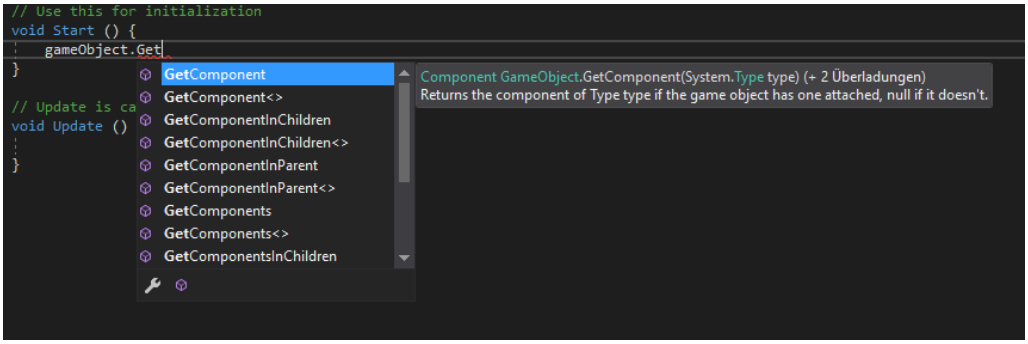

Bild 5.3 Beginnst du zu tippen, schlägt dir Visual Studio passende Klassen, Namen und Variablen vor.

Tippe bis die Auswahl klein genug ist

Während du tippst, werden die Ergebnisse automatisch weiter gefiltert. Wenn du effizient schreiben willst, hörst du also nicht sofort auf zu tippen, sobald die Liste erscheint, sondern tippst so lange, bis die Liste nur noch ein paar Einträge hat. Dann kannst du den aktuellen Eintrag bequem mit den Pfeiltasten auswählen und mit Return bestätigen.

5.3.1.4.2 Alle Verwendungen einer Variablen oder Methode finden

Je komplexer dein Projekt wird, desto komplexer werden deine Scripte und nicht immer ist es dann noch möglich, sich schnell in dem Script zurechtzufinden, selbst wenn es gut strukturiert ist. Deswegen bietet dir Visual Studio einige Funktionen, die dir helfen, sich selbst in komplexen und auch fremden Scripten schneller zurechtzufinden. Eine dieser Funktionen ist *Alle Verweise finden*.

Über diese Funktion kannst du *alle* Verwendungen einer bestimmten Variablen, Eigenschaft, Methode etc. finden. Alles, was du dafür tun musst, ist, mit der rechten Maustaste auf die jeweilige Variable, Methode etc. zu klicken und dann im Kontextmenü ALLE VERWEISE FINDEN auszuwählen.

Anschließend siehst du, wie in Bild 5.4 rot markiert, die Suchergebnisse im unteren Bereich des Fensters aufgelistet. Mit einem Doppelklick auf einen der Einträge springst du automatisch an die jeweilige Stelle, auch wenn sie in einer anderen Datei ist.

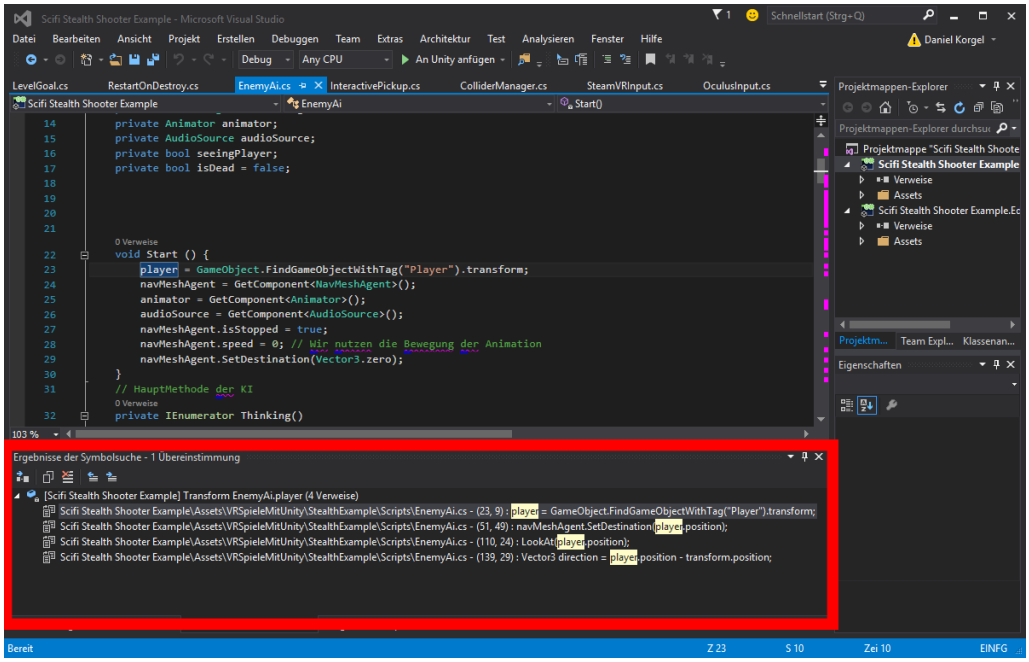

Bild 5.4 In dem Fensterbereich unten werden dir alle Stellen angezeigt, an denen das gesuchte Element gefunden wurde; in diesem Fall die Variable „Player“.

5.3.1.4.3 Variablen und Methoden umbenennen

Wenn du eine Variable, einen Parameter oder eine Methode umbenennen möchtest, kannst du das natürlich einfach im Code machen. Du scrollst an die Stelle und bearbeitest das Script. Wenn du gerade erst angefangen hast, dein Script zu schreiben, ist das auch eine vollkommen legitime Methode. Wenn dir aber mitten im Projekt auffällt, dass du einen Tippfehler in einem Variablennamen hast oder eine Methode doch lieber anders hätte lauten sollen, würde dies zu einem Problem werden. Du müsstest an allen Stellen, wo das jeweilige Element verwendet wurde, hinspringen und das jeweilige Script verändern.

Während du dein Spiel entwickelst, wird es jedoch immer wieder vorkommen, dass du einen Namen in Nachhinein ändern möchtest. Deswegen stellt dir Visual Studio dafür ein „Umbenennen“-Werkzeug zur Verfügung, mit dem du das Element und alle Stellen, wo es verwendet wird, automatisch umbenennen kannst.

Klicke dafür einfach mit der rechten Maustaste auf die jeweilige Variable, Methode etc. und wähle im Kontextmenü **Umbenennen**. Dabei ist es egal, ob du auf die Deklaration klickst oder auf eine Stelle, an der das Element verwendet wird. Wie in Bild 5.5 werden dann alle Vorkommen des jeweiligen Elementes grün markiert und du kannst den Namen an einer der markierten Stellen ändern. Deine Änderungen werden sofort auf alle Vorkommen des jeweiligen Elementes übertragen.

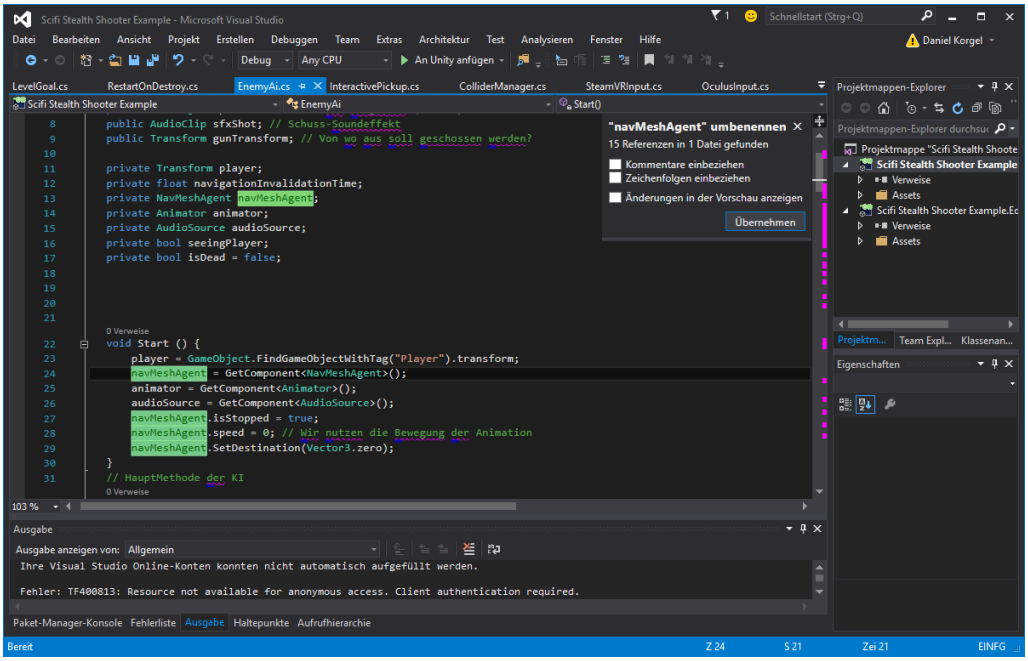

Bild 5.5 Beim Umbenennen werden alle Vorkommen des jeweiligen Elements gleichzeitig umbenannt; hier die Variable *navMeshAgent*.

5.3.2 Script, Behaviour, Component und ihr Aufbau

Script bezeichnet immer die *Quellcode-Datei*, die sich in deinem Projekt befindet. Mit dem Wort *Behaviour* bezeichnet man die *Klasse*, die in dem Script definiert wird, weil sie von der Klasse *MonoBehaviour* erbt, wie du in Listing 5.53 sehen kannst. Zusammen erben *Script* und *Klasse* ein *Component*, das du im Inspector zu einem GameObject hinzufügen kannst. In Unity müssen Script-Dateiname und Klassenname allerdings immer identisch sein und legen den Component-Namen fest. Aus diesem Grund werden die drei Wörter im Unity-Kontext häufig äquivalent verwendet, wenn es um Scripte geht.

Wenn du ein neues Script anlegst, hat das neue Script immer einen bestimmten Aufbau und ist bereits mit zwei Methoden ausgestattet. Listing 5.53 zeigt dir, wie ein vollkommen neues Script mit dem Namen *NewBehaviourScript* aussieht.

Listing 5.53 Ein leeres Script mit dem Namen „NewBehaviourScript“

```
using UnityEngine;
using System.Collections;

public class NewBehaviourScript : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Ganz oben in einem Script findest du die sogenannten using-Anweisungen. Darunter befindet sich die Definition der Klasse. In Listing 5.53 nennt sie sich *NewBehaviourScript*. Die Klasse erbt bei Unity-Scripten immer von der Unity-Klasse *MonoBehaviour*, wodurch sie zu einem *Behaviour* wird. Das erlaubt dir, das Script als ein *Component* zu GameObjects hinzuzufügen.

Innerhalb der Klassendefinition befinden sich zwei Methoden: `Start` und `Update`. Beide sind sogenannte *Event-Methoden*, welche jeweils zu bestimmten Zeitpunkten automatisch aufgerufen werden. Neben diesen beiden können noch einige andere Event-Methoden hinzugefügt werden, die jeweils zu anderen Zeitpunkten aufgerufen werden, das werden wir uns erst später noch im Detail ansehen. `Start` und `Update` sind die beiden wichtigsten Event-Methoden.

Die `Start`-Methode wird bei der Aktivierung des GameObjects ausgeführt. Der Standardfall hierfür ist das Laden der Scene, in der sich dieses Component befindet.

Die `Update`-Methode wird zum ersten Mal nach der *Start*-Methode aufgerufen. Danach wird sie regelmäßig für jeden berechneten Frame aufgerufen. Wenn dein Spiel mit 60 Bildern pro Sekunde läuft, wird auch die Update-Methode 60-mal pro Sekunde aufgerufen. In dieser Methode wirst du einen großen Teil deiner Spiellogik einfügen.

5.3.3 Debug-Ausgaben in die Console

Jetzt, wo du den grundsätzlichen Aufbau eines *Behaviours* kennst, wirst du gleich deinen ersten eigenen Code schreiben und testen können. Danach werden wir uns die fortgeschrittenen Dinge in der Programmierung mit C# und Unity ansehen. Da du dann aber bereits weißt, wie man neue Scripte anlegt und eigenen Code hinzufügt, kannst du alles, was ich dir erkläre, selber testen, wenn du willst.

Wir beginnen mit sogenannten *Debug*-Ausgaben, mit denen du Texte in die *Console* schreiben kannst. Deine Spieler können Debug-Ausgaben nicht sehen, du solltest hier also keine Nachrichten, die für den Spieler bestimmt sind, ausgeben. Debug-Ausgaben helfen dir dabei zu prüfen, ob dein Code auch wirklich das macht, was du von ihm erwartest. Debug-Ausgaben fügst du deswegen typischerweise an strategischen Stellen hinzu, sodass du nachzuvollziehen kannst, wo der Fehler liegt, wenn das Script einmal nicht das gewünschte Ergebnis liefert.

In der Unity-Einführung zu *Console* hatte ich dir bereits gesagt, es gibt drei Kategorien von Debug-Ausgaben, *Info*, *Warning* und *Error*. Für jede Art gibt es einen eigenen Befehl, wie du in Listing 5.54 sehen kannst.

Listing 5.54 Beispiele für Debug-Logs

```
Debug.Log("Das ist eine Info-Meldung");
Debug.LogWarning("Das ist eine Warnung, sie hat ein gelbes Icon.");
Debug.LogError("Das ist eine Fehlermeldung, sie hat ein rotes Icon.");
```

- **Info-Logs** solltest du für allgemeine Informationen verwenden, zum Beispiel um den aktuellen Status eines Gegners oder eines Spielers auszugeben.
- **Warnings**, solltest du in Ausnahmesituationen verwenden, die aber nicht kritisch für den Ablauf deines Spiels sind.
- **Errors** solltest du immer dann verwenden, wenn eine Situation eintritt, die auf jeden Fall vermieden werden sollte; also für spielkritische Fehler, zum Beispiel, wenn ein bestimmtes GameObject, das du für dein Script benötigst, nicht gefunden wurde.

Bild 5.6 zeigt, wie die Ausgaben aus Listing 5.54 in der *Console* aussehen würden.

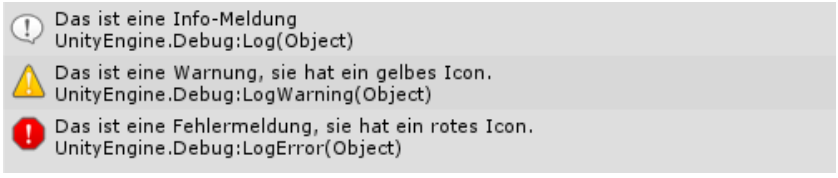

Bild 5.6 Die Ausgaben der in Listing 5.54 angegebenen Codezeilen

5.3.4 Eigenes Script testen

Wenn du nicht nur über die ganzen C#-Funktionen lesen möchtest, sondern sie auch ausprobieren willst, dann kannst du dir jetzt ein eigenes *C#-Script* erstellen, in dem du die Funktionen selber testen kannst.

Zu Erinnerung: Klicke dazu im *Project Browser* des Unity-Editors auf deinen Ordner *Scripts*. Darin kannst du nun über das Kontextmenü ein neues *C#-Script* erstellen (**Create/C# Script**). Gib diesem Script den Namen *MyFirstBehaviour*.

Mit einem Doppelklick öffnest du das Script in *Visual Studio*. Für den ersten Test kannst du zunächst einmal verschiedene Debug-Ausgaben 5.3.3 in die Start-Methode des Scripts schreiben. Die `Start`-Methode in deinem Script könnte danach so aussehen wie in Listing 5.55.

Listing 5.55 Start-Methode mit Debug-Logs

```
void Start()
{
    Debug.Log("Das ist eine Info-Meldung");
    Debug.LogWarning("Das ist eine Warnung, sie hat ein gelbes Icon.");
    Debug.LogError("Das ist eine Fehlermeldung, sie hat ein rotes Icon.");
}
```

Anschließend kannst du das Script mit der Tastenkombination **Strg + S** speichern. Alternativ kannst du auch in der Visual Studio-Toolbar auf **Datei/Alles Speichern** klicken, um alle geöffneten Dateien zu sichern.

Du kannst *Visual Studio* offen lassen und jetzt wieder in den *Unity Editor* wechseln. Das neue Script wird dann automatisch kompiliert. Du solltest bereits eine leere Scene offen haben, klicke ansonsten in der Toolbar auf **File/new Scene**. Erstelle in der Scene dann über das *Create*-Menü in der *Hierarchy* ein leeres GameObject und füge diesem ein selbst erstelltes Script hinzu.

1. Klicke dazu in der Hierarchy auf **Create/Create Empty.**
2. Das neue GameObject ist automatisch in der *Hierarchy* ausgewählt und im *Inspector* solltest du ein „GameObject“ sehen, das nur ein Transform-Component besitzt.
3. Über die **Add Component**-Schaltfläche kannst du jetzt das Script hinzufügen: **Add Component/Scripts/My First Behaviour**

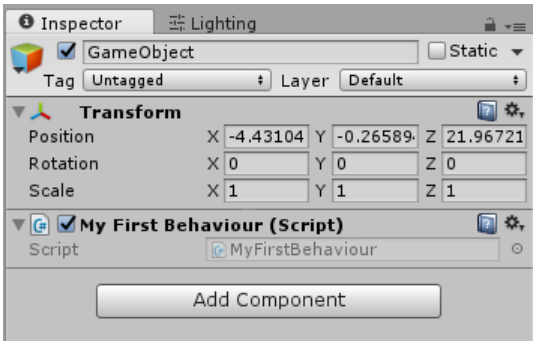

Bild 5.7 So sollte dein Behaviour an dem GameObject aussehen.

Wenn du alles richtig gemacht hast, solltest du, sobald du mit der Play-Taste des Editors das Testen startest, die drei Ausgaben wie in Bild 5.6 (Seite 34) in der *Console* sehen.

Fügst du die Zeilen aus der `Start`-Methode einmal in die `Update`-Methode ein, kannst du beobachten, wie sie immer und immer wieder ausgeführt werden. Die Zähler oben rechts in der *Console* helfen dir, das Geschehen besser einschätzen zu können.

Da die Update-Methode stetig ausgeführt wird, empfiehlt es sich, die fortgeschrittenen Grundlagen aus den kommenden Kapiteln immer in der `Start`-Methode zu testen, wo sie nur einmalig ausgeführt werden; da du ansonsten nur schwer nachvollziehen kannst, was genau passiert.

Ab hier kannst du beschriebene Funktionen selbstständig testen!

Alles, was in den folgenden Kapiteln beschrieben wird, kannst du nun selbstständig testen und damit herumspielen, wenn du möchtest. Das Ausprobieren kann beim Verstehen der einzelnen Funktionen sehr hilfreich sein.

5.3.5 Notation von Methoden und Variablen in diesem Buch

In den nachfolgenden Abschnitten werde ich dir immer wieder einzelne Methoden und Variablen vorstellen. Dabei verwende ich zugunsten der Übersicht nicht einfach die Deklarationen aus den Scripten, sondern eine Notation, die sich auf die Namen und Funktionen der Methoden und Variablen bezieht.

Hier findest du drei Beispiele für die Notation, die wir in diesem Buch verwenden werden:

- **MethodenName(***ParameterName* : `DatentypParameter`**)** : `RückgabeDatentyp`

  Hier würde die Beschreibung für die Methode mit einem Parameter stehen.

- **MethodenName(***ParameterName* : `DatentypParameter`, [*ParameterName* : `Datentyp Parameter`]**)** : `RückgabeDatentyp`

  Hier würde die Beschreibung für die Methode mit zwei Parametern stehen. Der zweite Parameter ist optional, der in eckigen Klammern steht.

- **variableName** : `Datentyp`

  Hier würde die Beschreibung der Variablen stehen.

5.4 Erweiterte Grundlagen

Du hast jetzt schon einiges über die Grundlagen von C# und auch von der Script-Erstellung erfahren. Darauf aufbauend machen wir jetzt mit den am häufigsten verwendeten Funktionen in C# weiter. Dazu gehören unter anderem *Bedingungen*, *Abfragen* und *Schleifen*. Diese Funktionen stellen die Grundlagen für so gut wie jedes Script dar, weshalb es wichtig ist, dass wir sie uns anschauen, bevor wir mit der Unity-spezifischen Programmierung loslegen.

5.4.1 Bedingungen und Verzweigungen

Nachdem du jetzt die Grundlagen kennst, können wir nun mit dem Schreiben von Scripten beginnen, die ein wenig komplexer werden. Nur mit dem Deklarieren von Variablen und einfachen Rechenanweisungen kannst du nämlich noch keine Spiellogik entwickeln.

Beim Programmieren geht es immer wieder darum, basierend auf einer bestimmten Situation eine Entscheidung zu treffen. An diesen Stellen verzweigt sich dann das Script und je nach Situation werden andere Anweisungen ausgeführt. Die Kernfunktionalität für *Verzweigungen* sind *Bedingungen*, auf deren Basis der eine oder andere Weg gewählt wird. Das Ergebnis einer *Bedingung* entspricht immer einem *Boolean*. Es kann also nur `true` (wahr) oder `false` (falsch) sein. Du definierst solche Bedingungen mit sogenannten *Vergleichsoperatoren*. In Tabelle 5.3 siehst du eine Liste aller wichtigen Operatoren.

Tabelle 5.3 Vergleichsoperatoren

Operator	Beschreibung
`a == b`	Vergleicht, ob `a` und `b` gleich sind
`a != b`	Vergleicht, ob `a` und `b` ungleich sind
`a > b`	Vergleicht, ob `a` größer ist als `b`
`a < b`	Vergleicht, ob `a` kleiner ist als `b`
`a >= b`	Vergleicht, ob `a` größer oder gleich `b` ist
`a <= b`	Vergleicht, ob `a` kleiner oder gleich `b` ist
`a && b`	Logischer UND-Operator. *Beide Operanden müssen Booleans sein*. Haben `a` *und* `b` den Wert `true`, ergibt diese Operation den Wert `true`, ansonsten `false`.
`a || b`	Logischer ODER-Operator. *Beide Operanden müssen Booleans sein*. Haben `a` *oder* `b` den Wert `true`, ergibt diese Operation den Wert `true`, ansonsten `false`. Die Bedingung ist also erfüllt, sobald *mindestens* ***einer*** der beiden Operanden den Wert `true` hat.
`!a`	Logischer NICHT-Operator. Der Operand muss ein *Boolean* sein. Das Ergebnis der Operation entspricht immer dem Gegenteil von `a`. Hat `a` den Wert `true`, ergibt `!a` den Wert `false` und umgekehrt.

Da das Ergebnis einer Vergleichsoperation immer einen Boolean ergibt, können UND-, ODER- und NICHT-Operatoren auch verwendet werden, um mehrere Bedingungen miteinander zu verknüpfen. Auf diese Weise kannst du sehr komplexe Bedingungen definieren, wenn sie benötigt sind.

Zum besseren Verständnis findest du in Tabelle 5.4 ein paar Beispiele für Bedingungen.

Tabelle 5.4 Beispiele für Bedingungen

Bedingung	Ergebnis
1 == 1	true
2 != 2	false
5 > 5	false
5 >= 5	true
!(5 >=5)	false
true && *true*	true
false && *false*	false

Bedingung	Ergebnis
true && false	false
(1 != 2) && (4 > 3)	In diesem Fall werden zuerst die beiden Operanden des UND-Operators ausgewertet. Das ergibt dann die Bedingung *false && true*, also *false UND true*. Das Endergebnis der Bedingung wäre demnach *false*.
true || *true*	true
false || *false*	false
true || false	true
(1 != 2) || (4 > 3)	Auch hier werden zuerst die Operanden ausgewertet. Das ergibt *false* || *true*, also *false ODER true*. Das Endergebnis der Bedingung wäre demnach *true*.

Es ist eine gute Angewohnheit, wenn du deine Bedingungen immer in sinnvollen Gruppen klammerst, so wie ich es in den komplexeren Beispielen gemacht habe. Das ist allerdings nicht zwingend notwendig. Ähnlich wie in der Mathematik gibt es nämlich eine feste Rangordnung, in der die Ausdrücke ausgewertet werden, wenn du sie nicht klammern solltest.

Die „normalen" Vergleichsoperatoren binden zum Beispiel stärker als die logischen Operatoren. Bei den logischen Operatoren bindet der UND-Operator stärker als der ODER-Operator. Der NICHT-Operator bindet allerdings noch stärker als der UND-Operator. In der Microsoft C#-Dokumentation findest du eine nach Bindung sortierte Liste aller Operatoren. Die entsprechende Seite habe ich dir in dem Kasten verlinkt. Du musst diese Bindungs-Reihenfolge allerdings nicht kennen, wenn du die Operanden, wie von mir empfohlen, sorgfältig klammerst, da geklammerte Ausdrücke immer als Erstes aufgelöst werden.

C#: Operatorvorrang und -orientierung

https://msdn.microsoft.com/de-de/library/aa691323(v=vs.71).aspx

5.4.1.1 If-Abfragen

Bedingungen können zum Beispiel in sogenannten *If-Abfragen* verwendet werden, um bestimmte Anweisungen nur dann auszuführen, falls die jeweilige Bedingung erfüllt ist. Im Code verwendest du dazu das Schlüsselwort `if`, gefolgt von einer in Klammern geschriebenen Bedingung. Anschließend kannst du einen Codeblock einfügen, der nur ausgeführt wird, *falls die Bedingung erfüllt ist*.

Innerhalb der Bedingung kannst du sowohl Variablen als auch konkrete Werte verwenden, wie du in Listing 5.56 sehen kannst.

Listing 5.56 Eine einfache If-Abfrage

```
if(energy > 10) {
    Debug.Log("Ausreichend Energie! (mehr als 10)");
}
```

Zusätzlich kannst du mit dem `else`-Schlüsselwort auch optional noch einen Codeblock einfügen, der nur ausgeführt wird, *falls die Bedingung nicht erfüllt ist*. Listing 5.57 zeigt dir, wie eine solche If-Abfrage mit Else-Teil aussehen könnte.

Listing 5.57 Eine If-Abfrage mit Else-Teilen

```
if(energy > 10) {
    Debug.Log("Ausreichend Energie! (mehr als 10)");
} else {
    Debug.Log("Achtung, wenig Energie!");
}
```

Die beiden Schlüsselwörter `if` und `else` können auch zusammen als `else if` verwendet werden. Das ist immer dann hilfreich, wenn du im Else-Teil deiner If-Anweisungen noch weitere Überprüfungen vornehmen möchtest. Die Else-If-Bedingung wird nämlich ausschließlich dann ausgeführt, wenn die ursprüngliche If-Abfrage *nicht erfüllt* ist. Falls diese Else-If-Abfrage ebenfalls nicht erfüllt ist, wird der nächste Else-Teil ausgeführt, wenn einer vorhanden ist.

Listing 5.58 Beispiel für eine If-Abfrage mit `else if` und `else`

```
if(energy == 100){  // 1) Wird als erstes geprüft
    Debug.Log("Energie ist voll!");
}else if(energy == 0) { // 2) wird geprüft falls 1) nicht erfüllt ist
    Debug.Log("Energie ist vollständig verbraucht!");
}else{ // 3) wird ausgeführt falls 2) ebenfalls nicht erfüllt ist
    Debug.Log("Energie ist weder voll, noch vollständig verbraucht");
}
```

5.4.2 Schleifen

Schleifen ermöglichen dir, bestimmte Codezeilen beliebig oft auszuführen, bis eine von dir angegebene Bedingung nicht mehr erfüllt ist. Schleifen sind zum Beispiel hilfreich, damit du den gleichen Code nicht zehnmal hintereinander schreiben musst, wenn sich etwas zehnmal wiederholen soll. Dadurch wird der Code übersichtlicher und besser wartbar.

Außerdem kommt es beim Programmieren auch häufig vor, dass du beim Schreiben des Scripts noch gar nicht weißt, wie oft der Code ausgeführt werden soll. Das kann zum Beispiel passieren, weil das von den Eingaben des Spielers abhängt.

Beim Schreiben von Schleifen musst du aufpassen, dass sie dein Spiel nicht „einfrieren“. Das kann dann passieren, wenn du eine Schleife schreibst, deren Bedingung *immer* erfüllt ist. Dann wird diese Schleife unendlich lange ausgeführt und dein Spiel bleibt in der Schleife hängen und friert ein.

C# bietet dir unterschiedliche Arten von Schleifen, welche ich dir nun einmal vorstellen werde.

5.4.2.1 While-Schleife

Die *While-Schleife* ist wie eine *If-Abfrage* aufgebaut. Anders als bei der If-Abfrage wird der Codeblock, der zu ihr gehört, allerdings nicht nur einmal ausgeführt, sondern so lange, wie die angegebene Bedingung erfüllt ist. Bevor die While-Schleife das erste Mal ausgeführt wird, prüft sie, ob die Bedingung bereits von Anfang an erfüllt ist. Wenn dem so ist, wird die Schleife direkt übersprungen und der dazugehörige Codeblock überhaupt nicht ausgeführt.

Ein Counter, der bis 9 zählt, würde zum Beispiel wie in Listing 5.59 aussehen.

Listing 5.59 Ein Counter, der mit einer While-Schleife erstellt wurde

```
int counter = 0;
Debug.Log("Vor While-Schleife - counter ist " + counter);

while(counter < 10) {
    Debug.Log("In While-Schleife - counter ist " + counter);
    counter++;
}

Debug.Log("Nach While-Schleife - counter ist " + counter);
```

5.4.2.2 Do-While-Schleife

Do-While-Schleifen sind eine spezielle Art von *While-Schleifen*, bei denen erst am Ende die Bedingung überprüft wird. Das bedeutet, der dazugehörige Codeblock wird immer *mindestens einmal* ausgeführt, auch wenn die Bedingung von Anfang an nicht erfüllt ist.

Listing 5.60 Eine Do-While-Schleife

```
int counter = 0;
Debug.Log("Vor Do-While-Schleife - counter ist " + counter);

do {
    Debug.Log("In While-Schleife - counter ist " + counter);
    counter++;
} while(counter > 10000);

Debug.Log("Nach Do-While-Schleife - counter ist " + counter);
```

Obwohl die Bedingung in Listing 5.60 von Anfang nicht erfüllt ist, wird der Codeblock einmalig ausgeführt und `counter` um 1 erhöht.

Do-While-Schleifen sollten immer dann verwendet werden, wenn der Codeblock mindestens einmal ausgeführt werden soll und man mit einer normalen While-Schleife doppelten Code erzeugen müsste.

5.4.2.3 For-Schleife

Während die *Do-While-Schleife* zugegebenermaßen ein besonderer Fall ist, den du nur selten verwenden wirst, ist die *For-Schleife* ein wichtiges, immer wieder vorkommendes Element.

For-Schleifen sollten immer dann verwendet werden, wenn man eine Variable hoch oder runter zählen möchte. Anders als bei der While-Schleife, welche jede Bedingung akzeptiert, ist die For-Schleife auf das Hoch- oder Runterzählen von Variablen ausgelegt.

Bei der For-Schleife gibst du, getrennt durch *Semikolons*, zuerst die *Initialisierungsanweisung* für den Zähler, dann die *Abbruch-Bedingung* und als letztes die *Anweisung zur Veränderung des Zählers* an.

Wenn die For-Schleife betreten wird, wird einmalig die *Initialisierungsanweisung* ausgeführt. Wenn die *Abbruch-Bedingung* nicht erfüllt ist, wird der Codeblock betreten. Nach jedem Ausführen des Codeblocks wird die *Anweisung zur Veränderung des Zählers* ausgeführt. Ist die Bedingung anschließend immer noch erfüllt, wird der Codeblock erneut betreten.

Ein Counter, der bis *9* zählt, würde zum Beispiel wie in Listing 5.61 aussehen.

Listing 5.61 Ein Counter, der mit einer For-Schleife realisiert wurde

```
for (int counter = 0; counter < 10; counter += 1)
{
    Debug.Log("In For-Schleife - counter ist " + counter);
}
```

In Situationen, wo ein Zähler hochgezählt werden muss, ist die For-Schleife also deutlich kürzer und übersichtlicher als eine normale While-Schleife, weil die Zähler-Variable innerhalb der Schleife deklariert wird. Die `counter`-Variable kannst du benennen, wie du möchtest. Außerdem kannst du sie auch mit einem anderen Start-Wert als 0 initialisieren. Anstelle der Anweisungen `counter += 1` kannst du natürlich auch Anweisungen mit negativer Schrittweite, z. B. `counter -= 3`, angeben. Du musst dann lediglich die Abbruch-Bedingung entsprechend anpassen, z. B. `counter > 0`.

5.4.2.3.1 Array mit For-Schleife durchlaufen

For-Schleifen wirst du sehr häufig vorfinden, wenn ein Array durchlaufen werden muss. Üblicherweise wird die Counter-Variable dann „`i`“ wie „Index“ genannt.

Listing 5.62 zeigt, wie du mit einer *For-Schleife* alle Elemente eines Arrays durchlaufen und als *Debug-Log* ausgeben kannst.

Listing 5.62 Eine typische For-Schleife zum Durchlaufen eines Arrays

```
string[] messages = new string[4]{
                "Hello World",
                "Zweite Nachricht",
                "Die dritte Nachricht",
                "Letzte Nachricht"
            };

for (int i = 0; i < messages.Length; i++)
{
    Debug.Log("Nachricht mit Index " + i + " enthält: " + messages[i]);
}
```

Die For-Schleife gibt alle Nachrichten aus, die in dem String-Array `messages` gespeichert sind.

5.4.2.4 Foreach-Schleife

Die *Foreach-Schleife* wurde gezielt dafür entwickelt, um *Arrays* und *Listen* zu durchlaufen und für jeden Eintrag einen bestimmten Codeblock auszuführen. Hier musst du dich nicht um einen Index oder eine Zähler-Variable kümmern, sondern sagst einfach: „Für jedes Element vom Typ X in diesem Array führe folgenden Code aus“.

In der Praxis sieht das dann aus wie in Listing 5.63. Hinter `foreach` folgt in Klammern zunächst eine Variable von dem Datentyp deines Arrays. Dahinter folgt das Schlüsselwort `in` und der jeweilige Variablenname des Arrays, der durchlaufen werden soll. Innerhalb der Schleife hat die angegebene Variable dann immer den Wert des aktuellen Array-Elementes.

Listing 5.63 Beispiel für eine Foreach-Schleife

```
string[] messages = new string[4]{
                "Hello World",
                "Zweite Nachricht",
                "Die dritte Nachricht",
                "Letzte Nachricht"
            };

foreach (string aMessage in messages)
{
    Debug.Log("Der Array enthält den Text " + aMessage);
}
```

Im Vergleich zur For-Schleife spart dir die *Foreach*-Variante ein paar Zeilen Programmcode, wenn du Arrays oder Listen durchlaufen möchtest. Außerdem ist sie auch ein wenig schneller, weil sie für diesen Einsatzzweck optimiert wurde. Anders als bei der *For-Schleife* hast du allerdings keine Zähler-Variable. Dies verhindert, dass du basierend auf dem Index bestimmte Operationen durchführen kannst (zum Beispiel nur jedes zweite Element ausgeben).

For oder Foreach?

Es gibt keine universale Regel hierfür. Alles, was du mit einer Foreach-Schleife machen kannst, kannst du grundsätzlich auch mit einer For-Schleife erledigen. Der Performance-Vorteil einer Foreach-Schleife macht sich zudem erst bei sehr langen Listen bemerkbar. Der größte Vorteil ist also die Einfachheit der Foreach-Schleife.

Deswegen mein Tipp: Verwende immer eine Foreach-Schleife, um Arrays und Listen zu durchlaufen, es sei denn, du musst für deine Schleife wissen, an welcher Stelle das aktuelle Element liegt, du also eine Zähler- oder Index-Variable brauchst.

5.4.2.5 Schleifen-Steuerung

Innerhalb einer Schleife, also in dem Codeblock, der zur Schleife gehört, hast du die Möglichkeit, die umschließende Schleife zu steuern. Du kannst zum Beispiel die Schleife frühzeitig zu verlassen, obwohl die Abbruch-Bedingung immer noch erfüllt ist.

Das frühzeitige Abbrechen einer Schleife erfolgt über das Schlüsselwort `break`. Die Schleife wird bei Erreichen des Schlüsselwortes *sofort* verlassen. Alles, was hinter dem break-Schlüsselwort steht, wird bereits nicht mehr ausgeführt.

Listing 5.64 zeigt einen Code, der einen beliebigen Zahlen-Array `numbers` nach dem Wert 8 durchsucht, indem er mit einer Schleife über den Array läuft und jeden Wert vergleicht. Bei einem Fund wird die Schleife mittels `break` abgebrochen und die Ausführung wird nach der Schleife fortgesetzt.

Listing 5.64 Break-Beispiel: die Suche nach der Zahl 8 in einem Integer-Array

```
int[] numbers = new int[6] { 0, 3, 4, 8, 6, 7 };
Debug.Log("Suche nach 8 in Array");

for (int i = 0; i < numbers.Length; i++)
{
    if (numbers[i] == 8)
    {
        Debug.Log("Eine 8 bei Index " + i + " gefunden, breche ab!");
        break;
    }
    Debug.Log("Kein Fund bei Index " + i + " – Wert:" + numbers[i]);
}
Debug.Log("Suche beendet");
```

Neben `break` gibt es auch noch das Schlüsselwort `continue`. Dieses Schlüsselwort bricht im Gegensatz zu `break` nicht die komplette *Schleife* ab, sondern nur den aktuellen *Durchlauf*. Wird ein `continue` erreicht, springt die Schleife sofort zum Ende des Codeblocks und beginnt mit dem nächsten Schleifen-Durchlauf. Listing 5.65 zeigt ein Beispiel für die Verwendung von `continue`, bei dem alle geraden Zahlen bis zu einer bestimmten Grenze gezählt werden. Ungerade Zahlen werden mittels *Modulo* erkannt und mit `continue` übersprungen.

Listing 5.65 Continue-Beispiel: Zähle gerade Zahlen bis 1024.

```
int evenNumbers = 0;
Debug.Log("Suche nach Gerade Zahlen");
for (int i = 0; i <= 1024; i++)
{
    if (i % 2 != 0)
    {
        Debug.Log("Keine gerade Zahl, überspringe: " + i);
        continue;
    }
    Debug.Log("Gerade Zahl, zähle: " + i);
    evenNumbers++;
}
Debug.Log("Es wurden " + evenNumbers + " gerade Zahlen gefunden.");
```

5.4.3 Eigenschaften

Eigenschaften sind eine Kombination aus *Variablen* und *Methoden,* mit denen du beispielsweise sicherstellen kannst, dass einer bestimmten Variablen nur gültige Werte zugewiesen werden können. Mit ihnen ist es aber unter anderem auch möglich, sogenannte *Ready Only*-Eigenschaften zu erstellen, die von anderen Klassen zwar gelesen, aber nicht verändert werden können. Eigenschaften sind C-Sharps Pendant zu Getter- und Setter-Methoden, wie du sie in anderen Programmiersprachen findest.

Eigenschaften oder im Englischen *Properties* erlauben dir, jeden Zugriff auf eine Variable zu überwachen und auf Wunsch eigenen Code auszuführen. Listing 5.66 zeigt, wie eine Eigenschaft typischerweise aufgebaut ist: Es existiert eine Variable, deren Zugriff du überwachen willst. Damit andere Klassen nicht darauf zugreifen können, wird sie als `private` deklariert. Dann erstellst du eine *Eigenschaft*, in dem Beispiel nennt sie sich *MyProperty*. Für Eigenschaften gilt die gleiche Namenskonvention wie für Variablen, außer dass ihr Name immer mit einem großen Buchstaben beginnt (*Pascal Case*).

Listing 5.66 Typischer Aufbau einer Eigenschaft

```
private int gamePoints = 0;
public int GamePoints {
    get {
        return gamePoints;
     }
     set {
         gamePoints = value;
     }
}
```

Eigenschaften enthalten in der Regel zwei Codeblöcke: `get` und `set`.

- In dem *get*-Block kannst du bestimmen, was genau passieren soll, wenn die Eigenschaft gelesen wird. Dieser Codeblock *muss* am Ende mit `return` einen Wert zurückgeben, der zum Datentyp der Eigenschafft passt (im Beispiel *int*). Bevor das passiert, kannst du noch zur Überprüfung Berechnungen ausführen. Der zurückgegebene Wert muss auch nicht der überwachten Variablen entsprechen, sollte aber zu dem passen, was der Name der Eigenschaft verspricht.
- In dem *set*-Block kannst du wiederum bestimmen, was passieren soll, wenn die Eigenschaft geschrieben wird. Innerhalb dieser Methode enthält die Variable `value` den Wert, welcher dieser Eigenschaft zugewiesen werden soll. Damit kannst du grundsätzlich machen, was du möchtest, typischerweise prüft man ggf., ob der Wert gültig, und weist ihn dann der internen Variablen zu.

Als Beispiel nehmen wir eine Variable, welche die Lebensenergie einer Spielfigur speichert. In diesem Beispiel legen wir fest, dass die Lebensenergie maximal den Wert 100 haben darf. Außerdem soll eine Methode aufgerufen werden, sobald die Lebenspunkte auf *0* fallen. Listing 5.67 zeigt, wie diese Lebenspunkte-Eigenschaft aussehen könnte.

Listing 5.67 Eine Beispiel-Eigenschaft „Health“

```
private int health; // interne Variable
// Kontrollierende Eigenschaft:
```

```
public int Health
{
    get {
        return health;
    }
    set {
        if (value > 100) {
            health = 100;
        } else {
            health = value;
        }
        if (health <= 0) {
            Die();
        }
    }
}
private void Die()
{
    Debug.Log("Enemy is Dead");
}
```

Die `Health`-Eigenschaft könntest du jetzt wie eine Variable von anderen Klassen lesen und schreiben.

5.4.4 Dynamische Listen

Wie du bereits erfahren hast, kannst du mit einem Array eine Liste mit einer festen Länge erstellen. Es kommt jedoch immer mal vor, dass du beim Schreiben deines Codes nicht weißt, wie viele Elemente du in die Liste einfügen wirst. Manchmal möchte man flexibel Elemente hinzufügen und entfernen, das ist mit Arrays jedoch nur umständlich möglich.

Deshalb bietet dir C# zwei Alternativen zu Arrays an: die *List*-Klasse und die *Dictionary*-Klasse. Beide Klassen sind sogenannte „generische Klassen“. Das bedeutet, du musst zum Zeitpunkt der Initialisierung angeben, welchen Datentypen du in ihnen speichern möchtest.

Beide Klassen befinden sich in dem Namespace `System.Collections.Generic`, den du über eine `using`-Anweisung einbinden musst.

5.4.4.1 List-Klasse

Die List-Klasse ist einem klassischen Array sehr ähnlich. Allerdings musst du bei der Initialisierung keine feste Größe angeben. Du kannst jederzeit mit der `Add`-Methode weitere Elemente zu der Liste hinzufügen und mit der `Remove`-Methode wieder entfernen. Die Länge der Liste wird dabei automatisch angepasst und kann über die Eigenschaft `Count` jederzeit ausgelesen werden. Wie bei einem Array kannst du über die eckigen Klammern und den Index hinter dem Variablennamen einzelne Elemente aus der Liste auslesen. Anders als bei einem Array kann die List-Klasse *nicht* denselben Wert mehr als einmal aufnehmen.

Die List-Klasse beinhaltet noch weitere Methoden wie `Contains`, mit denen du prüfen kannst, ob die Liste einen bestimmten Wert enthält, oder auch `Sort`, welcher die Liste auto-

matisch sortiert. Das Sortieren funktioniert jedoch nur für unterstützte Datentypen wie Integer und Floats, für alle anderen musst du die Vergleichsfunktionalität selber implementieren.

Bei der List-Klasse gibst du den Datentyp, den du in der Liste speichern möchtest, in *Spitzenklammern* „< >“ hinter dem Klassennamen an. Listing 5.68 zeigt dir ein Beispiel für die Verwendung der List-Klasse, um *Integer* darin zu speichern.

Listing 5.68 Beispielhafte Verwendung einer Liste (List-Klasse)

```
using System;
using UnityEngine;
using System.Collections.Generic;
public class ListExample : MonoBehaviour
{
    void Start()
    {
        List<int> listOfNumbers = new List<int>(); // Neue Liste anlegen
        Debug.Log("Länge der Liste ist: " + listOfNumbers.Count); // = 0
        listOfNumbers.Add(22); // Elemente an das Ende der Liste hinzufügen
        listOfNumbers.Add(5);
        listOfNumbers.Add(83);
        listOfNumbers.Add(2);
        Debug.Log("Länge der Liste ist: " + listOfNumbers.Count); // = 4
        if (listOfNumbers.Contains(5)) // Prüfen, ob der Wert "5" enthalten ist
        {
            Debug.Log("5 Ist in Liste enthalten! Entferne die Zahl.");
            listOfNumbers.Remove(5); // Entferne den Wert "5" aus der Liste
        }
        listOfNumbers.Sort(); // Sortiere Liste aufsteigend
        for (int i = 0; i < listOfNumbers.Count; i++)
        {
            // Lese einzelne Elemente mittels Index aus der Liste aus
            Debug.Log("Index: " + i + " Zahl: " + listOfNumbers[i]);
        }
        listOfNumbers.Clear(); // lösche alle Elemente in der Liste
    }
}
```

5.4.4.2 Dictionary-Klasse

Die *Dictionary*-Klasse (dt. „Wörterbuch“) ähnelt in ihrer Funktion sehr stark der *List*-Klasse. Jedoch funktioniert diese Variante mehr wie ein Wörterbuch, wie der Name es schon sagt. Zu jedem Eintrag in dem Dictionary gehört ein *Key* (dt. „Schlüssel“) und ein *Value* (dt. „Wert“). Um einen Wert aus einem Dictionary auslesen zu können, musst du den Schlüssel kennen. Es ist mit der Dictionary-Klasse nicht möglich, zu einem Wert den Schlüssel zu erhalten.

Ziehen wir den naheliegenden Vergleich zu einem echten Wörterbuch, entspricht das *Dictionary* also nicht einem bidirektionalen Wörterbuch, sondern einem, das nur von Sprache A nach Sprache B übersetzen kann; nicht umgekehrt. Bei einem „Deutsch–Englisch“-Wörterbuch wäre das deutsche Wort bei jedem Eintrag der Schlüssel und das englische Wort der Wert, den man vom Wörterbuch erhält.

Genauso wie bei einer *List* gibt man auch bei einem *Dictionary* bei der Deklarierung und Initialisierung an, welche Datentypen die Liste aufnehmen soll. Im Gegensatz zur *List*-Klasse gibt man hier in den Spitzenklammern bei der *Dictionary*-Klasse jedoch *zwei Datentypen* an: einen für den Schlüssel und einen für den Wert. Ein klassisches Wörterbuch würde einem `<string,string>`-Dictionary entsprechen, da es Text in Text übersetzt. Du kannst jedoch zwei beliebige Datentypen verwenden, sodass du zum Beispiel einer Zeichenkette auch eine Zahl zuordnen kannst.

- Die `Add`-Methode hat bei einem *Dictionary* zwei Parameter: einen für den Schlüssel und einen für den Wert. Jeder Schlüssel darf nur einmal in jedem Dictionary vorkommen.

  Beispiel: `dict.Add("key", "value");`
- Die `Remove`-Methode erwartet den Schlüssel als Parameter.

  Beispiel: `dict.Remove("key");`
- Um den Wert zu einem Schlüssel zu erhalten, schreibst du den Schlüssel in eckigen Klammern [ ] hinter den Variablennamen.

  Beispiel: `string value = dictionaryVariable["key"];`
- Wenn du den Wert zu einem Schlüssel ändern möchtest, kannst du dieselbe Schreibweise verwenden und dem Eintrag einen Wert zuweisen.

  Beispiel: `dictionaryVariable["key"] = "other value";`
- Zusätzlich hast du die Möglichkeit, über die *Eigenschaft* `Keys` eine Liste aller Schlüssel und über `Values` eine Liste aller Werte zu erhalten. Diese Werte kannst du dann in einer Schleife durchlaufen.

Listing 5.69 zeigt, wie du ein einfaches *Inventar* mit einem Dictionary realisieren könntest. Das Dictionary in dem Beispiel verwendet als *Key* eine Zeichenkette (`string`) und als *Value* eine Ganzzahl (`int`). Demnach sind die Schlüssel die einzelnen Items und der Wert hält fest, wie oft der Spieler dieses Item besitzt.

Listing 5.69 Ein einfaches Inventar mit einem Dictionary

```
using System;
using UnityEngine;
using System.Collections.Generic;
public class InventarExample : MonoBehaviour
{
    private Dictionary<string, int> inventarDict = new Dictionary<string, int>();
    void Start()
    {
        inventarDict.Add("Pfeile", 83);
        inventarDict.Add("Tränke", 5);
        inventarDict.Add("Gold", 8654);
        inventarDict.Add("Schlüssel", 2);

        WriteGoldToConsole();
        if (PayWithGoldGold(2000))
        {
            Debug.Log("2000 Gold ausgegeben!");
        }
        else
        {
            Debug.Log("Nicht ausreichend Gold vorhanden");
```

```
        }
        DropItem("Pfeile");
        WriteAllItemsToConsole();
    }

    private void WriteGoldToConsole() // Schreibe Gold-Wert in die Console
    {
        // Ist ein bestimmter Schlüssel vorhanden?
        if (inventarDict.ContainsKey("Gold")) {
            int goldValue = inventarDict["Gold"];
            Debug.Log("Der Spieler hat derzeit " + goldValue + " Gold");
        }
        else
        {
            Debug.Log("Der Spieler besitzt derzeit kein Gold.");
        }
    }
    private void WriteAllItemsToConsole() // Schreibe alle Items in die Console
    {
        // Dictionary mittels foreach durchlaufen
        Debug.Log("Alle Items:");
        foreach(string key in inventarDict.Keys)
        {
            // erhalte Wert für Schlüssel
            int itemCount = inventarDict[key];

            Debug.Log("Item: " + key + " - " + itemCount);
        }
    }
    public bool PayWithGoldGold(int amount) // Eine einfache Bezahl-Methode
    {
        bool success = false;
        // Einen Wert ändern:
        if (inventarDict.ContainsKey("Gold"))
        {
            if (inventarDict["Gold"] >= amount)
            {
                inventarDict["Gold"] -= amount; //Gold abziehen
                success = true;
            }
        }
        return success; // gibt zurück, ob genügend Gold vorhanden war
    }
    public void DropItem(string item) // Item fallen lassen
    {
        // Einen Wert entfernen:
        if (inventarDict.ContainsKey(item))
        {
            inventarDict.Remove(item);
            Debug.Log("Item " + item + "wurde abgelegt.");
        }
    }
}
```

5.5 Unity-spezifische Grundlagen

Damit haben wir auch schon die wichtigsten Grundlagen für das Programmieren in C# abgeschlossen. Jetzt folgen einige Unity-spezifische Funktionen, die du auf jeden Fall kennen solltest, weil du sie in Zukunft immer wieder verwenden wirst.

5.5.1 Zufallswerte

Wir beginnen die Unity-spezifischen Grundlagen mit etwas Einfachem, aber trotzdem Wichtigem: *Zufallswerten*. Nicht immer möchtest du, dass dein Spiel vollkommen vorhersehbar ist. Wenn du beispielsweise die Gegner für dein Spiel programmierst, möchtest du nicht, dass sie immer perfekt zielen und treffen. Ihre Entscheidungen sollen möglichst nicht vorhersehbar sein, sondern natürlich wirken. Solche nicht perfekten Entscheidungen, die deine Gegner treffen, sind in vielen Fällen einfach zufallsgesteuert. Es gibt eine gewisse Wahrscheinlichkeit, dass der Gegner perfekt trifft. Schwierigere Gegner haben beispielsweise eine höhere Treffer-Wahrscheinlichkeit und einfachere haben eine geringere. Ob der Gegner nun wirklich trifft, bestimmt für jeden Angriff sozusagen ein digitaler Würfelwurf, bei dem die Wahrscheinlichkeit mit eingerechnet wird.

In Unity ist es relativ einfach, eine solche Zufallszahl zu erhalten, denn dafür gibt es die *Random*-Klasse im *UnityEngine*-Namespace. Diese Klasse bietet dir eine Methode `Range`, welche dir eine zufällige Zahl zwischen zwei Grenzen zurückgibt.

Bei der Verwendung ist jedoch *Vorsicht* geboten. Denn je nachdem, wie du die Methode verwendest, verhält sie sich anders, da es zwei Varianten gibt:

- **Random.Range(***min* : `float`, *max* : `float`**)** : `float`

  Verwendest du für die untere und obere Grenze *Floats*, ist das Ergebnis eine zufällige *Kommazahl* mit zufälligen Nachkommastellen. Der Rückgabewert ist dann *größer gleich* der Untergrenze und *kleiner gleich* der Obergrenze.

- **Random.Range(***min* : `int`, *max* : `int`**)** : `int`

  Verwendest du für die untere und obere Grenze *Integer*, ist das Ergebnis eine zufällige *Ganzzahl*. Ein wichtiger Unterschied ist jedoch, wie die Grenzen gehandhabt werden: Der Rückgabewert ist hier *größer gleich* der Untergrenze und *kleiner* der Obergrenze.

Um also einen sechsseitigen Würfel zu werfen, muss man die Methode wie in Listing 5.70 mit der Untergrenze *1* und der Obergrenze *7* aufrufen, um eine Ganzzahl zwischen 1 und 6 zu erhalten.

Listing 5.70 Ein virtueller Würfelwurf in Unity

```
public int Throw6SidedDice(){
    return Random.Range(1,7);
}
```

Dies wurde deshalb so gelöst, damit man sich sehr einfach einen zufälligen Wert eines Arrays oder eine Liste ausgeben lassen kann, bei denen die *Length*- bzw. *Count*-Variable

immer um eins höher ist als bei der mit höchstem Index. Listing 5.71 zeigt, wie du ein zufälliges Array-Element eines beliebigen Arrays erhalten kannst:

Listing 5.71 Ein zufälliges Array-Element erhalten

```
string[] myItems = new string[4] { "Gold", "Arrows", "Bow", "Sword" };
public string GetRandomItem()
{
    int randomItemIndex = Random.Range(0, myItems.Length);
    return myItems[randomItemIndex];
}
```

Die *Float*-Variante funktioniert hingegen genauso, wie du erwartest. Das Ergebnis ist eine Kommazahl, muss also keine Ganzzahl sein und bei den Grenzen werden beide Grenzwerte mit einbezogen. Die Methode in Listing 5.72 gibt zum Beispiel eine zufällige Größe für den Spieler zurück, die zwischen 1.55 und 2.2 liegt. Der zufällige Wert kann auch mehr Nachkommastellen als die Grenzen haben.

Listing 5.72 Ein Beispiel für Random.Range mit Floats

```
public float getRandomPlayerHeight(){
    float randomHeight = Random.Range(1.55f, 2.2f);
    return randomHeight;
}
```

Wenn du die Float-Variante mit Ganzzahlen als Ober- und Untergrenze ausführen möchtest, musst du einfach das „*f*“ für „Float“ hinter die Zahl schreiben.

Beispiel: `Random.Range(1f, 55f);`

5.5.2 Editor-Eigenschaften

Deklarierst du in deinem *Behaviour* eine öffentliche (*public*) Variable, kannst du ihren Initialwert, den sie bei Spielstart zugewiesen bekommt, über den Editor anpassen. Das ist vor allem dann hilfreich, wenn du das Script zu unterschiedlichen GameObjects hinzufügst, bei denen es jeweils unterschiedlich konfiguriert sein soll. Grundsätzlich funktioniert das mit allen primitiven Datentypen, Arrays, Listen und allen *Unity Components*, auch selbst geschriebenen.

Wir schauen uns hier zwei gängige Beispiele an.

5.5.2.1 Editor-Eigenschaft mit Zahlenwert

Stelle dir ein *Behaviour* vor, das es einer Figur erlaubt, sich mit einer gewissen Geschwindigkeit zu bewegen. Wenn du nun einer anderen Figur die gleiche Fähigkeit, aber mit einer anderen Geschwindigkeit, geben möchtest, musst du kein zweites Script schreiben, sondern kannst einfach die Laufgeschwindigkeit im Inspector der zweiten Figur anpassen.

Listing 5.73 zeigt, wie du eine solche Variable anlegen würdest.

Listing 5.73 Ein Beispiel für eine Variable, die im Inspector angepasst werden kann

```
public class MoveTestBehaviour : MonoBehaviour
{
    public float walkSpeed = 3; // Kann im Inspector angepasst werden

    void Start()
    {
        Debug.Log("My Walk Speed is: " + walkSpeed);
    }
}
```

Das Script aus Listing 5.73 würde, wenn man es zu einem GameObject hinzufügt, im *Inspector* wie in Bild 5.8 aussehen.

Bild 5.8 Die Inspector-Ansicht des Scripts aus Listing 5.73

Änderst du die „Walk Speed"-Eigenschaft im *Inspector*, wird für dieses GameObject dann der neue Wert verwendet. Die „Walk Speed"-Eigenschaft an allen andern GameObjects, die auch das Component besitzen, wird dadurch nicht verändert.

5.5.2.2 Verweise auf GameObjects aus der Scene

Wenn dein Script auf ein anderes GameObject in der Scene zugreifen soll, benötigst du zunächst einen Verweis auf das jeweilige GameObject. Den Verweis kannst du ebenfalls über eine solche öffentliche Variable, die du dann im Inspector zuweist, erhalten.

Listing 5.74 zeigt dir, wie eine solche Variable aussehen würde, und Bild 5.9 zeigt, wie das Component anschließend im Inspector aussieht.

Listing 5.74 Ein Beispiel für eine öffentliche Variable vom Typ GameObject

```
public class MoveOtherTestBehaviour : MonoBehaviour
{
    public GameObject otherGameObject;
    public float walkSpeed = 3;

    void Start()
    {
        Debug.Log("Walk Speed of " + otherGameObject.name + " is: " + walkSpeed);
    }
}
```

Bild 5.9 Hier kannst du der Variablen jetzt ein beliebiges GameObject aus der Scene zuweisen.

Durch das Zuweisen der Variablen über den *Inspector* kannst du dann mit anderen GameObjects in deinem Script arbeiten und auf dieses und dessen Components zugreifen.

5.5.2.3 Attribute

Ganz allgemein erlauben dir Attribute in C#, Variablen, Methoden und Klassen mit bestimmten Eigenschaften zu versehen, die der *Compiler* dann berücksichtigt. In diesem Abschnitt beschäftigen wir uns in erster Linie mit einigen Attributen, die dir von Unity zur Verfügung gestellt werden. Generell sind diese Attribute etwas, das du wahrscheinlich erst bei späteren Projekten verwenden wirst. Ich möchte die wichtigsten der Vollständigkeit halber trotzdem kurz erklären.

In der Regel dienen Attribute nur dazu, ein falsches Verwenden der Components zu verhindern oder beim Einrichten der Components zu helfen.

Um ein Element mit einem Attribut auszustatten, schreibst du das gewünschte Attribut in eckigen Klammern vor bzw. über das betroffene Element. Listing 5.75 zeigt beispielhaft, wie das für das Attribut *HideInInspector* aussieht.

Listing 5.75 Variable mit einem Attribut

```
[HideInInspector]
public float currentSpeed = 55;
```

Tabelle 5.5 zeigt dir die wichtigsten Attribute, die dir durch Unity zur Verfügung gestellt werden, und beschreibt ihre Auswirkung.

Tabelle 5.5 Unity-Attribute und ihr Auswirkung

Attribut	Eigenschaft
RequireComponent(typeof(*Component*)) Beispiel: [RequireComponent(typeof(Light))] public class LightSwitch :MonoBehaviour {	Dieses Attribut kann nur für Behaviours verwendet werden und sorgt dafür, dass dieses Component nur zu einem GameObject hinzugefügt werden kann, wenn bereits ein Component von dem angegebenen Typ (im Beispiel *Light*) vorhanden ist. Ist keins vorhanden, wird automatisch ein entsprechendes Component hinzugefügt.
HideInInespctor Beispiel: [HideInInspector] public float currentSpeed;	Versteckt die Variable im Unity *Inspector*, obwohl sie *public* ist. Man verwendet es, wenn andere Klassen auf die Variable zugreifen müssen, der Wert aber nicht im Inspector auftauchen soll.
SerializeField Beispiel: [SerializeField] private float initialSpeed;	Zeigt eine Variable im *Inspector* an, auch wenn sie *private* ist. Man verwendet es, wenn andere Klassen **nicht** auf die Variable zugreifen dürfen, du sie aber trotzdem über den Inspector verändern möchtest.
Range(*min*, MAX **)** Beispiel: [Range(0,1)] public float progress; [Range(-5,5)] public int position;	Hierüber kannst du für eine Zahlen-Variable einen minimalen und maximalen Wert bestimmen, den man im *Inspector* dieser Variablen zuweisen kann. (Es ist möglich, der Variablen im Code andere Werte zuzuweisen.) *Nur für Datentypen, die Zahlen speichern (int, float)*

Attribut	Eigenschaft
Header(*Überschrift* **)** Beispiel: [Header("Important Values")] public int startPosition;	Zeigt im *Inspector* eine fett gedruckte Überschrift über der Variablen an. Wenn mehrere Variablen den gleichen Header haben, werden sie zusammen gruppiert.
Tooltip(*Hilfstext* **)** Beispiel: [Tooltip("The size of the player")] public float playerSize = 1.78f;	Fügt im *Inspector* einen Hilfstext zu der Variablen hinzu. Dieser Hilfstext wird angezeigt, wenn du mit der Maus auf den Variablennamen zeigst. Dies ist sehr hilfreich, um dich nach längerer Zeit noch daran zu erinnern, was man beim Ändern der Variablen beachten muss.

Wenn du alle Beispiele aus Tabelle 5.5 in ein *Script* kopierst und es dann zu einem *GameObject* hinzufügst, wird es im Unity *Inspector*- wie in Bild 5.10 aussehen. Da `currentSpeed` durch sein Attribut versteckt wird, ist es im *Inspector* nicht sichtbar.

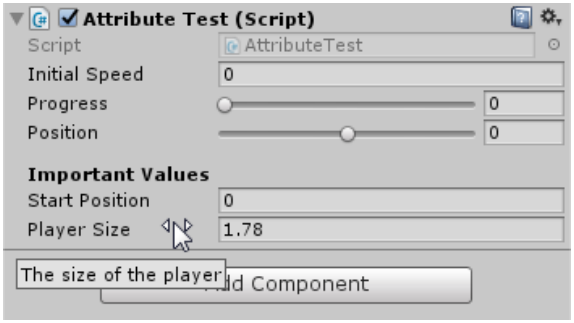

Bild 5.10 Alle Beispiele aus Tabelle 5.5 in derselben Reihenfolge

5.5.3 MonoBehaviour Lebenszyklus-Methoden

Alle *Behaviours* erben von der *MonoBehaviour*-Klasse, welche dir die Möglichkeit gibt, verschiedene Lebenszyklus-Methoden zu programmieren. Diese besonderen Event-Methoden werden automatisch und regelmäßig zu bestimmten Zeitpunkten aufgerufen, wenn sie in deinem Script vorhanden sind. Sie sind nämliche alle optional und müssen daher nicht eingefügt werden. Ein neues Behaviour, das über den Unity-Editor angelegt wurde, enthält standardmäßig erst nur die *Start*- und die *Update*-Methode.

- **Awake ()** : `void`

  Diese Methode wird aufgerufen, wenn das dazugehörige *GameObject* in der Scene aktiviert wird, auch wenn das Script selbst noch deaktiviert ist. In der Regel geschieht dies beim Laden der *Scene*.

- **Start ()** : `void`

  Diese Methode wird aufgerufen, wenn das Script aktiviert wird. Das ist in den meisten Fällen ebenfalls beim Laden der Scene der Fall. Zeitlich findet die Ausführung immer nach der *Awake*-Methode und vor dem ersten Ausführen der *Update*-Methode statt.

  Hier initialisierst du für gewöhnlich deine ganzen Klassen-Variablen.

- **Update ()** : `void`

  Die *Update*-Methode wird einmal pro Frame ausgeführt. Hier fügst du deine Spiellogik ein, welche die ganze Zeit über ausgeführt wird. Die Variable `Time.deltaTime` enthält die Zeit, die seit dem letzten Aufrufen der Update Methode vergangen ist, und kann benutzt werden, um von der Framerate unabhängige Bewegungen zu erzeugen.

- **LateUpdate ()** : `void`

  Diese Methode ist im Prinzip identisch mit der *Update*-Methode, nur dass sie immer erst direkt nach der Ausführung der Update-Methode aufgerufen wird. Diese Methode wird häufig verwendet, um Dinge zu überschreiben, die von anderen Komponenten in der Update-Methode ausgeführt werden (z. B. Animationen).

- **FixedUpdate ()** : `void`

  Diese Methode ist eine spezielle Version der *Update*-Methode, deren Ausführung nicht abhängig von der Framezahl ist, sondern in regelmäßigen Zeitabständen ausgeführt wird, die durch die Physik-Engine bestimmt werden. Diese Methode wird typischerweise für Physik-Berechnungen verwendet.

Listing 5.76 zeigt ein Beispiel-Behaviour mit allen Methoden, die zum Lebenszyklus eines Components gehören. Damit keine *NullReferenceException* fliegt, musst du zu dem gleichen GameObject auch ein *Rigidbody*-Component hinzufügen. (**Add Component/Physics/Rigidbody**). Die Funktionen in diesem Beispiel sind jedoch nur exemplarisch für das, was man in diesen Methoden typischerweise erledigt, und ergeben nicht zwingend zusammen Sinn.

Listing 5.76 Ein Behaviour mit allen Lebenszyklus-Methoden

```
using UnityEngine;
using System.Collections;

public class ExampleBehaviour : MonoBehaviour {
    private Rigidbody rb;

    void Awake()
    {
        Debug.Log("Awake-Methode wurde aufgerufen.");
    }
    // Use this for initialization
    void Start () {
        Debug.Log("Start-Methode wurde aufgerufen.");
        rb = GetComponent<Rigidbody>();
       }

    // Update is called once per frame
    void Update()
    {
        Debug.Log("Update-Methode wurde aufgerufen.");
        transform.Translate(0, 0, Time.deltaTime * 5);
    }

    void LateUpdate()
    {
        Debug.Log("LateUpdate-Methode wurde aufgerufen.");
    }
```

```
    void FixedUpdate()
    {
        Debug.Log("FixedUpdate-Methode wurde aufgerufen.");
        rb.AddForce(10.0f * Vector3.up);
    }
}
```

5.5.4 MonoBehaviour-Event-Methoden

Die Event-Methoden, die wir hier jetzt besprechen werden, sind vom Prinzip her ähnlich wie die Lebenszyklus-Methoden. Allerdings gehören sie eben nicht zum Lebenszyklus des Spiels und werden deswegen nicht regelmäßig angewandt. Diese Methoden werden nur in bestimmten Situationen ausgeführt.

Nachfolgend findest du eine Beschreibung der am häufigsten verwendeten Unity-Event-Methoden:

- **OnCollision Enter(***collisionInfo* : `Collision`**)** : `void`

  OnCollisionExit(*collisionInfo* : `Collision`**)** : `void`

  Die *OnCollisionEnter*-Methode wird aufgerufen, wenn dieses Objekt mit einem anderen zusammenstößt; *OnCollisionExit* entsprechend dann, wenn sich die GameObjects nicht mehr berühren. Mehr dazu folgt in der erweiterten Einführung in Unitys Physik-System.

- **OnTriggerEnter (***other* : `Collider`**)** : `void`

  OnTriggerExit(*other* : `Collider`**)** : `void`

  OnTriggerStay(*other* : `Collider`**)** : `void`

  OnTriggerEnter wird aufgerufen, wenn der *Collider* dieses GameObjects einen anderen *Collider* berührt und mindestens einer der beiden Collider als *Trigger* markiert ist. Zusätzlich muss noch mindestens ein GameObjects ein *Rigidbody*-Component besitzen. *OnTriggerExit* wird entsprechend ausgeführt, wenn die Collider sich nicht mehr berühren.

  OnTriggerStay wird in jedem Update ausgeführt, solange sich die Collider berühren.

  Mehr dazu folgt in der erweiterten Einführung in Unitys Physik-System.

- **OnDestroy ()** : `void`

  Wird aufgerufen, wenn dieses Component oder das zugehörige GameObject aus der Scene gelöscht wird.

- **OnApplicationQuit ()**: `void`

  Wird aufgerufen, wenn der Spieler dein Spiel beendet. Wenn diese Methode aufgerufen wird, kannst du das Beenden nicht mehr verhindern. Sie ist also *nicht geeignet*, um eine Abfrage wie „Bist du dir sicher, dass du das Spiel beenden möchtest?“ anzuzeigen.

Eine vollständige Liste aller Event-Methoden mit Beschreibungen und Beispielen findest du unter der Kategorie *Messages* in der Dokumentation zur *MonoBehaviour*-Klasse. Einen passenden Link findest du in dem Kasten.

MonoBehaviour-Dokumentation

https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

5.5.5 Auf GameObjects zugreifen

Da GameObjects ein elementarer Bestandteil in der Entwicklung mit Unity sind, wirst du regelmäßig auch von deinem Code aus auf sie zugreifen wollen. Innerhalb eines Components kannst du praktischerweise direkt auf das GameObject zugreifen, an dem sich das Component derzeit befindet. Das machst du über die Variable `gameObject`.

Listing 5.77 Ein Beispiel für den Zugriff auf das eigene GameObject

```
using UnityEngine;
using System.Collections;

public class SayMyName : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {
        string myName = gameObject.name;
        Debug.Log("Ich gehöre zu dem GameObject mit dem Namen: " + myName);
    }
}
```

Neben der `name`-Variable, mit der du den Namen eines GameObjects auslesen und ändern kannst, gibt es noch ein paar andere interessante Variablen in der GameObject-Klasse:

- **Name** : `string`

  Enthält den Namen des GameObjects und erlaubt es, diesen zu ändern

- **activeSelf** : `bool`

  Ist dieses GameObject aktiv oder inaktiv?

- **activeInHierarchy** : `bool`

  Sind das GameObject *und alle seine Eltern-Objekte aktiv*? (Wird ein Eltern-Objekt deaktiviert, werden auch alle Kinder ebenfalls in der Hierarchy deaktiviert, obwohl sie selbst noch aktiv wären und *aktivSelf* ein positives Ergebnis zurückgeben würde.)

- **transform** : `Transform`

  Enthält einen Verweis auf das *Tansform*-Component, über das jedes GameObject verfügt (mehr zu Transforms im nachfolgenden Kapitel 5.5.8)

- Die GameObject-Klasse enthält auch ein paar Methoden, die du häufiger verwenden wirst:

- **SetActive(***value* : bool**)** : `void`

  activeSelf ist ein *Read Only*-Wert. Das bedeutet, dass man diese Variable nur lesen, aber nicht schreiben kann. Deshalb existiert diese Methode, um ein GameObject aktiv oder inaktiv zu schalten.

- **GetComponent<T>()** : T

  GetComponentInChildren<T>() : T

  GetComponentInParent<T>() : T

  Diese Methoden durchsuchen das GameObject, dessen Kinder oder dessen Eltern nach *einem* Component des angegebenen Datentyps T. Es wird stets das erstbeste Component zurückgegeben. Von diesen Methoden gibt es auch jeweils eine Variante, die dir alle Components zurückgibt, die zu dem angegebenen Typ passen. Mehr dazu findest du in Kapitel 5.5.9.

- **AddComponent<T>()** : T

  Diese Methode erlaubt es dir, zur Laufzeit weitere Components zu dem jeweiligen GameObject hinzuzufügen. Hierzu wirst du in Kapitel 5.5.10 mehr Infos erhalten.

5.5.5.1 Auf andere GameObjects zugreifen

Wenn du auf andere GameObjects zugreifen möchtest, benötigst du eine Variable, welche einen *Verweis* auf das GameObject enthält. Über diese Variable kannst du dann auf das fremde GameObject zugreifen, so wie du über die `gameObject`-Variable auf das GameObject deines Components zugreifen kannst. Um so einen Verweis zu erhalten, hast du mehrere Möglichkeiten, die ich dir nun vorstellen werde.

Öffentliche Variable und ein GameObject über den Inspector zuweisen

Die erste Variante kennst du bereits: Du erstellst eine öffentliche (*public*) Variable vom Typ GameObject. Dieser Variablen kannst du über den *Inspector* ein beliebiges GameObject zuweisen. Das machst du, indem du das gewünschte GameObject einfach von der *Hierarchy* auf das Feld im *Inspector* ziehst. Diese Variante ist die performanteste Art und Weise, ein anderes GameObject zu finden und zu verwenden.

Listing 5.78 Über eine öffentliche Variable auf ein anderes GameObject zugreifen

```
using UnityEngine;
using System.Collections;
public class UseOtherGameObject : MonoBehaviour {
    public GameObject otherGameObject;
    void Start () {
        string otherName = otherGameObject.name;
        Debug.Log("Das zugewiesene GameObject hat den Namen: "+ otherName);
    }
}
```

Ein anderes GameObject über seinen Namen finden

Alternativ hast du die Möglichkeit, ein GameObject auch über seinen Namen zu suchen. Das funktioniert über die statische *Find*-Methode der Klasse GameObject:

- **GameObject.Find(***name* : `string`**)** : `GameObject`
- Ein GameObject kann allerdings nur gefunden werden, wenn es zum Zeitpunkt der Suche in der *Hierarchy* aktiv ist. Listing 5.79 zeigt, wie du die Methode verwenden kannst, um auf ein anderes GameObject zugreifen zu können.

Listing 5.79 Ein GameObject über seinen Namen finden

```
using UnityEngine;
using System.Collections;
public class FindOtherGameObject : MonoBehaviour {
    private GameObject otherGameObject;
    void Start () {
        otherGameObject = GameObject.Find("GameObjectB");
        string otherName = otherGameObject.name;
        Debug.Log("Das zugewiesene GameObject hat den Namen: "+ otherName);
    }
}
```

Diese Methode kostet viel Performance und sollte deswegen immer nur in der *Start*- oder *Awake*-Methode und nicht in einer *Update*-Methoden verwendet werden. Wenn du den Verweis in jedem Update benötigst, solltest du ihn, wie in Listing 5.79 zu sehen, in einer Klassen-Variablen zwischenspeichern.

GameObejct über Tag finden

Anstelle ein GameObject über seinen Namen zu finden, hast du auch die Möglichkeit, ein oder mehrere GameObjects über das jeweilige *Tag* zu finden. (Zur Erinnerung: Tags kannst du im GameObject-Kopf im *Inspector* zuweisen). Finden kannst du das getaggte GameObject dann über die statische Methode *FindWithTag*.

- **GameObject.FindWithTag(***tag* : `string`**)** : `GameObject`

  Dies hat gegenüber der Suche über den Namen den Vorteil, dass Tags leichter handzuhaben sind. Im *Inspector* kannst du sie nämlich, nach dem Anlegen, aus einer Drop-down-Liste auswählen. Ein weiterer Vorteil ist, dass du auch mehrere GameObjects mit demselben Tag versehen kannst. Möchtest du dann alle GameObjects mit einem bestimmten Tag finden (anstelle des Erstbesten), kannst du folgende Methode verwenden:

- **GameObject.FindGameObjectsWithTag(***tag* : `string`**)** : `GameObject[]`

  Diese Methode gibt dir einen *Array* mit allen gefundenen GameObjects zurück. Beide Methoden finden allerdings ebenfalls nur GameObjects, die zum Zeitpunkt der Suche *aktiv* in der Hierarchy sind. Listing 5.80 zeigt beide Methoden in einem praktischen Beispiel.

Listing 5.80 Ein oder mehr GameObjects über ein Tag finden

```
using UnityEngine;
using System.Collections;
public class FindOtherGameObject : MonoBehaviour {
    private GameObject mainCamera;
    private GameObject[] allPlayers;
    void Start () {
        mainCamera = GameObject.FindWithTag("MainCamera");
        allPlayers = GameObject.FindGameObjectsWithTag("Player");
        Debug.Log("Die Main Camera hat den Namen: "+ mainCamera.name);
        Debug.Log("Es wurden " + allPlayers.Length + " Spieler gefunden");

    }
}
```

5.5.6 GameObjects via Code erstellen

Du musst nicht zwingend alle GameObjects, die dein Spiel benötigt, vorab im Unity-Editor anlegen, sondern kannst sie auch erzeugen, während dein Spiel läuft. Wenn du beispielweise ein Spiel mit einer sehr großen Anzahl an Gegnern programmierst, die den Spieler nach und nach in Wellen angreifen, würdest du sie idealerweise nicht irgendwo in der Scene deaktiviert platzieren. Besser wäre, sie zur Laufzeit zu erzeugen, sobald sie benötigt werden.

Beim Anlegen von GameObjects hast du zwei Möglichkeiten: Entweder du legst ein vollkommen leeres GameObject an und konfigurierst es dann über deinen Code oder du erzeugst ein vollständig vorkonfiguriertes GameObject aus einem *Prefab*. Je nach Situation sind beide Vorgehensweisen wichtig, weshalb wir uns beide einmal ansehen werden.

5.5.6.1 Leere GameObjects

Leere GameObjects anlegen ist sehr intuitiv: Du legst einfach innerhalb deines Codes ein *GameObject* über das Schlüsselwort `new` an. Das GameObject wird dann zu der aktuellen *Scene* und *Hierarchy* hinzugefügt. Dieses GameObject ist ab dem Moment der Erzeugung vollkommen eigenständig und nicht von deinem Component oder der Variable, in der du den Verweis speicherst, abhängig.

Der Konstruktor hat einen String-Parameter, wo du den Namen des GameObjects angeben kannst. Listing 5.81 zeigt, wie das Erstellen eines neuen GameObjects im Code aussieht. Auch wenn die Variable `aNewGameObject` nur innerhalb der `createGameObject`-Methode existiert, bleibt das GameObject weiterhin in der Scene bestehen.

Listing 5.81 Ein leeres GameObject über Code anlegen

```
public void createGameObject(){
    GameObject aNewGameObject = new GameObject("NeuesGameObject");
}
```

5.5.6.2 Aus Prefab erstellen

Wenn du, während dein Spiel läuft, komplexere GameObjects, zum Beispiel Gegner mit vielen Components, erzeugen möchtest, ist es in der Regel sinnvoll, vorkonfigurierte *Prefabs* dafür zu verwenden.

Um ein GameObject aus einem *Prefab* zu erzeugen, benötigst du als Erstes einen Verweis auf das jeweilige *Prefab*, das du erzeugen willst. Dies erledigst du im einfachsten Fall über eine öffentliche Variable vom Typ *GameObject*. Über den *Inspector* kannst du diesem Feld dann ein *Prefab* aus deinem *Project Browser* zuweisen.[3]

Das Erzeugen erfolgt dann über die `Instantiate`-Methode:

- **Instantiate(** *prefab* : `GameObject`, *globalPosition* : `Vector3`, *globalRotation* : `Quaternion`**)** : `GameObject`

[3] Es ist auch möglich, hier einen Verweis auf ein GameObject aus der Scene anzugeben. In diesem Fall würde das jeweilige GameObject dann anstelle eines Prefabs als Vorlage verwendet werden.

Listing 5.82 zeigt dir, wie ein einfaches Script aussehen könnte, das in der `Start`-Methode ein neues GameObject aus einem *Prefab* anlegt.

Listing 5.82 Ein GameObject aus einem Prefab erzeugen

```
using UnityEngine;
using System.Collections;
public class CreatePrefab : MonoBehaviour
{
    public GameObject aPrefab;
    void Start()
    {
        Instantiate(aPrefab, new Vector3(0, 0, 0), Quaternion.Euler(0, 0, 0));
    }
}
```

5.5.7 GameObjects zerstören

Wenn du GameObjects zur Laufzeit anlegst, wirst du sie auch zur Laufzeit zerstören wollen. Das machst du über die statische Methode `Destroy`.

- **Destroy (***object* : `Object`**)** : `void`

  Das GameObject wird allerdings nicht unmittelbar gelöscht, sondern erst sobald der aktuelle Frame vollständig berechnet wurde. Zerstörst du ein GameObject in der Update-Methode, ist das Objekt erst beim nächsten Ausführen der Update-Methode tatsächlich gelöscht.

Listing 5.83 Methode, um das eigene GameObject zur Laufzeit zu zerstören

```
public void destroyMe(){
    Destroy(gameObject);
}
```

5.5.8 Transforms: GameObjects bewegen

Natürlich möchtest du GameObjects auch über deinen Code bewegen können, zum Beispiel um den Spieler laufen zu lassen. Das machst du über das *Transform*-Component, welches an jedem GameObject vorhanden ist. Auf das Transform-Component kannst du innerhalb eines Behaviours über die Variable `transform` zugreifen. Wenn du auf das Transform eines anderen GameObjects zugreifen möchtest, kannst du das über eine gleichnamige Variable innerhalb der GameObject-Klasse tun. Zum Beispiel: `otherGameObject.transform`

Als Erstes möchte ich dir einmal die wichtigsten Variablen des Transform-Components kurz vorstellen:

- **position** : `Vector3`

  Die aktuelle Position im Weltkoordinatensystem.

- **localPosition** : `Vector3`

  Die aktuelle Position im lokalen Koordinatensystem, also die Position relativ zum Eltern-GameObject. Dieser Wert wird dir im *Inspector* als *Position* angezeigt.

- **rotation** : `Quaternion`

  Die aktuelle Rotation im Weltkoordinatensystem

- **localRotation** : `Quaternion`

  Die aktuelle Rotation im lokalen Koordinatensystem

- **eulerAngles** : `Vector3`

  Die aktuelle Rotation im Weltkoordinatensystem. Die einzelnen Werte des Vektors geben jeweils die Rotation auf der x-, y- und z-Achse in Grad an. In vielen Fällen fällt es leichter, mit diesen eulerschen Winkeln zu arbeiten als mit den *Quaternions*, die Unity intern verwendet.

- **localEulerAngles** : `Vector3`

  Die aktuelle Rotation im lokalen Koordinatensystem. Dieser Wert wird dir im *Inspector* als *Rotation* angezeigt.

- **localScale** : `Vector3`

  Die aktuelle Skalierung im lokalen Koordinatensystem. Dieser Wert wird dir im *Inspector* als *Scale* angezeigt.

- **parent** : `Transform`

  Diese Variable gibt dir das Transform-Component des Eltern-Objekts deines Transforms zurück. Ist kein Eltern-Objekt vorhanden, ist der Rückgabewert *null*. Dieser Wert ist *ReadOnly*. Um das Parent eines GameObjects über den Code zu ändern, musst du die Methode *SetParent*-verwenden.

- **gameObject** : `GameObject`

  Diese Variable zeigt wiederum auf das GameObject, das zu diesem Transform gehört. Es sind also Kreisbezüge wie `gameObject.transform.gameObject.name` möglich, welche du aber vermeiden solltest. Das Beispiel wäre identisch mit `gameObject.name`.

- **forward** : `Vector3`

  Diese Variable enthält einen Richtungsvektor, der aus der Sicht des GameObjects (also basierend auf seiner aktuellen Rotation) nach vorne zeigt. Dieser Richtungsvektor entspricht dem blauen Pfeil in der *Scene View*, wenn das Koordinatensystem auf *local* steht.

- **up** : `Vector3`

  Diese Variable enthält einen Richtungsvektor, der aus der Sicht des GameObjects nach oben zeigt.

Listing 5.84 zeigt dir beispielhaft, was du mit diesen Variablen bereits machen kannst.

Listing 5.84 Ein Beispiel, welches das GameObject kontinuierlich nach vorne bewegt und dabei dreht

```
using UnityEngine;
using System.Collections;
public class MoveMeInCircles : MonoBehaviour
{
```

```
    public float forwardSpeed = 5;
    public float rotationSpeed = 3;
    void Update()
    {
        // Vorwärts bewegen
        Vector3 posChange = transform.forward * forwardSpeed;
        transform.position = posChange * Time.deltaTime;
        // Aktuelle Drehung auslesen, ändern und wieder zuweisen:
        Vector3 currentEulerAngles = transform.localEulerAngles;
        // Der Y-Wert beschreibt den Winkel auf der Y-Achse
        currentEulerAngles.y += rotationSpeed * Time.deltaTime;
        transform.eulerAngles = currentEulerAngles; //Neue Rotation zuweisen
    }
}
```

5.5.8.1 Bewegungen unabhängig von der FPS

Dir ist vielleicht aufgefallen, dass in Listing 5.84 Änderungen an der Rotation und Position des GameObjects immer mit der Variablen `Time.deltaTime` multipliziert werden. Diese Variable enthält immer die Zeit, die seit dem letzten Aufruf dieser Update-Methode verstrichen ist.

Wenn du Änderungen an der Rotation und Position mit dieser Variablen multiplizierst, ist die Bewegungsgeschwindigkeit unabhängig von der Framezahl, sondern bezieht sich auf *Pro Sekunde*.

Drehst du ein GameObject in der Update-Methode jedes Mal um 3 Grad, wird das GameObject für jeden berechneten Frame ebenfalls um 3 Grad gedreht. Das bedeutet, das GameObject würde sich doppelt so schnell drehen, wenn dein Spiel mit 60 FPS statt mit 30 läuft. Drehst du das GameObject um `3 * Time.deltaTime`-Grad, dreht es sich unabhängig von der FPS-Zahl mit *3 Grad pro Sekunde*. Grundsätzlich solltest du immer, wenn du die Rotation oder Position eines GameObjects flüssig verändern möchtest, diese Variable verwenden.

5.5.9 Auf andere Components zugreifen

Auf andere GameObjects zugreifen reicht in der Regel zum Entwickeln eines Spiels noch nicht aus. Du musst dafür häufig auch auf andere Components zugreifen können.

Um Eigenschaften von einem anderen Component zu ändern, benötigst du zunächst einen Verweis auf das jeweilige Component. Diesen Verweis erhältst du, indem du die Methode `GetComponent<`*ComponentName*`>()` auf dem jeweiligen GameObject ausführst. Dabei musst du in die Spitzenklammern den Datentyp des gewünschten Components einfügen, welcher in der Regel dem Namen des Components *ohne Leerzeichen* entspricht (zum Beispiel `GetComponent<BoxCollider>()`).

Diesen Verweis kannst du anschließend verwenden, um auf alle sichtbaren Variablen des Components zuzugreifen. In der Regel sind das alle Eigenschaften, die du auch im Inspector anpassen kannst. Teilweise gibt es aber auch noch weitere Eigenschaften und Methoden, die im Inspector nicht sichtbar sind.

Findet `GetComponent` kein passendes Component, ist der Rückgabewert *null*.

Listing 5.85 zeigt dir, wie du GetComponent verwenden kannst, um die Farbe eines *Light*-Components anzupassen.

Listing 5.85 GetComponent-Beispiel

```
void Start()
{
    Light myLight = GetComponent<Light>();
    myLight.enabled = true; // Component aktivieren (entspricht Checkbox)
    myLight.color = Color.red;
    myLight.range = 15;
}
```

Damit der Aufruf aus Listing 5.85 funktioniert, muss sich am gleichen GameObject, wie das Script, ein *Light*-Component befinden. Sobald du dein Spiel dann startest, wird die Farbe des Lichtes auf Rot und die Reichweite auf 15 geändert. Damit die Änderung in der Scene sichtbar ist, muss der *Mode* des Lichtes auf *Realtime* stehen. Bild 5.11 zeigt, wie das Light-Component nach der Anpassung durch das Script zur Laufzeit aussieht.

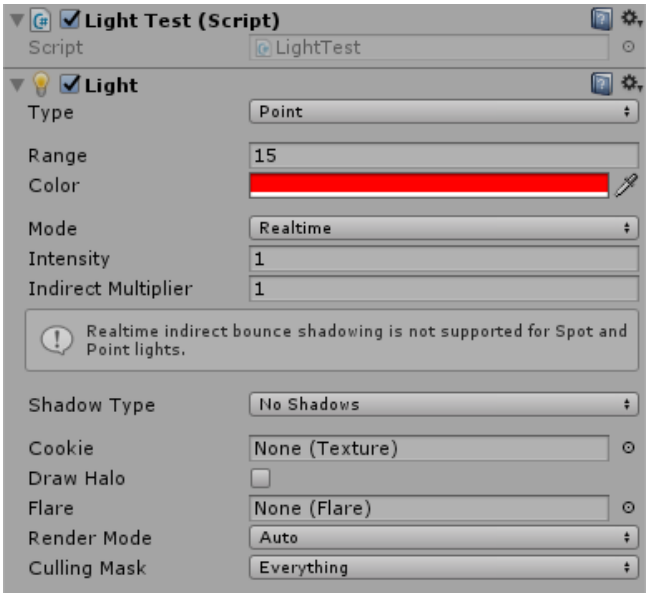

Bild 5.11 Die Start-Methode des Light-Test-Scripts ändert die Color- und die Range-Eigenschaft des Light-Components.

Welche Variablen gibt es in welchen Components?

In der Scripting-API von Unity findest du für alle Unity-internen Components eine detaillierte Auflistung aller Variablen und Methoden, welche dir zur Verfügung stehen. Verwende dazu am besten die Suchen oben rechts.

https://docs.unity3d.com/ScriptReference/

5.5.9.1 Alle Components eines Typs erhalten

Befinden sich an einem GameObject mehrere Components desselben Typs, gibt dir die `GetComponent`-Methode immer nur das erstbeste Component mit dem angegebenen Typ zurück. Wenn du Verweise zu allen Components eines bestimmten Typs erhalten möchtest, musst du auf dem jeweiligen GameObject die Methode `GetComponents <`*ComponentName*`>` aufrufen.

Diese Methode gibt dir einen Array mit allen passenden Components zurück, die auf dem jeweiligen GameObject gefunden wurden. Wenn keine passenden Components gefunden werden, ist der Rückgabewert ein Array mit der Länge *0*.

Listing 5.86 zeigt, wie du zum Beispiel alle Light-Components, die sich an einem GameObject befinden, ausschalten kannst.

Listing 5.86 Beispiel mit GetComponents

```
void Start()
{
    Light[] lightsOnThisObject = GetComponents<Light>();
    foreach (Light light in lightsOnThisObject)
    {
        // setze auf Grün und schalte Licht aus
        light.color = Color.green;
        light.enabled = false;
    }
}
```

Damit der Code aus Listing 5.86 einen Effekt hat, müssen sich ein oder mehr *Light*-Components an demselben GameObject befinden wie das Script.

5.5.9.2 Components an anderen GameObjects

Die GetComponent-Methoden suchen immer nur an dem GameObject, auf dem sie ausgeführt werden. Wenn du auf einem fremden GameObject suchen möchtest, benötigst du zunächst einen Verweis auf das entsprechende GameObject (siehe Kapitel 5.5.5.1). Anschließend rufst du, wie in Listing 5.87 zu sehen, die `GetComponent`-Methode des jeweiligen GameObject-Objekts auf.

Listing 5.87 Beispiel für GetComponent-Aufrufe an einem anderen GameObject. In diesem Beispiel wird das Camera-Component an dem „Main Camera"-GameObject deaktiviert.

```
void Start () {
      GameObject mainCamera = GameObject.FindWithTag("MainCamera");
      Camera cameraComponent = mainCamera.GetComponent<Camera>();
      cameraComponent.enabled = false;
}
```

5.5.9.3 Components in Kindern/Eltern suchen

Neben den Methoden `GetComponent` und `GetComponents`, welche nur das GameObject durchsuchen, auf dem sie ausgeführt werden, gibt es auch die Möglichkeit, alle Kind- beziehungsweise Eltern-GameObjects zu durchsuchen. Dafür verwendest du folgende Methoden:

- *GetComponentInChildren* <Typ>() : Typ

  GetComponentsInChildren <Typ>() : Typ[]

  Diese Variante durchsucht das jeweilige GameObject und alle seine Kinder, inklusive Kinder der Kinder etc.

- *GetComponentInParent* <Typ>(): Typ

  GetComponentsInParent <Typ>() : Typ[]

  Diese Variante durchsucht das jeweilige Parent-GameObject nach dem Component und ist identisch mit `transform.parent.GetComponent<Typ>()`.

5.5.9.4 Components überall in der Scene finden

Es gibt auch Möglichkeiten, alle Components eines bestimmen Typs aus der gesamten Scene zu finden. Diese Methoden sind jedoch deutlich weniger performant als die GetComponent-Alternative, weshalb du noch besser darüber nachdenken solltest, ob du die Variablen wirklich auf diese Weise finden musst. Wenn du sie verwendest, solltest du sie nicht in jedem Frame (z. B. in der Update-Methode) ausführen, sondern nur einmalig, und das Ergebnis zwischenspeichern.

Die beiden Methoden `FindObjectOfType <Typ>` und `FindObjectsOfType <Typ>` sind von der Handhabung identisch zu *GetComponent* und *GetComponents*, durchsuchen aber die *gesamte* Scene. Listing 5.88 zeigt dir, wie du mit den beiden Components ein einzelnes bzw. alle Lichter in der Scene finden kannst.

Listing 5.88 Beispiel FindObjectOfType und FindObjectsOfType mit Zwischenspeichern

```
using UnityEngine;
using System.Collections;
public class FindOtherGameObject : MonoBehaviour
{
    private Light someLight;
    private Light[] allLights;
    void Start()
    {
        //finde irgendein(!) Licht in der Scene
        someLight = FindObjectOfType<Light>();
        //finde alle(!) Lights in der Scene
        allLights = FindObjectsOfType<Light>();
    }
}
```

5.5.10 Components zu GameObject hinzufügen

Du kannst mit einem Script auch jederzeit ein Component zu einem GameObject hinzufügen. Das Hinzufügen von Components geschieht über die Methode `AddComponent <Typ>()`. Anders als bei *GetComponent* musst du *AddComponent* immer auf dem Verweis zu einem GameObject ausführen. Listing 5.89 zeigt, wie du beispielsweise zu dem GameObject, an dem sich das Script befindet, ein Component hinzufügen kannst.

Listing 5.89 Beispiel für AddComponent

```
public void AddANewLightToThisGameObject()
{
    Light newLight = gameObject.AddComponent<Light>();
}
```

5.5.11 Components entfernen

Wenn du zur Laufzeit Components hinzufügen kannst, wirst du wahrscheinlich früher oder später auch Components von einem GameObject entfernen wollen. Das funktioniert, wie schon bei den GameObjects, mit der `Destroy`-Methode. Anstelle der Methode den Verweis zu einem GameObject als Parameter zu übergeben, übergibst du ihr den Verweis zu einem Component, das zerstört werden soll. Listing 5.90 zeigt als Beispiel, wie du alle Lichter in der Scene zerstören kannst.

Listing 5.90 Beispiel für das Entfernen von Components

```
public void RemoveAllLights()
{
    Light[] someLights = FindObjectsOfType<Light>();
    for (int i = 0; i < someLights.Length; i++)
    {
        Light aLight = someLights[i];
        Destroy(aLight);
    }
}
```

5.5.12 Parallele Methoden mit Coroutines

Mit Coroutines kannst du Methoden ausführen, deren Logik nicht wie bei der Update-Methode innerhalb der Berechnung eines einzigen Frames ausgeführt wird, sondern unabhängig von der Frame-Berechnung.

Für gewöhnlich verwendest du Coroutines zum Beispiel, ...

- ... wenn du innerhalb deines Codeblocks auf etwas warten möchtest (z. B. Downloads, Timer, Position des Spielers)
- ... wenn du einen bestimmten Codeblock nur für einen gewissen Zeitraum ausführen möchtest (z. B. geScriptete Abläufe, Bewegungen oder Animationen)

Wie eine Coroutine aussieht, zeigt dir Listing 5.91.

Listing 5.91 Eine einfache Coroutine

```
IEnumerator ExampleCoroutine()
{
    Debug.Log("Coroutine gestartet " + Time.realtimeSinceStartup);
    yield return new WaitForSeconds(2); // Warte 2 Sekunden
    Debug.Log("Coroutine gestoppt. Zeit: " + Time.realtimeSinceStartup);
}
```

Wie du in Listing 5.91.sehen kannst, haben Coroutines immer den Rückgabetyp `IEnummerator`. Über `yield return` kannst du die Ausführung an einer beliebigen Stelle in deiner Coroutine unterbrechen und später fortsetzen. Jede Coroutine muss mindestens eine „yield return"-Anweisung besitzen. Was hinter „yield return" steht, bestimmt, wie lange gewartet werden soll. Hier gibt es viele verschiedene Möglichkeiten, von denen zwei jedoch am häufigsten verwendet werden:

- **yield return null;**

  Unterbricht die Ausführung bis zum nächsten Frame.

- **yield return new WaitForSeconds (***Wartezeit* : `float`**);**

  Unterbricht die Ausführung für die angegebene Anzahl von Sekunden.

Eine Coroutine kannst du nicht wie eine normale Methode aufrufen. Du musst sie *starten*. Das funktioniert über die Methode `StartCoroutine`, wie dir Listing 5.92 zeigt.

Listing 5.92 So startest du eine Coroutine.

```
void Start () {
    StartCoroutine ( ExampleCoroutine() );
}
```

Coroutines haben keinen eigenen Thread/Prozess!

Coroutines werden nicht wirklich parallel (z. B. in einem eigenen Prozess) ausgeführt, sondern *stückweise* im Hauptprozess des Spiels, sie können also bei falscher Verwendung das Spiel ebenfalls einfrieren.

5.5.13 Scripted-Event-Beispiel

Da du Coroutines bei komplexeren Spielen häufiger verwenden wirst, möchte ich dir noch ein Beispiel zeigen. Die *Coroutine* in Listing 5.93 bewegt zum Beispiel das GameObject des Scripts 60 Frames lang senkrecht oben. Dann schaltet es ein Light-Component aus, wartet fünf Sekunden und schaltet es wieder ein.

Listing 5.93 Beispiel für eine Coroutine, die dein gescriptetes Event abspielt

```
void Start() {
    StartCoroutine(TriggerLightForPlayer());
}

IEnumerator TriggerLightForPlayer()
{
    Light light = GetComponent<Light>();
    // Bewege GameObject nach oben
    for (int i = 0; i < 60; i++) // Wiederhole 60-mal
    {
        transform.position += new Vector3(0, 1, 0) * Time.deltaTime;
        yield return null; // Warte auf nächsten Frame
```

```
    }
    Debug.Log("Schalte Licht aus");
    light.enabled = false;
    yield return new WaitForSeconds(5.0f); //Wartet für 5 Sekunden
    Debug.Log("Schalte Licht wieder ein");
    light.enabled = true;
}
```

Als Erstes holt sich die Coroutine einen Verweis auf das Light-Component, welches später ein- und ausgeschaltet werden soll. Dann folgt eine *For-Schleife,* die insgesamt 60-mal ausgeführt wird. In der For-Schleife findest du die Zeile `yield return null;`. Durch diese Zeile sagst du dem Spiel, dass die Ausführung dieser Coroutine an dieser Stelle bis zum nächsten Update unterbrochen werden soll. Das bedeutet vereinfacht gesagt, dein Spiel läuft für einen Frame weiter und dann wird die Ausführung an der Stelle nach der *yield*-Anweisung fortgesetzt. In dem Beispiel würde dann die For-Schleife erneut durchlaufen werden, es sei denn, die Abbruch-Bedingung ist erfüllt. Anschließend wird das Licht ausgeschaltet und es folgt die nächste `yield return`-Anweisung, welche die Ausführung erneut unterbricht; in diesem Fall allerdings nicht bis zum nächsten Update, sondern für fünf Sekunden. Danach wird die Ausführung bis zum Ende der Methode fortgesetzt, da keine weitere *yield*-Anweisung mehr folgt.

5.5.14 Methoden verzögert aufrufen mit Invoke

Wenn du normale Methoden verzögert aufrufen möchtest, kannst du das grundsätzlich natürlich mit einer Coroutine lösen, die du mittels *WaitForSeconds* warten lässt. Es geht jedoch auch einfacher, wenn nur eine bestimmte Methode nach x Sekunden aufgerufen werden soll. Das kannst du nämlich mit einem *Invoke* realisieren, der genau für diesen Einsatzzweck entwickelt wurde.

Der `Invoke`-Methode übergibst du als Parameter einfach den Namen der Methode, die du aufrufen möchtest, als `string`. Der zweite Parameter bestimmt, wie viele Sekunden bis zum Aufruf gewartet werden soll. Listing 5.94 zeigt, wie das beispielhaft für eine Methode mit dem Namen *DelayedOutput* aussehen würde.

Listing 5.94 Beispielhafter Aufruf von Invoke

```
void Start(){
    Invoke("DelayedOutput", 3.5f);
}
void DelayedOutput(){
    Debug.Log("Diese Nachricht wurde verzögert");
}
```

Neben der eigentlichen `Invoke`-Methode existieren noch drei, mit ihr verwandte, Methoden. Diese helfen dir, verzögerte Methodenaufrufe genauer zu steuern:

- **Invoke(***methodenName* : `string`, *warteZeit* : `float`**)** : `void`

  Die Invoke-Methode zum Planen von verzögerten Methodenaufrufen.

- **InvokeRepeating(***methodenName* : `string`, *warteZeitZwischenAufrufen* : `float`**)** : `void`

  Diese Variante funktioniert wie die normale `Invoke`-Methode, nur wird die angegebene Methode immer wieder nach der angegebenen Zeit aufgerufen, bis die Methode `CancelInvoke` mit dem gleichen Methodennamen aufgerufen wird, und nicht nur einmal.

- **CancelInvoke(** [*methodenName* : `string`] **)** : `void`

  Wenn ein Aufruf für die angegebene Methode mittels `Invoke` oder `InvokeRepeating` geplant wurde, wird der geplante Aufruf abgebrochen. Wird kein Methodenname übergeben, werden alle geplanten Aufrufe abgebrochen.

- **IsInvoking(***methodenName* : `string`**)** : `bool`

  Diese Methode gibt dir zurück, ob derzeit ein Aufruf für die angegebene Methode mittels `Invoke` oder `InvokeRepeating` geplant wurde.

5.6 Problemlösung

In diesem Abschnitt findest du Informationen zu häufiger auftretenden Fehlern und Ansätze, wie du sie lösen kannst.

5.6.1 Wort im Code ist rot unterstrichen

Wenn ein Wort in Visual Studio rot unterstrichen ist, stimmt etwas mit der Variablen oder der Methode nicht. Um herauszufinden, was genau nicht stimmt, kannst du mit der Maus auf das Wort zeigen. Visual Studio öffnet dann ein kleines Hilfsfenster, in dem der Grund für die Markierung steht. Wenn du den Fehler nicht behebst, resultiert das in der Regel in einem Compiler-Fehler, die wir uns in Kapitel 5.6.2 noch ansehen werden.

Nachfolgend findest du gängige Fehler, die Liste ist allerdings nicht vollständig.

„Der Name X ist im aktuellen Kontext nicht vorhanden“

- Die Variable, Methode, Klasse etc. kann nicht gefunden werden, prüfe den unterstrichenen Bereich auf Tippfehler.
- Wenn es eine Variable ist, prüfe den Variablennamen auf Tippfehler. Prüfe die Deklaration der Variablen auf Fehler (siehe 5.2.3).
- Wenn du auf ein Objekt oder eine Klasse zugreifst, prüfe, ob der benötigte Namespace mit einer `using`-Anweisung geladen wird (siehe 5.2.9.1).

„[Zeichen] Erwartet“

- Du hast an der markierten Stelle vermutlich das angegebene Zeichen vergessen. Häufig ist dies eine Klammer oder ein Semikolon.

- Wenn du eine Klammer vergessen hast, prüfe, ob die Stelle, die Visual Studio vorschlägt, auch korrekt ist. Für jede geöffnete Klammer muss auch eine schließende Klammer vorhanden sein und anders herum.

Ein großer Abschnitt ist rot unterstrichen

- Du hast vermutlich an einer Stelle eine Klammer oder ein paar Anführungsstriche vergessen. Prüfe deinen Code dahingehend.

5.6.2 Fehler in der Console im Editor

Wenn du Fehler in der *Console* siehst, während du dein Spiel gerade nicht testest, handelt es sich dabei entweder um Compiler-Fehler oder um einen Fehler im Unity-Editor. Hin und wieder wirft Unity interne Fehler auf, weil es eine interne Datei nicht öffnen oder den Unity-Server nicht erreichen kann. Diese Fehler kannst du in der Regel aber erst einmal ignorieren. Klicke als Erstes in der *Console* auf CLEAR, um diese Fehler zu löschen. Wichtige Fehler tauchen wieder auf.

Die Fehler, die durch ein Leeren der *Console* nicht verschwinden, sind in der Regel *Compiler*-Fehler. Diese beziehen sich typischerweise auf eine bestimmte Script-Datei und bemängeln zum Beispiel ein *Unexpected Symbol*. Mit einem Doppelklick auf die Meldung öffnet sich dann das betroffene Script in Visual Studio. Dort sollte die fehlerhafte Stelle rot unterstrichen sein. Für weitere Hilfe, wie du das Problem lösen kannst, kannst du in Kapitel 5.6.1 nachlesen.

5.6.3 Fehler in der Console, während das Spiel läuft

Fehler, die während des Spiels auftreten, sind in der Regel *Exceptions*. Diese treten, wie schon in Kapitel 5.2.12 beschrieben, auf, wenn dein Script versucht, etwas auszuführen, das nicht möglich ist. Es gibt eine große Anzahl an unterschiedlichen Exceptions, zumal jeder Entwickler auch seine eigenen definieren kann. Nachfolgend stelle ich dir einmal die Exceptions vor, die am häufigsten auftreten. Tritt eine Exception auf, stehen für gewöhnlich das betroffene Script und die Zeile in der Console-Meldung.

Null Reference Exception

Das ist wohl die Exception, die am häufigsten von allen auftritt. Du versuchst auf eine Variable zuzugreifen, die noch nicht initialisiert wurde und deshalb den Wert `null` hat.

- Überprüfe die betroffene Stelle im Code.
- Überprüfe, ob du im Inspector alle Werte zugewiesen hast.
- Überprüfe, ob GameObjects und Components, die du versuchst, über deinen Code zu finden, auch wirklich in der Scene vorhanden sind.

Index Out Of Range Exception

Du versuchst, auf einen Array oder eine Liste zuzugreifen, und der Index, den du verwendest, ist größer als der höchste Index der Liste.

- Überprüfe die betroffene Stelle im Code und den Index, den du verwendest.

Divide By Zero Exception

Du versuchst, durch 0 zu dividieren, was mathematisch nicht möglich ist.

- Überprüfe die Rechnung an der betroffenen Stelle im Code.

Invalid Cast Exception

Du verwendest einen expliziten Cast, allerdings ist der Cast nicht möglich.

- Überprüfe die betroffene Stelle und überprüfe, ob ein Cast von der Quellklasse in die Zielklasse möglich ist.

5.6.4 „Can't add script"-Meldung oder Script ist nicht im „Add Component"-Menü vorhanden

Die Meldung „Can't add script", wie du sie in Bild 5.12 links siehst, erhältst du typischerweise, wenn du versuchst, ein Component aus deinem *Project Browser* zu einem GameObject hinzuzufügen. Erhältst du diese Meldung, ist es in der Regel auch nicht möglich, das Script über die ADD COMPONENT-Schaltfläche zu einem GameObject hinzuzufügen, weil es hier erst gar nicht aufgeführt wird.

- Als Erstes solltest du die *Console* prüfen, ob hier irgendwelche Fehler vorhanden sind. Wenn ja, behebe sie und versuche es erneut.
- Sind keine Fehler in der Console vorhanden, liegt es an einem Fehler mit dem jeweiligen Script, das du versuchst hinzuzufügen. In den meisten Fällen liegt es daran, dass der *Dateiname* nicht mit dem *Klassennamen* in der Script-Datei übereinstimmt. Passe den Datei- oder Klassennamen an, um das Problem zu beheben. Siehe Bild 5.12 für mehr Details.

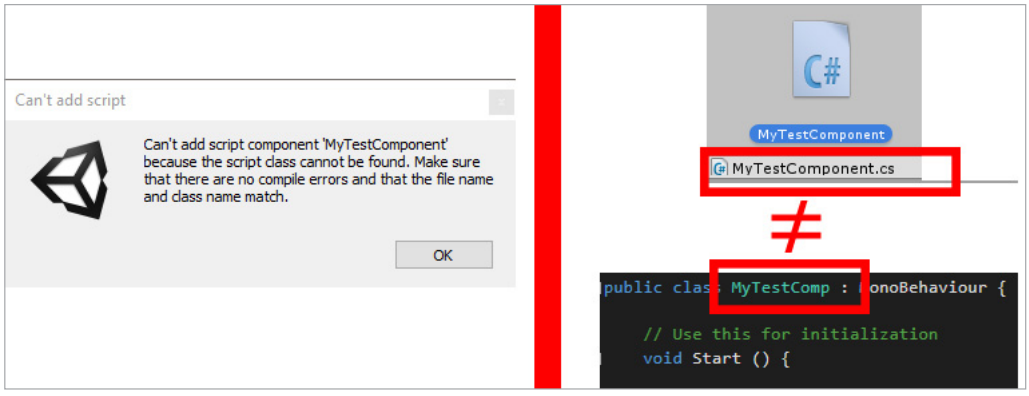

Bild 5.12 Kannst du ein Script nicht zu einem GameObject hinzufügen, überprüfe den Datei- und Klassennamen. Sie müssen identisch sein.

5.6.5 Sonstige Fehler

An dieser Stelle noch ein wichtiger Tipp für Fehlermeldungen, die hier nicht aufgelistet sind: Unity ist eine weit verbreitete Software, das bedeutet, in den meisten Fällen bist du nicht der Erste, der diese Fehlermeldung bekommt, auch wenn der Fehler hier nicht aufgeführt wird.

Um die Ursache und eine Lösung für Fehler zu finden, die hier nicht aufgelistet sind, reicht es deswegen häufig, den Fehlertext in einer Internet-Suchmaschine, zum Beispiel Google, einzugeben. Häufig findest du dann bereits andere Entwickler, die dasselbe Problem hatten, und kannst lesen, wie sie das Problem lösen konnten.