Arrays

Anleitung am Beispiel von Tic Tac Toe

Otto-Bennemann-Schule

Deutsch

Inhalt

[2 Bedeutung von Arrays in der Programmierung 2](#_Toc475908227)

[2.1 Was sind Arrays 2](#_Toc475908228)

[2.2 Arrays in C# 2](#_Toc475908229)

[2.2.1 Deklarierung von Arrays 2](#_Toc475908230)

[2.2.2 Inline Deklarierung 3](#_Toc475908231)

[2.2.3 Jagged und Zweidimensionale Arrays 3](#_Toc475908232)

[2.3 Arrays in moderner Programmierung 3](#_Toc475908233)

[3 Quellcode Dokumentation 4](#_Toc475908234)

[3.1 Klassenübersicht 4](#_Toc475908235)

[3.1.1 Main 4](#_Toc475908236)

[3.1.2 Game 4](#_Toc475908237)

[3.2 Spiellogik (Game Klasse) 4](#_Toc475908238)

[3.2.1 Properties 4](#_Toc475908239)

[3.2.2 Methoden 5](#_Toc475908240)

[3.2.3 Anzeige / User Interface (Main Klasse) 6](#_Toc475908241)

[4 Fazit 8](#_Toc475908242)

[5 Weitere Dokumentation und Quellcode 8](#_Toc475908243)

[6 Lizenz 8](#_Toc475908244)

[7 Literaturverzeichnis 9](#_Toc475908245)

[8 Abbildungsverzeichnis 9](#_Toc475908246)

# Bedeutung von Arrays in der Programmierung

## Was sind Arrays

Aus einfacher Sicht sind Arrays Listen. Bei genauer Betrachtung sind Arrays Datenstrukturen, die mehrere Variablen eines Datentyps speichern können (Louis, 2015).

Ein Array ist eine der einfachsten Listen, die einige Einschränkungen hat. So müssen bei einem Array nicht nur der Typ, sondern auch die Anzahl der Elemente angegeben werden. Die Länge und der Typ müssen dabei bereits beim Erstellen des Arrays feststehen und können später in der Regel nicht geändert werden. Dies hat den Hintergrund, dass der entsprechende Speicher dem Array zugewiesen werden muss.

intArray = new int[4];

intArray[0] = 300;

intArray[1] = 301;

intArray[2] = 302;

intArray[3] = 303;

0 ┌───────────────────┐

│ 4 │ -- length of array

4 ├───────────────────┤

│ 300 │ -- `intArray[0]`

8 ├───────────────────┤

│ 301 │ -- `intArray[1]`

12 ├───────────────────┤

│ 302 │ -- `intArray[2]`

16 ├───────────────────┤

│ 303 │ -- `intArray[3]`

20 └───────────────────┘

Um auf ein Element des Arrays zuzugreifen, muss der Index (Position des Elements) angegeben werden. Da durch den Variablentyp die Länge eines Wertes bekannt ist, kann durch Multiplikation die Position des Wertes oder die Referenz errechnet werden: Das erlaubt den Zugriff auf diesen Wert. Aus diesem Grund ist das erste Element des Arrays auch an Position 0 zu finden.

Abbildung 1: Aufbau eines Arrays im Speicher (Daddy, 2009)

## Arrays in C#

### Deklarierung von Arrays

Arrays in C# funktionieren ähnlich wie in den meisten Programmiersprachen. Der Variable wird mit dem Datentyp, gefolgt von zwei eckigen Klammern und dem Variablennamen, deklariert. Die Wertzuweisung erfolgt mit dem ‚new‘ Keyword, gefolgt von dem Datentyp und der Länge in eckigen Klammern. Um Werte zuzuweisen oder auszulesen, wird der Variablenname mit dem Index in eckigen Klammern benutzt.

int[] variablenName;

variablenName = new int[5];

variablenName[0] = 42;

### Inline Deklarierung

Da dieser manuelle Weg der Wertzuweisung sehr unhandlich sein kann, gibt es eine alternative Schreibweise, welche die Erstellung von Arrays mit statischen Werten vereinfacht. Dafür gibt man bei der Initialisierung die Werte einfach in geschweiften Klammern und mit Komma getrennt an.

### Jagged und Zweidimensionale Arrays

Um mehrere Dimensionen / Spalten darzustellen, gibt es in C# zwei verschiedene Konzepte. Sollte in jeder Zeile gleich viele Spalten existieren, so spricht man von zweidimensionalen Arrays. Zweidimensionale Arrays werden mit einem Komma in den eckigen Klammern gekennzeichnet und auf die Werte kann mit einem Komma getrennte Indizes in den Klammern zugegriffen werden.

Jagged Arrays haben unterschiedliche viele Spalten in den Zeilen. Das bedeutet, in jedem Element des Arrays ist ein weiteres Array, welches eine andere Länge als das „Haupt“-Array hat. Bei einem Jagged Array kann bei der Initialisierung nur die Länge des Hauptarrays festgelegt werden. Alle Arrays in diesem Array müssen extra deklariert werden. Um ein Jagged Array zu definieren, werden die geöffneten und geschlossenen eckigen Klammern zwei Mal hintereinandergeschrieben. Für den Zugriff muss in jede der Klammern der Index geschrieben werden.

int[,] multidimensionalArray = new int[10, 5];

multidimensionalArray[0, 1] = 3;

int[][] jaggedArray = new int[10][];

jaggedArray[0] = new int[5];

jaggedArray[0][1] = 3;

## Arrays in moderner Programmierung

Wie bereits angesprochen, kann die Länge eines Arrays in der Regel nicht geändert werden. Wenn man nun also einen Wert hinzufügen oder entfernen möchte, müsste man ein neues Array erstellen und alle übrigen Werte in dieses kopieren. Ein ähnliches Problem hat man beim Sortieren von Arrays, da beim Einfügen / nach vorne Sortieren alle hinteren Elemente einzeln aufgerückt werden müssen. Dies verursacht besonders bei langen Arrays enorme Performanceprobleme. Aus diesem Grund werden Heutzutage in vielen Anwendungsfällen eher Listen als Arrays verwendet. Eine Liste kann ohne Probleme erweitert und sortiert werden, da weder die Anzahl der Elemente noch deren Reinfolge fix definiert werden.

# Quellcode Dokumentation

Im Folgenden wird der Quellcode und die Funktionsweise des Beispielprogramms ‚Tic Tac Toe‘ dokumentiert. Die Spielregeln des Spiels werden dabei als bekannt vorausgesetzt. Tastenkombinationen werden: Pfeiltasten / WASD für Cursor Navigation, Enter / Leertaste für Bestätigen, F5 für ein neues Spiel und Escape zum Beenden.

## Klassenübersicht

### Main

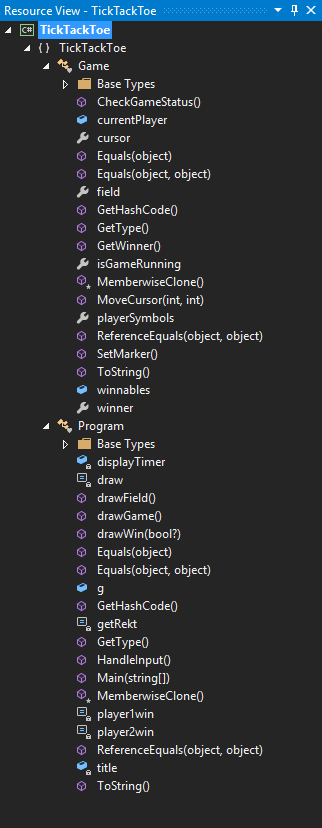
Die Main Klasse stellt den Einstiegspunkt der Anwendung da. Sie beinhaltet die *Main(string[] args)* Methode, welche die Hauptroutine des Programmes ist. Zusätzlich wird in dieser Klasse das Userinterface ausgegeben und der Input an die Spiellogik weitergegeben.

Abbildung 2: Übersicht über die Ressourcen. (Für genauere Details siehe Docs auf github)

### Game

Die Game-Klasse beinhaltet die gesamte Logik, welche für das Spiel relevant ist. Es findet keine Ausgabe an den User statt. Dies ist Aufgabe der Main Klasse. Dadurch ist Trennung von Interface und Logik gegeben.

## Spiellogik (Game Klasse)

### Properties

Als Grundlage für die Logik des Spiels sind mehrere Eigenschaften deklariert. Das Spielfeld ist durch ein Zweidimensionales Array abgebildet, da sowohl jede Spalte als auch jede Zeile drei Felder hat. Als Datentyp ist ein Nullable<bool> gesetzt. Dies erlaubt sowohl false (Player 1), true (Player 2) als auch null, was einem noch nicht besetzten Spielfeld entspricht. Da der Standardwert null ist, wird das Spielfeld automatisch leer erstellt.

Zusätzlich wird ein Array mit den Symbolen für die verschiedenen Spieler angelegt. Dieses Array hat an Position 0 und 1 die Symbole für den ersten und zweiten Spieler und an Position 2 das Symbol für ein freies Feld. Diese Belegung erlaubt es das Spielersymbol durch einen booleschen Wert, wie er auch im Spielfeld abgelegt wird und Position 2 als extra Element abzufragen.

Der Cursor beschreibt die Position im Spielfeld, welche momentan ausgewählt ist. Da es die X und Y Koordinaten angibt, kann dies in einem Array der Länge 2 abgebildet werden und standardmäßig auf die Position 0/0 gesetzt werden.

Für die Ermittlung des Gewinners wird zum einen der Gewinner (soweit es einen gibt), ob das Spiel noch läuft und mit welchen Kombinationen gewonnen werden kann, gespeichert.

Zum Abschluss wird noch der momentan aktive Spieler gespeichert und standardmäßig auf den Spieler 1 gesetzt.

public char[] playerSymbols { get; } = { 'X', 'O', ' ' };

public bool?[,] field { get; private set; } = new bool?[3, 3];

public int[] cursor { get; private set; } = { 0, 0 };

public bool currentPlayer = false;

public bool? winner { get; private set; } = null;

public bool isGameRunning { get; private set; } = true;

public int[][] winnables =

{

/\* SPOILER: Possible combinations \*/

};

### Methoden

Da alle Properties nicht von außen gesetzt werden können, um Logik von Interface zu trennen, müssen alle Änderungen durch die öffentlichen () Methoden vorgenommen werden. Jede Eigenschaft hat jedoch einen öffentlichen Getter[[1]](#footnote-1), welcher das Betrachten des momentanen Spielstandes erlaubt.

#### Platzieren von Spielsteinen

Zum Platzieren eines Spielsteines muss der Spieler den Cursor auf das gewünschte Feld navigieren und dann das Setzen seines Spielsteins bestätigen. Dafür gibt es zwei Methoden, die genau diese Funktionen abbilden.

Zum Navigieren mit dem Cursor (WASD oder Pfeiltasten) wird die Methode *MoveCursor(int x, int y)* mit der Änderung in X und Y Position aufgerufen. In der Methode wird zuerst überprüft, ob die neue Position noch im Spielfeld liegt. Nur wenn dies Zutrifft wird die Position auch geändert.

Da durch die Methode zum Navigieren bereits sichergestellt wurde, dass die Cursor Position gültig ist, braucht die Methode zum Setzen des Spielsteins nur noch überprüfen, ob an der gegebenen Position schon ein Spielstein ist. Nur wenn dort noch kein Stein platziert wurde, wird der Wert im Spielfeld geändert.

// Check if values are in bound

if (x < -1 || x > 1 || y < -1 || y > 1) return;

if (cursor[0] + x < 0 || cursor[0] + x > 2) return;

if (cursor[1] + y < 0 || cursor[1] + y > 2) return;

// Set cursor position

cursor[0] += x;

cursor[1] += y;

#### Gewinnprüfung

Bei der Überprüfung, ob ein Spieler gewonnen hat, gibt es mehrere mögliche Ansätze. Da sich die Anzahl der möglichen Kombinationen, die zu einem Gewinn führen, in Grenzen halten (3 Vertikal, 3 Horizontal und 2 Diagonal), werden diese Kombinationen einfach in einem Jagged Array abgelegt. Zur Überprüfung des Gewinners, muss jetzt einfach für jede dieser Gewinnkombinationen überprüft werden, ob alle drei Felder in der Kombination von demselben Spieler besetzt wurden. Ist dies der Fall, so hat dieser Spieler gewonnen.

##### Edgecases[[2]](#footnote-2)

Zu einem Edgecase kommt es bei der Überprüfung des Gewinnes. Denn es gibt nicht nur die Möglichkeit, dass Spieler 1 oder Spieler 2 gewinnen, sondern auch ein Unentschieden ist möglich.

Da allerdings der null Wert für den Zustand (noch kein Sieger) definiert wurde, kann dies nicht der Rückgabewert für ein Unentschieden sein. Nach unserer Logik würde dieses Spiel also nie enden, da es keinen Gewinner geben kann.

Die Lösung für dieses Problem ist die Einführung eines weiteren Booleschen Wertes, welcher das Ende eines Spiels überprüft. Wird bei der Überprüfung der Gewinnbedingungen immer noch als Gewinner null zurückgegeben, wird beim Spielende ein Flag gesetzt.

// Finish game when all fields are occupied

if (!field.Cast<bool?>().Any(v => !v.HasValue)) { this.isGameRunning = false; }

### Anzeige / User Interface (Main Klasse)

Die Hauptaufgabe der Main Klasse ist die Anzeige der Benutzeroberfläche und das Verarbeiten von Benutzereingaben. Dafür existiert ein Updateloop[[3]](#footnote-3), welcher sowohl für das neu zeichnen des Interfaces als auch das Erkennen von Eingaben zuständig ist. Um ein Flackern oder Mehrfacheingaben zu verhindern, wird zusätzlich eine kurze Zeitverzögerung eingebaut.

#### Benutzeroberfläche

Die Benutzeroberfläche liegt in zwei Zuständen vor. Das Spielfeld an sich und wenn das Spiel beendet ist – Der End-Bildschirm.

Im Spielfeldmodus wird in der ersten Zeile der momentan aktive Spieler angezeigt. Darunter ist das Spielfeld zu finden. Um das Spielfeld zu zeichnen, wird zuerst über die erste Dimension des

Abbildung 3: Spielfeld mit einigen Spielsteinen schon gesetzt.

Spielfeld- Arrays iteriert[[4]](#footnote-4). Dabei wird jeweils eine neue Zeile in der Konsole erstellt und dann für jedes dieser Elemente über die zweite Dimension iteriert. So kann für jedes Feld ein Kästchen (dargestellt durch eckige Klammern) mit dem Spielstein, soweit einer platziert ist, ausgegeben werden.

Wenn die Position des Kästchens, der Position des Cursors, entspricht, wird das Kästchen und der Spielstein farbig dargestellt.

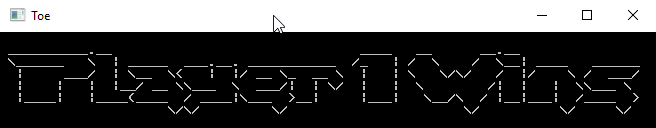
Nach Beendung des Spiels wird der Gewinner durch einen Text angezeigt. Die möglichen Gewinnzustände sind entweder Spieler 1, oder Spieler 2. Hat keiner der beiden Spieler gewonnen, so ist der dritte Zustand ein Unentschieden.

Abbildung 4: End-Zustand zeigt Gewinner an.

#### Benutzereingaben

Benutzereingaben werden in einem separaten Prozess entgegengenommen. Das Einlesen von Eingaben würde ansonsten die Aktualisierungsanzeige verhindern. Sobald eine Eingabe erkannt wurde, wird abhängig von der gedrückten Taste, die entsprechende Aktion ausgeführt.

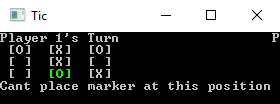
* ENTER oder LEERTASTE: Es wird die Spielstein setzen Funktion aus der Game Klasse aufgerufen und ein Feedback angezeigt, wenn es fehlschlägt.

Abbildung : Fehlermeldung, wenn schon ein Stein platziert ist.

* RICHTUNGSTASTEN oder WASD oder NUMBLOCK (2486): Die Cursor-beweg Funktion der Game Klasse wird aufgerufen.
* F5: Startet ein neues Spiel, indem eine neue Instanz[[5]](#footnote-5) der Game Klasse erzeugt wird.
* ESCAPE: Schließt die Anwendung

Das Spiel an sich gib also keine „Befehle“ an die Oberfläche. Alle Änderungen an der Oberfläche werden durch Änderungen am Status des Spiels hervorgerufen.

# Fazit

Zusammenfassend lässt sich sagen, dass für Listen mit statischer Größe, ein Array eine einfache und effiziente Möglichkeit der Abbildung im Programm darstellt.

Sobald das Programm jedoch in Komplexität steigt und weitere Anforderungen an die Liste hinzukommen, sollte zu einem dynamischeren Datentyp gegriffen werden. Dieser Datentyp hat den Vorteil, dass das Hinzufügen von Elementen an beliebigen Indizes erlaubt wird. Nur so kann man eine performante Umgebung schaffen, welche sich gut skalieren lässt. Selbst bei großen Datenmengen wird die Liste somit nicht zum „Schwachpunkt des Programmes“. In dem verwendeten Beispiel hat das noch ganz gut funktioniert, wäre aber mit einer List deutlich einfacher umzusetzen gewesen.

# Weitere Dokumentation und Quellcode

Der verwendete Quellcode und eine Dokumentation können auf meiner GitHub Seite unter folgender URL eingesehen werden:

GitHub: <https://github.com/Xenira/school-tic-tac-toe>  
Dokumentation: <https://xenira.github.io/school-tic-tac-toe/Docu/html/annotated.html>

# Lizenz

Sowohl diese Ausarbeitung als auch der Sourcecode stehen unter der MIT Lizenz. Der volle Lizenztext befindet sich im GitHub unter folgender Adresse: <https://github.com/Xenira/school-tic-tac-toe/blob/master/LICENSE>

# Literaturverzeichnis

chakrit. (11. April 2009). *c# - Checking for winner in TicTacToe? - Stack Overflow*. Von Stack Overflow: http://stackoverflow.com/questions/740467/checking-for-winner-in-tictactoe#740473 abgerufen

Daddy, P. (11. July 2009). *c# - Arrays, heap and stack and value types - Stack Overflow*. Von Stack Overflow: http://stackoverflow.com/questions/1113819/arrays-heap-and-stack-and-value-types abgerufen

Doxygen. (29. Dezember 2016). *Doxygen: Main Page*. Von Doxygen: http://www.stack.nl/~dimitri/doxygen/ abgerufen

Louis, D. (2015). Easy C++. In D. Louis, *Easy C++* (S. 216). 219: Markt + Technik Verlag.

patorjk.com. (kein Datum). *Text to ASCII Art Generator (TAAG)*. Von Text to ASCII Art Generator: http://www.patorjk.com/software/taag/ abgerufen

**Anmerkung:** Durch jahrelange Arbeit mit und im Internet kann es dazu gekommen sein, dass meine Meinungen und Ansichten durch den dort zu findenden Inhalt beeinflusst worden, was zu ähnlichen Formulierungen (Zumeist aus dem Englischen übernommen) oder Ausdrucksweisen geführt haben kann. Jegliche Referenzen außerhalb dieses Literaturverzeichnisses wurden nicht während der Erarbeitung dieser Ausarbeitung in Betracht gezogen und können unmöglich hier wiedergegeben werden, da dieser Teil meiner Bildung waren und nicht dokumentiert wurde und nicht ohne weiteres dokumentiert werden kann.

# Abbildungsverzeichnis

[Abbildung 1: Aufbau eines Arrays im Speicher (Daddy, 2009) 2](file:///D:\Development\school-tic-tac-toe\Array_Programmierung.docx#_Toc475908210)

[Abbildung 2: Übersicht über die Ressourcen. (Für genauere Details siehe Docs auf github) 4](file:///D:\Development\school-tic-tac-toe\Array_Programmierung.docx#_Toc475908211)

[Abbildung 3: Spielfeld mit einigen Spielsteinen schon gesetzt. 7](file:///D:\Development\school-tic-tac-toe\Array_Programmierung.docx#_Toc475908212)

[Abbildung 4: End-Zustand zeigt Gewinner an. 7](file:///D:\Development\school-tic-tac-toe\Array_Programmierung.docx#_Toc475908213)

[Abbildung 5: Fehlermeldung, wenn schon ein Stein platziert ist. 7](#_Toc475908214)

1. Ein Getter ist eine Methode, welche den Wert einer Variable zurückgibt. [↑](#footnote-ref-1)
2. Edgecases bezeichnen besondere Situationen, die nicht dem normalen Schema entsprechen [↑](#footnote-ref-2)
3. Loops sind sich wiederholende Vorgänge. Auch schleifen genannt. [↑](#footnote-ref-3)
4. In einer Schleife wird sich jedes Element angeguckt. [↑](#footnote-ref-4)
5. Eine Instanz ist eine Ausprägung einer Klasse. Wenn man so will [↑](#footnote-ref-5)