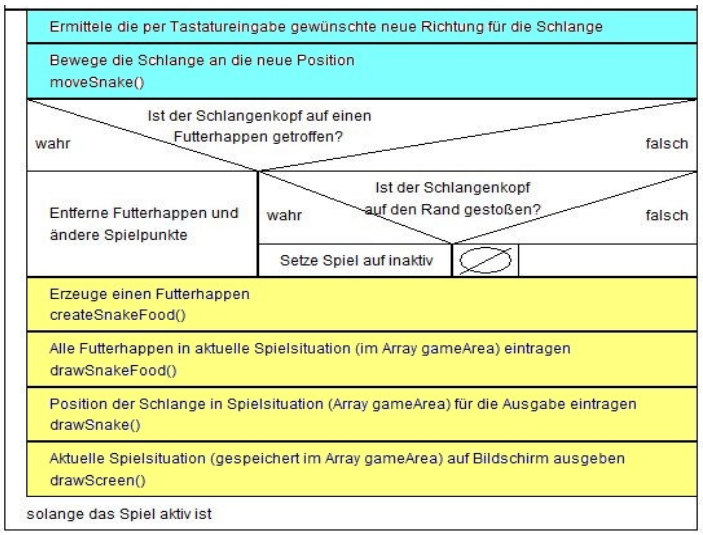
Die Schlange bewegt sich



Nun soll Bewegung in das Spiel kommen!

Hierzu ist nochmals die **`do...while`**-Schleife dargestellt, die den Spielablauf beschreibt.

Die gelb hinterlegten Blöcke wurden bereits in Teil 2 definiert. Die beiden cyanfarbenen Blöcke bilden das neue Thema, nämlich die Bewegungssteuerung und Bewegung der Schlange.

Dazu muss man sich überlegen, wie man die Bewegungssteuerung und die Bewegung der Schlange realisieren will.

Bewegungssteuerung

Fangen wir daher zuerst einmal damit an, welche Tasten zur Steuerung der Bewegungsrichtung verwendet werden sollen und wie man ermitteln kann, welche Taste gedrückt wurde bzw. ob keine Taste gedrückt wurde.

Nochmals zur Erinnerung: Die Schlange läuft in einer vorgegebenen Richtung los und bewegt sich immer weiter in diese Richtung, solange keine neue Richtung gewählt wird. Mögliche Bewegungsrichtungen für die Schlange sind: Nach links, nach oben, nach rechts oder nach unten.

Weiterhin gilt, dass die Schlange nicht entgegen ihrer aktuellen Bewegungsrichtung laufen darf!

Für die Steuerung bieten sich u.a. die vier Pfeiltasten an.

Zur Realisierung einer solchen Steuerung bietet das **`Console`**-Objekt eine nützliche Eigenschaft und eine Methode an:

- die Eigenschaft **`KeyAvailable`** hat den Datentyp **`bool`** und zeigt an, ob eine Taste gedrückt wurde oder nicht
- die Methode **`ReadKey()`** liest eine gedrückte Taste und liefert eine spezielle Datenstruktur mit dem Namen **`ConsoleKeyInfo`** zurück.

Sie werden vielleicht denken: „Das hört sich kompliziert an! Warum wird nicht einfach nur die gedrückte Taste zurück geliefert?“

Das liegt daran, dass man ja nicht nur eine einzelne Taste drücken kann, sondern man kann ja auch irgendwelche Tastenkombinationen drücken wie z.B. **`STRG-U`** oder **`ALT-W`**.

Diese spezielle Datenstruktur liefert eben auch diese zusätzlichen Informationen, so das nun auch komplexe Tastenkombinationen ausgewertet werden können! Näheres hierzu liefert die Hilfeseite zu dieser Datenstruktur[1].

Sehen Sie sich mit mir anhand des nachfolgenden Programmcodes das Prinzip an, wie man das nun implementiert und anschließend eine Auswertung der gedrückten Taste vornimmt.

In der Entscheidungsstruktur wird zuerst überprüft, ob überhaupt eine Taste gedrückt wurde. Falls die Eigenschaft **`KeyAvailabe`** den Wert **`true`** hat, wird der Anweisungsblock ausgeführt, sonst nicht.

Die erste Anweisung ist dann das Lesen einer Taste mit der Methode **`ReadKey()`**. Der Parame-

1 **`https://docs.microsoft.com/de-de/dotnet/api/system.consolekeyinfo?view=netcore-3.1`**, abgerufen am 28.05.2020

ter **`false`** bewirkt, dass eine gelesene Taste nicht im Konsolenfenster ausgegeben wird!

```
// Prüfen, ob eine Taste gedrückt wurde
if (Console.KeyAvailable)
{
    // Liest die gedrückte Taste ohne Anzeige im Konsolenfenster
    ConsoleKeyInfo pressedKey = Console.ReadKey(false);

    // Auswertung der gedrückten Taste
    switch (pressedKey.Key)
    {
        case ConsoleKey.LeftArrow:
            newDirection = 1;
            break;
        case ConsoleKey.UpArrow:
            // To Do
            break;
        case ConsoleKey.RightArrow:
            // To Do
            break;
        case ConsoleKey.DownArrow:
            // To Do
            break;
        case ConsoleKey.Escape:
            Environment.Exit(1);
            break;
        default:
            break;
    }
```

Diese Methode liefert die zuvor genannte Datenstruktur mit dem Datentyp **`ConsoleKeyInfo`**. Also kann dieser zurück geliefert Wert in einer solchen Variablen mit eben diesem Datentyp gespeichert werden. Der von mir gewählte Bezeichner für diese Variable lautet **`pressedKey`**.

Somit befindet sich die Information über die gedrückte Taste nun im Arbeitsspeicher in der Datenstruktur mit der Bezeichnung **`pressedKey`**.

Jeder Taste auf einer Tastatur ist in C# eine Bezeichnung zugeordnet. Man erhält diese Bezeichnung durch Eingabe von **`ConsoleKey`**, gefolgt von einem Punkt. Dann öffnet sich (ähnlich wie bei den Farben, wenn man **`ConsoleColor`** und einen Punkt eintippt) eine wirklich sehr lange Liste mit selbsterklärenden Bezeichnungen für jede mögliche Taste. Im Beispielprogramm oben sind die Bezeichnungen für die vier Pfeiltasten angegeben.

Außerdem ist auch noch die Bezeichnung für die **`Escape`**-Taste angegeben. Wird diese gedrückt, wird die Anweisung `Environment.Exit(1);` ausgeführt. Diese Anweisung beendet das Programm. Auch diese Anweisung für das (vorzeitige) Beenden des Konsolenprogramms ist neu.

Zurück zur Bewegungssteuerung! In Abhängigkeit von einer gedrückten Pfeiltaste soll sich ja u.U. die Bewegungsrichtung der Schlange ändern.

Aber woher wissen wir denn, in welcher Richtung sich die Schlange bisher bewegt?

Richtig! Das wissen wir noch gar nicht. Wir müssen diese Richtung ja irgendwie speichern! Bisher gibt es hierzu allerdings noch keine Information! Deshalb benötigen wir eine Variable, in der wir uns die aktuelle Bewegungsrichtung der Schlange merken. Nun ja, das kann man auf verschiedene Weisen machen.

Eine Idee wäre, dass man eine Variable vom Typ **`char`** nimmt und darin die Richtung der Schlange anhand des Anfangsbuchstabens der vier Himmelsrichtungen Norden, Osten, Süden und Westen speichert: Also **`'N'`**, **`'O'`**, **`'S'`**, **`'W'`**.

Eine andere Variante, für die ich mich entschieden habe[2], verwendet einen Integer-Wert. Hierbei entspricht die Zahl 1 der Richtung „links“, die Zahl 2 der Richtung „oben“, die Zahl 3 der Richtung „rechts“ und die Zahl 4 der Richtung „unten“.

Gespeichert wird dieser Wert in einer Integer-Variablen mit dem Bezeichner **`directionOfSnake`**. Diese Variable wird zu Beginn des Spiels erstellt und mit der „Wunschrichtung“ initialisiert.

2 Eine wesentlich bessere Variante wäre die Verwendung einer sogenannten „Enumeration“. Aber dazu mehr zu einem späteren Zeitpunkt :-)

In Abhängigkeit der gedrückten Taste, erhält diese Variable dann u.U. einen neuen Wert, der für die weitere Bewegung der Schlange verwendet wird. Dieser „neue“ Wert wird von der Methode geliefert, in der nun die Bewegungssteuerung durchgeführt wird.

Für diese Methode habe ich den Bezeichner **`checkKeyboardAndSetNewDirection`** gewählt.

Welcher Datentyp soll für die Rückgabe gewählt werden und welche Parameter sind erforderlich?

Nun, die Methode soll die „neue“ Bewegungsrichtung der Schlange als Integer-Wert (1 bis 4) zurückgeben. Somit ist klar, dass der Rückgabetyp **`int`** lautet.

Als Information benötigt die Methode die bisherige Bewegungsrichtung der Schlange, also den aktuellen Wert von **`directionOfSnake`**, damit geprüft werden kann, ob die neu gewählte Bewegungsrichtung möglich ist. Der Methodenkopf muss also folgendermaßen aussehen:

```
static int checkKeyboardAndSetNewDirection(int directionOfSnake)
```

Innerhalb der Methode wird eine Variable zur Aufnahme der gerade gewählten Bewegungsrichtung erstellt. Diese Integer-Variable erhält den Bezeichner **`newDirection`**.

Bei der Auswertung der gedrückten Taste im Rahmen der **`switch`**-Struktur erhält diese Variable nun den betreffenden Wert.

Anschließend muss überprüft werden, ob der neue Wert für die Bewegungsrichtung ein anderer Wert ist, wie der Wert für die bisherige Bewegungsrichtung. Ist dies nämlich nicht der Fall, kann die Überprüfung, ob die neue Bewegungsrichtung entgegen der bisherigen Bewegungsrichtung ist, entfallen!

Sind die beiden Richtungen jedoch unterschiedlich, muss eben überprüft werden, ob die neue Bewegungsrichtung entgegen der bisherigen Bewegungsrichtung ist. In diesen vier Fällen bleibt die bisherige Bewegungsrichtung erhalten. Hier gibt es verschiedene Möglichkeiten dies zu prüfen.

In jedem Fall erhält der Parameter **`directionOfSnake`** die neue oder bisherige Bewegungsrichtung der Schlange und wird mittels **`return`** zurückgeliefert. Dieser Ergebniswert kann nun bei dem Aufruf der Methode der Variablen **`directionOfSnake`** zugewiesen werden.

Der Aufruf in der **`do...while`**-Schleife in der **`Main`**-Methode sieht dann wie folgt aus:

```
directionOfSnake = checkKeyboardAndSetNewDirection(directionOfSnake);
```

Bewegen der Schlange

So...die neue (oder bisherige) Bewegungsrichtung ist in der Variablen **`directionOfSnake`** nun gespeichert. Jetzt muss die Schlange dementsprechend an die neue Position bewegt werden.

Nochmals zur Erinnerung: Die Positionen des Schlangenkörpers befinden sich ja in einer Datenstruktur, die für jedes Körperteil der Schlange dessen Zeilen- und Spaltennummer speichert. Der Bezeichner dieser Datenstruktur ist **`positionsOfSnake`**. Es handelt sich dabei um ein zweidimensionales Array mit einer gewissen Anzahl an Zeilen und zwei Spalten.

Die Anzahl der Zeilen in diesem Array bestimmt die Anzahl der Körperelemente und wird durch die Konstante **`maxLengthOfSnake`** bestimmt.

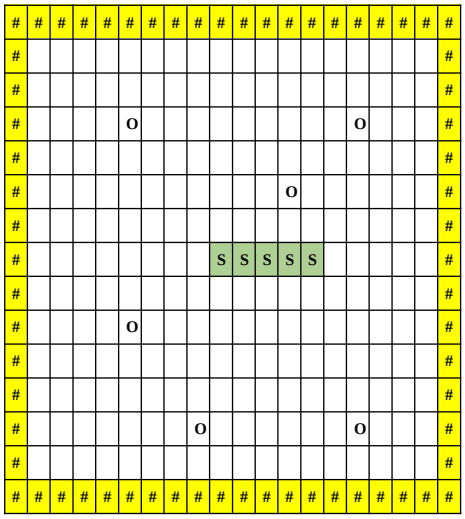
Die erste Spalte in diesem Array (mit dem Index 0) enthält die Zeilennummer und die zweite Spalte (mit dem Index 1) enthält die Spaltennummer für das betreffende Körperteil.

Der erste Eintrag in diesem Array enthält die Position des Schlangenkopfes. Alle nachfolgenden

Körperelemente haben einen Zeilen- und Spaltenwert, der größer Null ist. Die bisher unbenutzten Arrayelemente haben für Zeile und Spalte den Wert Null. Die aktuelle Länge der Schlange wird in einer Variablen mit dem Bezeichner **`lengthOfSnake`** gespeichert. Soviel zu den bisherigen Entscheidungen zur Speicherung von Informationen über die Schlange (siehe hierzu Teil 2).

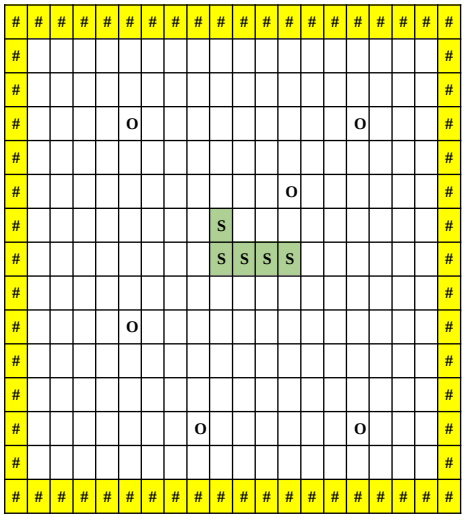
Nun soll die Schlange sich bewegen. Das bedeutet, dass in dem Array **`positionsOfSnake`** Änderungen an den Positionen aller Körperelemente durchgeführt werden müssen.

Sehen wir uns das an einem Beispiel an. Zur besseren Sichtbarkeit sind die aktuellen Positionen des Schlangenkörpers grün unterlegt. Der Schlangenkopf ist links. Die Tabelle zeigt die momentanen Positionen des Schlangenkörpers. Die Länge der Schlange ist 5. Deshalb sind die ersten fünf Arrayelemente mit Werten größer Null belegt. Alle nachfolgenden Arrayelemente haben für Zeile und Spalte den Wert Null.



Zeile	Spalte
7	9
7	10
7	11
7	12
7	13
0	0
0	0
0	0
0	0
0	0

Jetzt soll sich die Schlange bewegen. Gewählt wurde die neue Bewegungsrichtung „oben“, d.h. die Variable **`directionOfSnake`** hat den Wert 2. Somit sieht die neue Position wie folgt aus. Wie wirkt sich dies auf die Tabelle mit den neuen Positionsangaben aus?



Zeile	Spalte
6	9
7	9
7	10
7	11
7	12
0	0
0	0
0	0
0	0
0	0

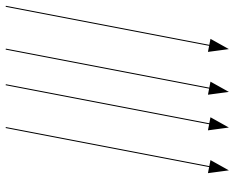
Aufgrund der Bewegungsrichtung „oben“ ändert sich die Zeilenposition für den Schlangenkopf; und zwar muss die Zeilennummer um Eins vermindert werden. Wäre die Bewegungsrichtung „unten“, würde die Zeilennummer um Eins erhöht werden. Hätte sich die Bewegungsrichtung nicht geändert, bliebe die Zeilennummer gleich, aber die Spaltennummer müsste um Eins vermindert

werden. Angenommen, die Bewegungsrichtung zuvor wäre „rechts“ gewesen, würde die Zeilennummer ebenfalls gleich bleiben und die Spaltennummer um Eins erhöht werden.

Für den Schlangenkopf ändert sich also immer nur eine Angabe, entweder die Zeilennummer oder die Spaltennummer und dann auch nur um den Wert „+1“ oder „-1“.

Das beschreibt aber nur die Positionsänderung des Schlangenkopfes. Wie bewegt sich aber nun der Körper? Schaut man sich die beiden Tabellen an (linke Tabelle: alte Positionen, rechte Tabelle: neue Positionen), erkennt man, dass das erste Körperelement der „neuen“ Schlange nun die Position des „alten“ Schlangenkopfes hat, das zweite Körperelement „rutscht“ auf die Position des vorher ersten Körperelementes usw. Die Körperelemente „rutschen“ also quasi dem neuen Schlangenkopf nach.

Zeile	Spalte
7	9
7	10
7	11
7	12
7	13
0	0
0	0
0	0
0	0
0	0



Zeile	Spalte
6	9
7	9
7	10
7	11
7	12
0	0
0	0
0	0
0	0
0	0

Die Bewegung der Schlange, also die Änderung der Positionen, ist also eigentlich nur ein Umkopieren der „alten“ Positionsangaben um eine Zeile nach unten und das Ersetzen der Position des Schlangenkopfes durch die neuen Positionsangaben.

Easy...oder?

Nicht ganz...für die Implementation muss man jetzt noch auf Folgendes aufpassen, da wir ja die Änderungen direkt in dem Array **`positionsOfSnake`** durchführen, quasi eine Operation am „offenen Herzen“ :-) Wir dürfen dabei keine benötigten Informationen überschreiben!

Wenn wir mit den alten Schlangenkopf nun in die Zeile darunter kopieren, überschreiben wir damit die „alte“ Position des ersten Körperelementes. Das funktioniert also nicht!

Wir müssen das Umkopieren am Ende starten! Die Wiederholungsstruktur und das ist in diesem Fall wieder eine **`for`**-Schleife, da ja aufgrund der Länge der Schlange bekannt ist, wie viele Umkopiervorgänge durchzuführen sind muss also die beiden Werte der letzten Zeile durch die beiden Werte in der vorletzten Zeile ersetzen, dann die beiden Werte der vorletzten Zeile durch die der vorvorletzten Zeile usw.

Nach der **`for`**-Schleife werden die beiden Werte in der ersten Zeile (Index 0) durch die Positionsangaben für den neuen Schlangenkopf ersetzt. Diese beiden Werte muss man sich hierzu gemerkt haben!

Nun kann die Implementation erfolgen. Die Methode zum Bewegen der Schlange soll den Bezeichner **`moveSnake`** erhalten. Wie immer, müssen wir überlegen, welchen Rückgabetyp die Methode haben soll/muss und welche Parameter erforderlich sind.

Die Methode macht ja eigenlich nichts anderes, als in dem Array **`positionsOfSnake`** irgend-

welche Werte zu kopieren. Ein Ergebnis entsteht nicht wirklich und es muss auch keine Meldung in irgendeiner Form zurück gegeben werden. Deshalb wieder mal der Datentyp **`void`**.

Als Parameter benötigt die Methode die aktuellen Positionsangaben zur Schlange in Form des Array **`positionsOfSnake`**, die aktuelle Länge der Schlange in **`lengthOfSnake`** und offensichtlich die Bewegungsrichtung der Schlange in **`directionOfSnake`**.

Der Methodenkopf sollte also folgendermaßen aussehen:

```
static void moveSnake(int[,] positionsOfSnake, int lengthOfSnake, int directionOfSnake)
```

Nun gilt es die Methode zu implementieren. Nachfolgend ist eine unvollständige Beispiel-Implementation gegeben.

Zu Beginn der Methode wird die aktuelle Position des Schlangenkopfes in einem zweidimensionalen Array gespeichert, da diese ja später benötigt wird.

Anscjließend wird anhand der Richtung die neue Position des Schlangenkopfes festgelegt. Für die Richtung „links“ ist dies bereits angegeben (in Kurzschreibweise!). Der Spaltenwert wird hierbei um Eins vermindert. Für die anderen Richtungen ist der Programmcode zu ergänzen.

Nach Festlegung der neuen Position des Schlangenkopfes können nun die Positionen der Körperelemente wie oben dargestellt quasi verschoben werden. Hier sind die Zeilen- und Spaltenpositionen zu kopieren.

Und zum Schluss muss in dem Array noch die gespeicherte neue Position des Schlangenkopfs eingetragen werden.

Ergänzungen sind an allen Stellen mit dem Kommentar **`// To Do`** durchzuführen.

```
static void moveSnake(int[,] positionsOfSnake,
                      int lengthOfSnake,
                      int directionOfSnake)
{
    // Aktuelle Position des Schlangenkopf merken
    int[,] newPositionOfSnakeHead = new int[1, 2];
    newPositionOfSnakeHead[0, 0] = positionsOfSnake[0, 0];
    newPositionOfSnakeHead[0, 1] = positionsOfSnake[0, 1];

    // Neue Position des Schlangekopfes anhand
    // der Richtung festlegen
    switch (directionOfSnake)
    {
        // Links
        case 1:
            newPositionOfSnakeHead[0, 1] -= 1;
            break;
        // Aufwärts
        case 2:
            // To Do
            break;
        // Rechts
        case 3:
            // To Do
            break;
        // Abwärts
        case 4:
            // To Do
            break;
    }

    // Hier wird nach einer Positionsänderung
    // die Lage der Schlange neu gespeichert

    // Verschieben der Schlangenkörperpositionen
    // beginnend am Ende
    for (int i = lengthOfSnake - 1; i > 0; i--)
    {
        // To Do
    }

    // Neue Position des Schlangenkopfes speichern
    // To Do
}
```

Aufgaben

1. Implementieren Sie die Methode **`checkKeyboardAnSetNewDirection`**.
2. Ergänzen Sie die Methode **`moveSnake`**.
3. Ändern Sie die Main-Methode so ab, dass im Rahmen der do...while-Schleife die neu erstellten Methoden aufgerufen werden. Orientieren Sie sich hierbei am anfangs dargestellten Struktogramm.

© Uwe Homm Version vom 31. Mai 2020

Aufräumarbeiten

Leider arbeitet das Programm zu diesem Zeitpunkt nicht ganz korrekt! Wenn Sie die Aufgaben zuvor richtig erledigt haben, bewegt sich die Schlange zwar in der gewünschten Richtung, aber das Ende der Schlange bleibt nach jedem Zeitschritt (Durchlaufen der **`do...while`**-Schleife in der **`Main`**) stehen. Es scheint, als würde die Schlange wachsen, was aber nicht der Fall ist.

Es wird lediglich im Array **`gameArea`** das bisherige Ende des Schlangenkörpers in Form des Zeichen **`snakeBodyCharacter`** nicht gelöscht, so dass dieses weiterhin enthalten ist. Statt dessen muss dort nun das Zeichen **`gameBackgroundCharacter`** gespeichert werden!

Für die Lösung dieses (und eines noch zu besprechenden) Problems, gibt es mehrere Lösungsansätze. Einer wäre, dass man im Rahmen der Neupositionierung der Körperelemente an der Position des „alten“ Schlangenendes direkt im Array **`gameArea`** diese Änderung durchführt. Das würde aber bedeuten, dass die Methode zur Bewegung der Schlange auch in einer anderen Datenstruktur (nämlich **`gameArea`**), die an dieser Stelle eigentlich nicht verändert werden sollte, Änderungen vornimmt. Dabei werden eigentlich voneinander abgegrenzte Funktionalitäten wieder vermischt. Das ist nicht wirklich sinnvoll!

Daher habe ich mich dafür entschieden, das Array **`gameArea`**, welches ja die jeweils aktuelle Spielsituation für die anschließende Ausgabe per Methode **`drawScreen`** enthält, bis auf die Umrandung zu löschen.

Die Umrandung muss natürlich stehenbleiben, da sie sonst beim nächsten Zeitschritt nicht mehr angezeigt würde.

Für diese Aufgabe soll es eine Methode mit dem Bezeichner **`clearGameArea`** geben.

Diese Methode muss keinen Wert zurück liefern. Daher ist der Rückgabetyp wieder void.

Allerdings soll die Methode ja auf die aktuelle Spielsituation zugreifen. Somit wird diese als Parameter übergeben.

Im Kasten ist die Beispiel-Implementation dargestellt.

```
// Lösche das Spielfeld ohne Rand
static void clearGameArea(char[,] gameArea)
{
    // Array mit Hintergrundzeichen vorbelegen
    for (int row = 1; row < windowHeight - bottomBorder - 1; row++)
    {
        for (int column = 1; column < windowWidth - 2; column++)
        {
            gameArea[row, column] = ' ';  // Leerzeichen
        }
    }
}
```

Sehen Sie sich vor allem die Startwerte und Laufbedingungen der beiden geschachtelten **`for`**-Schleifen an, um die Funktion nach zu vollziehen!

Aufgaben

1. Fügen Sie die Methode **`clearGameArea`** ihrem Projekt hinzu.
2. Überlegen Sie, an welcher Stelle innerhalb der **`do...while`**-Schleife in der **`Main`** diese Methode aufgerufen werden sollte und ergänzen Sie dies dementsprechend.

© Uwe Homm Version vom 31. Mai 2020