# Практически проект: Змия



Днес ще създадем конзолната игра "[**Змия**](https://en.wikipedia.org/wiki/Snake_(video_game_genre))". Това е игра, в която потребителят **контролира фигура с формата на змия**, която се движи в рамките на зададено поле, взимайки **храна** по пътя.

Text

Description automatically generated A picture containing text

Description automatically generated

## Писане на кода на играта

Нека да създадем играта "**Змия**".

### Създаване на Visual Studio проект

Първо, **стартирайте** **Visual Studio** и създайте ново C# конзолно приложение. **Важно** е да **преименувате главния клас** от **Program.cs** на нещо **по-смислено** – например **SnakeGame.cs**.

След като създадете проекта, трябва да изглежда така:

Graphical user interface, application, Word

Description automatically generated

## Създаване на GitHub репо

**Изберете** един от начините за **създаване на репо**, които сме ви показвали – от **GitHub сайта** или директно от **Visual Studio**.

|  |  |
| --- | --- |
| Icon  Description automatically generated | Важно е да изберете **ваше оригинално име** за проекта!  Вашият GitHub профил трябва да бъде **уникален** и да не бъде същият като този на вашите съученици.  Може да следвате тези инструкции за разработка на проекта, но можете и да **правите промени** и да **имплементирате проекта по различен начин** от вашите съученици. |

Graphical user interface, application

Description automatically generated with medium confidence

### Стъпки за имплементация на играта

Ще започнем, като имплементираме **два класа** и **една** **енумерация** (enum).

Първият **клас Position** ще репрезентира всеки елемент от змията. Класът **Position** ще има:

* **Конструктор**, който приема:
  + **Ред** (row) на текущия елемент
  + **Колона** (column) на текущия елемент
* **Свойства**:
  + int Row – реда на текущия елемент
  + int Row – колоната на текущия елемент

Вторият **клас** **Food** ще репрезентира **храна**, която змията **трябва да изяде**, за да **порасне** и да се **движи по-бързо**. Класът **Food** ще има:

* **Полета**:
  + **char[] foodSymbols –** съдържа различни **символи за храната**, които ще визуализираме на конзолата
* **Конструктор**, който приема:
  + **Височина** (height) на конзолния прозорец
  + **Ширина** (width) на конзолния прозорец
* **Свойства**:
  + int XCoordinate – хоризонталната (x) координата на храната
  + int YCoordinate – вертикалната (y) координата на храната
  + char Symbol – визуализация на храната чрез символ

Третият и последен **клас Snake** ще репрезентира змията, която ще се движи в рамките на конзолния прозорец. Класът **Snake** ще има:

* **Константи**:
  + **int StartingElementsCount** – началните елементи на змията
  + **int MaximumSnakeSpeed** – максималната скорост на змията
* **Конструктор**, който приема:
  + int bordersOffset – отстояние на границата (border)
  + int boardHeight – височина на полето (конзолния прозорец)
  + int boardWidth – ширина на полето
* Свойства:
  + **int** **StartingXPosition** – начална X-координата на змията
  + **int** **StartingYPosition** – начална Y-координата на змията
  + Queue<Position> Elements – съдържа **елементите на змията**
  + Position NewHeadPosition – елемент с **нова позиция** на **главата на змията**
* **Методи**:
  + Queue<Position> InitialCreate() – **създава началната змия**
  + bool IsOutOfBouders(int boardHeight, int boardWidth, int bordersOffset) – проверява дали **змията** е **извън границите** на прозореца
  + bool IsBitten() – проверява дали **змията** се е **захапала**
  + int GetReversedSpeed() – взима reverse скоростта на змията, за да визуализираме **правилното число** на конзолата.

**Енумерацията** Direction съдържа константи за **всяка посока**, в която **змията** може да се **движи**:

* **Right**
* **Left**
* **Down**
* **Up**

След като сме **готови** с **класовете** и **енумерацията**, ще имплементираме **главната логика** на играта. За тази цел се нуждаем от **седемнадесет метода**, които ще използваме, за да може играта да **работи правилно**.

* void SetWindowProperties() – задава **ширината** и **височината** на **конзолния прозорец**
* void DrawBorders() – създава **границите** и ги чертае на конзолата
* void DrawNewFood() – **премахва** **старата храна** от конзолата и визуализира **нова**
* void GenerateNewFood() – създава нова инстанция на **храна** със **случайни** **символ** и **координати**
* bool FoodIsInSnake() – проверява дали координатите на **генерираната храна** съвпадат с някой от **елементите на змията**
* void PlayGame() – **стартира играта**
* void DrawSnake() – визуализира **всеки елемент от змията** на конзолата
* void ChangeSnakeDirection() – **сменя посоката на движение** на змията в зависимост от **натиснатия** **клавиш** на клавиатурата
* void GenerateSnakeNewHead() – генерира **нова глава** за **змията** с **нови координати**, в зависимост от **натиснатия клавиш**
* string GetEndGameMessage() – връща съобщение за **край на играта**, когато змията се е **захапала** или е излязла **извън границите**
* void WriteEndGameMessage(string outcomeMessage) – показва **съобщение** в **панела с резултати**
* bool SnakeEatsFood() – проверява дали **храната** и **главата на змията** са на **едни и същи координати**
* void ModifyPlayerAndSnake() – **добавя точки** към резултата на потребителя и **увеличава скоростта на змията** с **1**
* void MoveSnake() – извършва **движението на змията** чрез **премахване** на **първия елемент** от опашката
* void DrawSnakeHead – показва **генерираната глава на змията** на конзолата
* string GetSnakeHeadElementSymbol() – определя **символа** за **глава на змията** в зависимост от **посоката**
* void DrawScorePanel() – чертае **панел с резултати** на конзолата

Това е **всичко**, от което се нуждаем, за да може **играта да работи**. Нека да продължим с **имплементацията**.

## Define the Classes

### Class Position

Let's create the first **class** and **name** it Position. You should already know how to do that:

A screenshot of a computer

Description automatically generated

The **first** **thing** in the **class** is to **create** the **properties** that we will **need**. We need **two** **properties** for the **row** and **column**. They will be of **type** **int** and will **help** us to **keep** **track** of **each** **element** of the **snake**.

A picture containing text

Description automatically generated

We are **done** with the **properties**. The **next** **thing** is the constructor. Create the **constructor** and **make** it **accept** a **row** and **column**, which we will **pass** to the **class** when we **create** an **instance** of it.

Text

Description automatically generated with medium confidence

Finally, **assign** the **row** and **column** **submitted** in the **constructor** to our **Row** and **Col** **properties**. It should look like this:

Application

Description automatically generated with low confidence

Your **class** should look **like** this:

Картина, която съдържа текст

Описанието е генерирано автоматично

This is all we need for this **class**. The **next** **step** is to **implement** the Food **class**.

### Class Food

Create a **new** **class** and **name** it **Food**. You should already know how to do this:

Text

Description automatically generated with low confidence

In the class first, we will need **three** **properties**. The **first** **two** will be of **type** **int** and will **represent** the X and Y **coordinates** of the **food**.

Text

Description automatically generated with low confidence

The **third** **property** will be of type **char** and will be the **representation** of the **food**.



Now, before we **create** the **constructor**, we will need a **private field**. It will be of **type** char[] and will **contain** all the **different** **food symbols** (what the food will look like when we display it on the console).



**\*Note** - You can **copy** these or **create** your **own** so your **project** is **different** from the other **students**.

|  |
| --- |
| '#', '@', '%', '\*', '&' |

We have all we need. What is **left** is to **create** the **constructor** and our **class** is **done**.

Create the **constructor** and **let** it **accept** the **int windowHeight** **and windowWidth**.

A picture containing text

Description automatically generated

In the **constructor** first, we need a **variable** of **type** **random**. The **variable** will help us to **generate** **random** **coordinates** and a **symbol**, for the **food** **whenever** we **create** a new **one**.



The next **step** is to **assign** a **random** **number** to our **coordinate** **properties**. We will do this by **using** our **random** **variable**.



Here we **start** from **two** **otherwise**, the **food** has a **chance** to be **generated** on the **borders** (the snake won't be able to reach it).

Finally, we should **generate** a **random** **symbol** for the **food**. Try to do this by yourself.



This is all we need for our **class Food**:

Картина, която съдържа текст

Описанието е генерирано автоматично

Be **careful** with the **order** of properties, fields, constructors, etc. **in the class** to be **correct**!

Let's **continue** with the **next one**.

### Class Snake

Create a **new** **class** and **name** it **Snake**.

Graphical user interface

Description automatically generated with low confidence

First, **two** **constants** int **should** be **created** for the **number** of **initial** **elements** of the **snake** and the **maximum** **speed** of the **snake**.



The **next** **thing** that we are **going** to **need** is **five** **properties**. The **first** **two** are **for** the **snake** **starting** **X** and **Y** **positions**.

Text

Description automatically generated

The next **property** is for the **speed** of the **snake**.

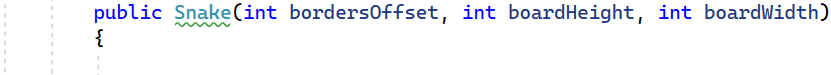


The **last** **two** **properties** are for the **snake** **elements** and the **snake's new head**. The **property** for the **elements** will be of **type** Queue<Position>, and the **property** for the **snake** **head** will be of type Position(the Position is a class we created earlier).

Text

Description automatically generated with medium confidence

The **next** **step** is to **set** the **properties** in the **constructor**. **Create** the **constructor** and **let** it **accept** the **borders** **offset**, **border** **height** and **width**.



In the **constructor**, we need a **variable** of type Random. The **random** **variable** will help us set the **starting** **positions** of the **snake** to **random** **coordinates** each time a **new** **game** is **started**.



The **X** **position** will be a **random** **number** **between** the **borders** **offset** and **border** **height** minus the **borders** **offset**. Here we **subtract** the **borders** **offset** to **prevent** the **snake** from **spawning** on the **borders**.



The **Y** **position** will be a **random** **number** **between** the **borders** **offset** and **border** **width** **divided** by **two**. Here we **divided** by **two** to **prevent** the **snake** from **spawning** **near** the **left** **border**.



Next is to **set** the **speed**, which will be **equal** to our **constant** for the **snake's maximum speed**.



The **final** **thing** is to **create** the **snake** **elements**. We will **create** the **snake** **elements** by **calling** the **method** InitialCreate(). The **method** is not **created** **yet** so **assign** it to the Elements **property** and **comment** **it** **on**.

A close up of a logo

Description automatically generated

Now the **constructor** is **ready** and **everything** is **set** to the **correct** **values**.

The **next** **step** is to **create** the **methods** for the **snake**.

#### Implement the InitialCreate() Method

The **method** will be of type Queue<Position> and will **create** and **equeue** each **element** of the **snake** to a **variable**. **Finally**, it will **return** the **created** **elements**, so we can **set** **them** to our Elements **property** in the **constructor**.

**Create** the **method** and **make** it **private** **because** we are **going** to use it only **inside** the **snake** **class**.

Text

Description automatically generated

The **first** **thing** in the **method** is to **check** that the **starting** **position** **Y** is not an **even** **number** and if it is **not**, we **need** to **add** **one** to it to **make** it **even**. We **should** do that **otherwise** the **snake** won't be **able** to **reach** the **left** **border**(the snake will die before it reaches the border).

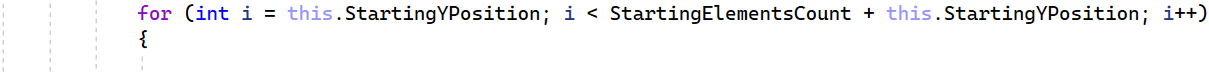
Text

Description automatically generated

Next is to **create** a **variable** **which** will **hold** the **newly** **created** **elements**.



Now we **need** a **for loop** which will **start** from the **Y starting position**, and **iterate** **until** is **less** than the **starting** **elements** **count** plus the **Y starting position**.



In the loop, we just have to **enqueue** a new **position** with **starting** **positions** **X** and i as **parameters**.



Finally, **return** the **variable**.



This is all we **need** for this **method**. Now go **back** to the **constructor** and **uncomment** this **part**.

Text

Description automatically generated

Let's **continue** with the **next** **method**.

#### Implement the IsOutOfBorders(int boardHeight, int boardWidth, int bordersOffset) Method

The **method** will be **bool** and will **check** if the **newly** **generated** **snake** **head** is out of **bounds**. For this **method**, we will **use** a **lambda** **expression**.



Now we **should** **check** if the **snake's** **new** **head** **position** – **row** **property** is **less** than **one**, or the **snake's new head position** – **row** **property** is **bigger** than the **board** **height** **minus** **borders** **offset**.



This way we have **checked** the **top** and **bottom** **boundaries**. think of a way to **check** the **left** and **right** **boundaries**.



This is all we need for this method. Let's continue with the next one.

#### Implement the IsBitten() Method

The **method** will be of **type** bool and will **check** if the **snake's new head** is at the **same** **coordinates** as **any** of the **elements** in the **queue**. We will again use **lambda** **expression** **here** but this time **combined** with LINQ.



Now we **should** **check** if any of the **elements** in the **queue** has a **column** and **row** the **same** as the **snake's new** **head**. You **should** **know** how to do **this** by **yourself**.



This is all we need for this method. Let's create the last one for this class.

#### Implement the GetReversedSpeed() Method

The last **method** for this **class** will **return** int, the **reverse** **speed** of the **snake**. We **want** to **reverse** the **speed** so we can **show** the **user** the **correct** **speed**.

**Create** the **method** and **think** of a way to **return** the **correct** **speed**. **Remember** we have a **constant** for the **maximum** **speed** – work with it.



This is all we need for this **method** and also for the **class**. The whole Snake **class** looks like this:

Картина, която съдържа текст

Описанието е генерирано автоматично

Let's **continue** with the **enumeration**.

### Enumeration Direction

The **enum** will **have** each **direction** the **snake** can **move** with **corresponding** **indexes**. **Create** the **enumeration** by yourself, you should know how to do it.

A computer screen shot of a code

Description automatically generated

What is **left** is to **create** **constants** for all **possible** **directions** (right, left, down, up) and **assign** **indexes** to **them**.

A picture containing table

Description automatically generated

This is all we need. With this **part** **finished**, we are **done** with the **preparation** of the **classes** and the **enumeration**. Let's **continue** with the **initialization** of the **game**.

## Initialize the Game

The **first** **thing** is to go to our **startup** class **where** we will **initialize** the **game**. In our case, it is named SnakeGame:

A screenshot of a computer

Description automatically generated

The first **thing** we should do is to set the **output** **encoding** to **Unicode** because we're using **Unicode** **characters** for the **snake**.



Next, we need four **constants** that we will use to **set** up the **console** **window** as well as **initialize** the **snake** and the **food**.

Text

Description automatically generated

After the **constants**, we will **use** our **first** **method** SetWindowProperties(). The **method** is not **implemented** yet so **type** its **name** and **comment** it on.



The **next** **thing** we need is to **create** an **array** of the **type** of our Position **class**. This **array** will **contain** all the **possible** **directions** the **snake** can **move**.

Text

Description automatically generated

**After** we have all **possible** **directions** we **need** two int **variables**. The **first** one will **hold** the player's **points**, and the **second** one will **hold** the **starting** **direction** of the **snake** (we will use our enumeration for this).

A black and blue letters and symbols

Description automatically generated with medium confidence

The **next** **thing** is to call the **method** DrawBorders(). The **method** will **create** and **display** the **borders** on the **console**, it is not **implemented** yet so **type** its **name** and **comment** it on.



Now we have to **initialize** the **food**. The **food** will be **initially** **null** **because** we will **create** a **new** **instance** for it in the **method** DrawNewFood().



After the **food**, we **should** **initialize** the **snake**. Create a new **instance** for the **snake** and pass the **constants** BorderOffset, BorderHeight and BorderWidth.



Finally, we should call the method DrawNewFood() and PlayGame(). The **methods** are not **implemented** yet so **type** their **names** and **comment** **them** **on**.

A picture containing text

Description automatically generated

This is all we **need** for our **game**, now we just have to **implement** the **methods** so the **game** can **work** **properly**. Let's **continue** with them.

### Implement the Methods

#### Implement the SetWindowProperties() Method

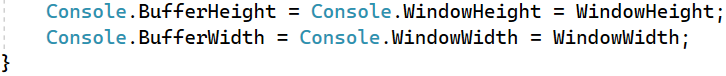
This **method** will **set** the **properties** we will **need** for our **console**. The **width** and **height** of the **console**, also the **visibility** of the **cursor** will be **set** to false. The **method** will **make** **adjustments** **before** the **game** **starts**.

Create the **method**, and **set** the **cursor** **visibility** to false so that it is not visible during the game:

A picture containing text

Description automatically generated

Now we **should** set the **console** **buffer** and **window** **size**. We **will** do **this** by **accessing** the **console** **window** and **console** **buffer** **width** and **height** **properties** and **setting** them to our **constants** for **width** and **height**. We can do it by **using** this **syntax**, or you **can** **set** **each** **property** by **itself**.



Now we can **uncomment** the **method** and **test** if it **works**.

🡪

Text

Description automatically generated

After we run the **program** the **console should shrink** to the given **size values from the constants**.

This is all we **need** for this **method** let's **continue** with the **next one**.

#### Implement the DrawBorders() Method

This **method** will **draw** the **vertical** and the **horizontal** borders of the **playfield**. This will **display** our borders and **help** us **see the walls**.

**Create** the **method** and use **StringBuilder** to **append the borders**:

Картина, която съдържа текст

Описанието е генерирано автоматично

StringBuilder class can be **used** when you want to **modify** **a string** without creating a **new** **object**. For **example**, using the StringBuilder class can **boost** **performance** when **concatenating** many **strings** together in a loop.

**Note**: If you **want to learn** more about StringBuilder, click here: <https://learn.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=net-7.0>.

The **next step** is to **choose** **a color** for our **borders**. We can do that by using the **Console.ForegroundColor** property and **choosing** the **color** we want to be **displayed**. In this case we **set** the **border’s color** to **cyan.**



After that, we need to **set** our **horizontal** and **vertical** border pieces, which will **later on** be **appended** in our **StringBuilder**.



The **next** thing we **need to do** is **set** the **horizontal side’s length**. We should **make a new string** with the piece we’ve made. For the count, we need to **subtract** the **border’s offset** from our **board’s width** and then we **add 2** more to make it **equal**.



After the **horizontal side’s length** we **need** to **set** the **empty horizontal side**. We can **make that** by **creating** **another string** with a **whitespace char** and a **count** of the **border’s offset substracted** from the **board’s width**.



What we **need** to do **next** is to **append the pieces** to the **StringBuilder**. This will **build** our **horizontal line** of the game.



For our **vertical line** we need to **make a** for-loop that will **create** our **rows**. It should **stop** before the **row** becomes **equal to** the **height** of the **board subtracted** bythe **border’s offset**.



**Inside** of the loop, we need to **append** the pieces that will **build** our **vertical line**:



After that we **need** to **create** our **down horizontal line**, the **same** **way** as we **made** the **upper** one. Do this by yourself:



**Finally** we **should** **make a new string** with our StringBuilder **converted** to a string and **trimmed in the end**.

Картина, която съдържа текст

Описанието е генерирано автоматично

#### Implement the DrawScorePanel() Method

The **method** will **store** the **player’s points** and also the **speed** of the **snake** while the player is playing.

**Create** the **method** and yourStringBuilder. In the StringBuilder, we will **append** our **points** and the **speed** of our **snake**.

A close-up of a logo

Description automatically generated

Now, we need to get our **snake’s speed**. We will do this by **using** **our** **method** made in the **snake** **class**: GetReversedSpeed(). Store the speed in an integer.



After getting our **speed**, we need to **append** our **points** and the **speed** of the **snake** in our StringBuilder. Also, we need to **add** some text before them so it can make **sense**:



After that, we **should** **make a new string** with our StringBuilder **converted** to a string and **trimmed in the end**, we did this in the last **method**, but try to do this by **yourself**.



Finally, **set the position** of the **text** and **draw the score panel**:



Let's **run** the **application** and see what the **results** are. If everything **works** **correctly**, after **running** the **application** the bottom part of the **console** **should** look like this:



This is how the **game** should **look** like once it’s **started**:

A screen shot of a computer

Description automatically generated

And this is what it should **look** like when you **crash** in the **wall**: Картина, която съдържа текст

Описанието е генерирано автоматично

## Upload Your Project to GitHub

**Upload** your **project** changes to GitHub, this time it's **up** **to** **you**. **Upload** it with **TortoiseGit**, **Git** **Bash** or **Visual** **Studio**. Choose the **best** **way** for **you**.

A screenshot of a computer

Description automatically generated

## \* Modify the Code, Write Your Own Features

Now, it’s time to **play with the code** and **modify it**.

|  |  |
| --- | --- |
| Icon  Description automatically generated | This is your own project. **Be unique**. Don’t be a copy/paster!   * Implement your **own features**. * **Implement the code yourself**, using your own coding style, code formatting, comments, etc. * Make the project **more interesting**. Learn by playing with the code and adding your own changes. |

Below are a few **ideas** of what you can implement or modify as an addition to your code.

### Add Unicode Characters

Search for a **more** **interesting** **character** to **represent** the **snake and the food**.

**Unicode characters table** – <https://www.rapidtables.com/code/text/unicode-characters.html>.

### Snake Getting Through Walls

**Make** the game **easier** for the **player** and let the **snake** get trough **walls**. For example, if the **player reaches the right border** of the **playfield**, the **snake** will continue moving from the **left border** and the player would **not lose the game**.

### Create GUI App

You can create a **GUI app** with **WinForms** for the game.

### Additional Ideas

* **Create** an **improvised** **database**, so you can **store** the **user's** **name** and **score** and **display** them on the **console**.
  + Use JSON – <https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/how-to?pivots=dotnet-6-0>.
  + Use XML – <https://learn.microsoft.com/en-us/dotnet/standard/serialization/how-to-serialize-an-object>.
* You can **add** anything else to your **code**, based on your own **ideas**.

### Commit to GitHub

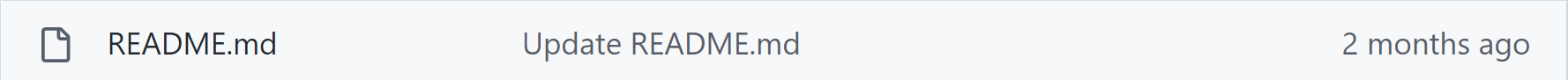
Now **commit and push your code changes** to your GitHub repo!

A picture containing chart

Description automatically generated

## Write a README.md File

It's highly recommended to provide **documentation as part of your project on GitHub** to describe what the project does. So, let's make one for this **project**. Let's start by editing the README.md file from our repo on GitHub:



### Documentation Sections

Add **information** about your project in your README.md file using **Markdown**: project goals, technologies used, screenshots, live demo, etc. Typically, you should have the following **sections**:

* **Project title** (should answer the question "What’s inside this project)
* **Project goals** (what problem we solve, e. g. we implement a certain game)
* **Solution** (should describe how we solve the problem 🡪 algorithms, technologies, libraries, frameworks, tools, etc.)
* **Source code link** (give a direct link to your source code)
* **Screenshots** (add screenshots from your project in different scenarios of its usage)
* **Live demo** (add a one-click live demo of your code)

### Project Goals

Start your documentation by describing your **project goals**. What problem does your project solve?

### Sample Documentation

This is an **example** of how you can document your project. **Don’t copy-paste:**

A green snake with a red tongue

Description automatically generated

|  |  |
| --- | --- |
| Icon  Description automatically generated | **Write the project documentation yourself**. Don’t copy/paste it!  This is your **unique GitHub profile** and your own unique project. **Be different** from others. |

Find an **appropriate** **image** and add it. You can add **images** as follows:



### Controls

Add information about the **controls of the game**:

A screenshot of a computer

Description automatically generated

### Your Solution

Describe how you **solve the problem**: algorithms, technologies, libraries, frameworks, tools, etc.

### Screenshots

Add **screenshots** of your project:

1. **Take a screenshot** with your favourite tool (e.g. the [Snipping Tool](https://support.microsoft.com/en-us/windows/open-snipping-tool-and-take-a-screenshot-a35ac9ff-4a58-24c9-3253-f12bac9f9d44) in Windows).
2. **Paste** the screenshot in the GitHub Markdown editor, using [Ctrl+V]:

Example screenshots for the "Snake" game:

Картина, която съдържа текст, монитор, екранна снимка

Описанието е генерирано автоматично

## Upload Your App to Replit

Let’s add our **project** to **Replit** so we can share it with our **friends** and add it to our **GitHub** profile. You already should know how to do that.

Note: use Mono C# template in Replit for **deploying your "Snake" game**, as with C# it will be **too slow** for you to play normally. You should have **all your classes in a single file**, so that they can **work together**.

## \* Running the Project in a Blazor Web App

To make the **Snake game** run in a **Blazor web app** you will need to use the **Snake-Blazor-Skeleton** found in the **resources** for the exercise. The skeleton is a simple **Blazor app**:

A screenshot of a computer

Description automatically generated

Inside you will find a new GameSnake.cs file:

A picture containing text, screenshot, font, line

Description automatically generated

Here we have an **asynchronous** method to run the code. It uses an already implemented BlazorConsole.

Inside the Run() **method** is where you will place the whole content of your original GameSnake class. Don't forget to also **move all other classes** that we made.

To **make the app run** you will need to do a **few adjustments**:

* First you will need to make **everything asynchronous**, every **method** and every **method** **call** (within this class)**.**
  + An async method runs **synchronously** until it reaches its first await **expression**, at which point the method is **suspended** until the **awaited task** is complete. That is how our Blazor app works.
  + To make a **method async** you add the async keyword. If a method is **void** you replace it with Task. If it returns something you make it a Task of that something:

A picture containing text, screenshot, font, line

Description automatically generated

A picture containing text, screenshot, font, line

Description automatically generated

* Second, in the PlayGame() **method** remove the DrawSnakeHead() call. After that remove the GenerateSnakeNewHead() call and move it at the beginning of the DrawSnakeHead() **method**:

A picture containing text, font, line, screenshot

Description automatically generated

* Third, in the DrawSnake() **method** we will add an await to DrawSnakeHead() at the end:

A picture containing text, screenshot, font, line

Description automatically generated

Otherwise, the **snake head** **will not be displayed**.

* Fourth in MoveSnake() we will change the WriteLine() to a Write(). We do this because the snake pushes the border out of bounds if we don't as it prints a whole line:

A picture containing text, font, line, screenshot

Description automatically generated

* Fifth change the **ASCII symbols** in GetSnakeHeadElementSymbol() to the following: ˃ ˂ ˄ ˅:

A screenshot of a computer code

Description automatically generated with medium confidence

A screenshot of a computer code

Description automatically generated with medium confidence

Those symbols **look better** in the Web app, and it again doesn't push the border.

Finally in PlayGame() remove the break statement and replace it with an else statement. This is because of our await, the game over message **doesn't display in time** because of the break:

A picture containing text, screenshot, font, number

Description automatically generated

Now you can build and run the app. It will open in your browser like so:

A screenshot of a computer

Description automatically generated

### Running the Project in Github Pages

We will now run the snake game in GitHubPages. To start **commit** the new Blazor **project** on github.

Now you need to add a **new workflow** for **github actions**.

GitHubActions is a **powerful automation tool** provided by GitHub. It enables developers to **define** and **automate** **workflows** for their software projects directly within the GitHub repository. You can easily set up a variety of **continuous integration** and **continuous deployment** (CI/CD) processes, **code linting**, **testing**, and **other tasks** to streamline your software development workflow.

**Workflows** are the **central building blocks** of GitHub Actions. They are defined using **YAML files** and consist of one or more **jobs**. Workflows can be triggered by various events, such as **pushing code** or creating a **pull request**.

Before you run any workflow make sure you have **write permissions** for actions:

A picture containing text, font, screenshot, line

Description automatically generated 🡪

A screenshot of a computer

Description automatically generated with medium confidence 🡪A screenshot of a computer

Description automatically generated with low confidence

Don't forget to **save changes**.

Then on your GitHub repository you can see the [Actions] button:

A picture containing text, font, screenshot, line

Description automatically generated

There aren't **any workflows yet**. Let’s change that.

To do that click on the [setupaworkflowyourself] button:

A picture containing text, screenshot, font, line

Description automatically generated

Use the following **yaml file:**

|  |
| --- |
| main.yml |
| # First we set the name of the workflow as "Deployment".  name: Deployment  # Specifies the trigger for the workflow, which is a push event to the main branch.  on:  push:  branches:  - main  # Defines a single job named "deployment" that runs on the "ubuntu-latest" runner. The runner represents the environment in which the job will be executed.  jobs:  deployment:  runs-on: ubuntu-latest  # This step uses the "actions/checkout" action, which fetches the repository's code to the runner, allowing subsequent steps to operate on it.  steps:  - uses: actions/checkout@v2  # This step sets up the .NET environment using the "actions/setup-dotnet" action. It specifies the desired .NET version as 7.0.x.  - name: Setup .NET  uses: actions/setup-dotnet@v1  with:  dotnet-version: 7.0.x  # This step runs the command dotnet publish -c Release -o release --nologo, which builds the code in the "Release" configuration and publishes it to the "release" folder.  - name: Publish  run: dotnet publish -c Release -o release  # This step uses the sed command to replace the base href tag in the index.html file. It changes <base href="/" /> to <base href="/<YOUR-REPOSITORY-NAME>/" />.  - name: Setup index.html base tag  run: sed -i 's/<base href="\/" \/>/<base href="\/<YOUR-REPOSITORY-NAME>\/" \/>/g' release/wwwroot/index.html  # This step copies the index.html file from release/wwwroot/ to release/wwwroot/404.html.  - name: Copy index.html to 404.html  run: cp release/wwwroot/index.html release/wwwroot/404.html  # This step creates an empty .nojekyll file in the release/wwwroot/folder. This file is used to disable Jekyll processing on GitHub Pages, ensuring that the files and directories starting with underscores (\_) are not treated as special.  - name: Create .nojekyll  run: touch release/wwwroot/.nojekyll  # This step uses the JamesIves/github-pages-deploy-action action to deploy the release/wwwroot/ folder to the gh-pages branch of the repository. It requires the GITHUB\_TOKEN secret for authentication and specifies the folder to be deployed.  - name: Commit wwwroot to GitHub Pages  uses: JamesIves/github-pages-deploy-action@3.7.1  with:  GITHUB\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}  BRANCH: gh-pages  FOLDER: release/wwwroot |

This workflow will **publish** our app for github pages with the **appropriate settings** that are needed.

After this click the [Actions] **tab** again to see the **workflow running**:

A white rectangular object with blue text

Description automatically generated

After all jobs have finished executing you should get a **green checkmark**:

A picture containing text, screenshot, font, line

Description automatically generated

Finally we need to deploy to pages. Go to your **repository settings** and select [Pages]:

A screenshot of a computer

Description automatically generated

Select our new gh-pages branch and click [Save]:

A screenshot of a computer

Description automatically generated with medium confidence

After a **few minutes** your app will be up:

A screenshot of a computer

Description automatically generated with medium confidence 🡪 A screenshot of a computer

Description automatically generated with medium confidence

This is how you can **run your** Blazor **app** in GitHub **pages**.