Phase #1

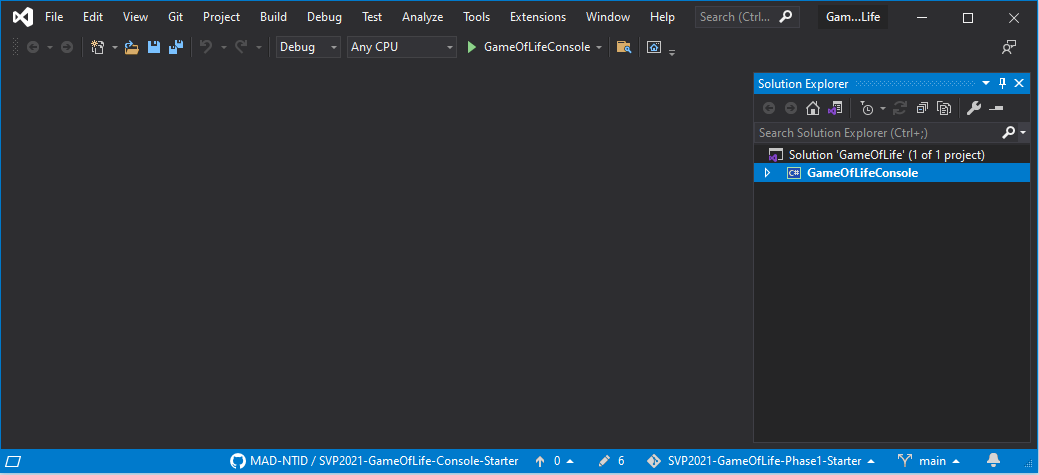
**Building a Console Application**

# Step 1 | Downloading

Download the starter code [here](https://github.com/MAD-NTID/SVP2021-GameOfLife-Phase1-Starter) to your desktop and then open the ***SVP2021-GameOfLife-Phase1-Start*** folder. Once opened, double-click on the ***GameOfLife.sln***file. This will open Visual Studio 2019 with our project.

# Step 2 | Code Editor Exploration

After letter Visual Studio 2019 (*or* VS 2019 for short) loads, you should be greeted with something similar as the following:



This is the code editor we will be using today for each phase.

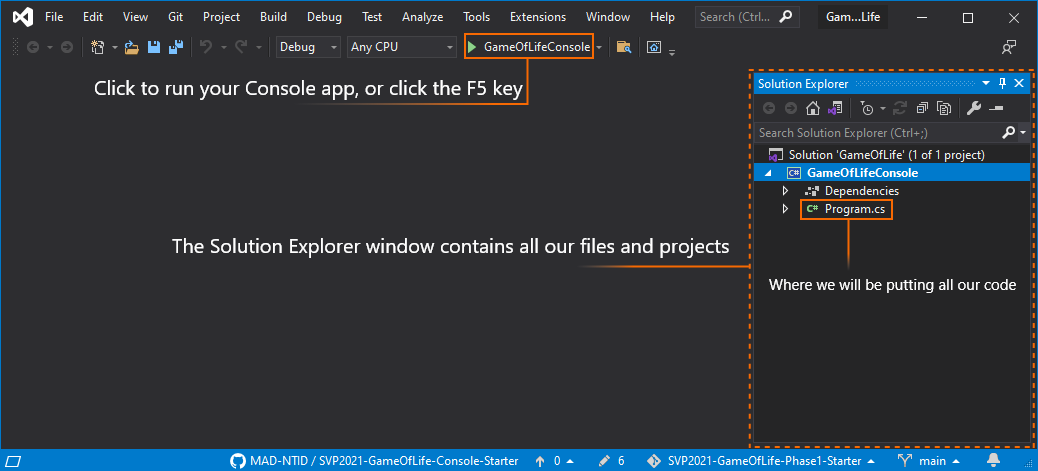
A *solution* is a collection of files that, when combined, create an executable program. When you double-clicked on the solution *(.sln)* file, you loaded the solution and all its dependencies into Visual Studio.

Open the solution by clicking the arrow button to the left of the *GameOfLifeConsole* text in the *Solution Explorer* window. (Refer to the image below)

Graphical user interface, text, application, email

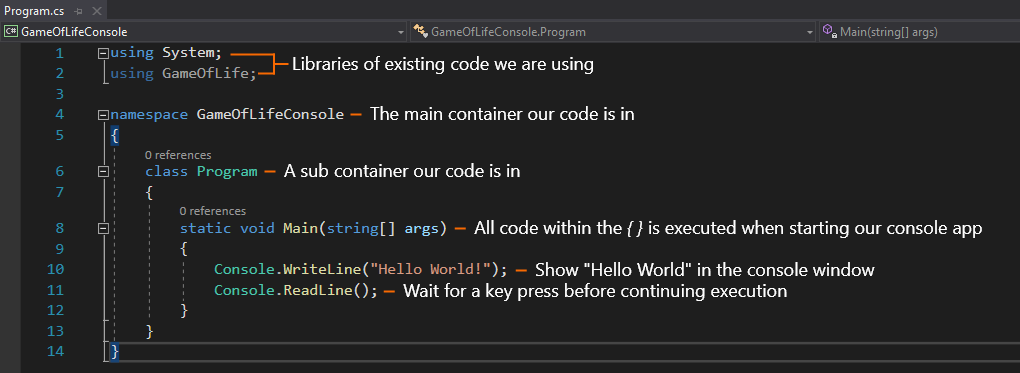
Description automatically generated

With our project’s contents now visible, let’s explore the code editor.



Try your best to remember these parts of the editor. We will be referring to them frequently.

***Program.cs*** is a file that is part of the solution. It contains the actual code we will write to create the simulation. Let’s look at the code itself. Double-click on ***Program.cs*** to open it. The contents of the file will display in the main editing window.



You probably noticed that the code is displayed in a number of different colors. This is done automatically by Visual Studio. The colors help you identify different parts of the code quickly. You’ll learn more about this in your first programming course.

You should also notice that the illustration shows lines of white text that isn’t displayed on your screen. Those are comments we added to help you understand what each line of code represents.

# Step 3 | Testing Existing Code

When you create a new solution in Visual Studio, it automatically includes the instructions for a very simple test program. In this step, we will run *(execute)* that code to make sure everything runs successfully before we start making modifications. This is an important step that applies to any code your write in the future.

To run the app, go ahead and click the **start** **button** in the button bar at the top of the screen. (Take a look at the full-screen image in *Step 2*.) The result should look like the following:

Text

Description automatically generated with medium confidence

**It Worked!**

As you can see, *“Hello World!”* was *written* to the console window.

|  |  |  |
| --- | --- | --- |
| 🡺 | Look at the code in the Visual Studio Editor. What is the line number for the line of code that caused the text to be displayed? |  |

Now press any key *(the spacebar for example)* to continue. Depending on the editor’s settings, the window may have closed or is now showing some odd information below the “Hello World” text. If the window hasn’t closed, go head, and press the *“X”* button in the top right to close it *(the console window, not VS 2019)*.

You may have noticed your editor changed its appearance when the application started; this is because the program runs in *debug* mode. Debug mode provides us with tools to examine our app as it runs *(typically to discover and fix issues)*. You’ll learn more about debugging programs in your Programming Foundations classes.

At this point, everything seems to be working correctly. Let’s go ahead and start creating our program.

# Step 4 | Pre-Game Preparation

In this step you will add the first few lines of code that are essential for everything else. Before moving on go ahead and remove the *Console.WriteLine* that prints *“Hello World”*. Leave the *Console.ReadLine* and put everything your write before it.

* Create a game variable (4.1)
* Change the console window’s title (4.2)
* Welcome users to the game (4.3)

### 4.1 | Create a Game Variable

|  |  |
| --- | --- |
| First, we want to add a *variable* that we will use to access the game. What did you type to create a variable named *game*? | |
| 🡺 |  |

Using the above variable (***game***)*,* we will be able to access our game and tell it what to do.

### 4.2 | Change the Console Window’s Title

|  |  |
| --- | --- |
| Normally, the title bar on the console window shows the name of the executable file and the full path to that file. It’s pretty long and ugly: | |
|  | |
| It would be nicer if it looked like this: | |
|  | |
| What command did you add to the program to cause this to happen? | |
| 🡺 |  |

### 4.3 | Welcome Users to The Game

Now it is time to welcome the user *(you)* to the game! We are going to change the color of the text in the console window to yellow and then yell “WELCOME TO THE GAME OF LIFE” in the console window. Therefore, what two lines of code did the instructor show that would provide the desired functionality?

|  |  |  |
| --- | --- | --- |
| 🡺 | What command did you type to change the display color of the text to yellow?? |  |
|  |  |  |
| 🡺 | What command did you type to display “WELCOME TO THE GAME OF LIFE”? |  |

Now, save your work and run your program!!!!! If you’ve done everything correctly, it will look like the image below.

Text

Description automatically generated

If your program didn’t run (or didn’t give you the result you expected), go back and check your typing very carefully. C# is (what we call) a case-sensitive language. Because of that,

Console.Writeline  
and  
console.writeline  
and  
Console.WriteLine

Are seen as three ***different*** commands. Make sure that your capitalization matches that of the instructor. Also, make sure you added the semi-colon character ( ; ) at the end of each line.

# Step 5 | Getting User Information

In this step you will get information from the user *(you)* entered via the console window that tells the game what to do. To do this, we usually display a *prompt* that tells the user what to enter for each item. A *prompt* is just a simple string.

For this application, the user can control three factors:

1. Prompt for the number of rows in the simulation
2. Capture the value entered by the user
3. Prompt for the number of columns in the simulation
4. Capture the value entered by the user
5. How often the simulation should update (in seconds)
6. Capture the value entered by the user

|  |  |
| --- | --- |
| **An important note** | Normally, when capturing a number entered by the user, we would *validate* the entry to make sure it is reasonable.   * The value is *really a number* (not text like “eighteen”) * The value is a number in an appropriate range (for example, rows and columns must be greater than 5) * The value is a *whole* number. (You can’t have 5.375 rows)   To keep things simple, we will pretend that all users enter data perfectly and never, ever, make a mistake. |

In this step we will be writing code to:

* Show options prompt (5.1)
* Prompt & capture the number of rows (5.2)
* Prompt & capture the number of columns (5.3)
* Prompt & capture the cycle time in seconds (5.4)

### 5.1 | Show Options Prompt (title)

First, we want to show the user a message that lets them know there will be some options to come.

|  |  |
| --- | --- |
| What code did you enter to show let the user know of the coming options? | |
| 🡺 |  |

### 5.2 | Prompt the User and Capture the Number of Rows

Next, we need to know how many rows the user wants. Therefore, we are going to need to provide a prompt explaining this. After prompting the user, you will need to wait until they provide the number of rows (1-30). Once they provide the number of rows, you will store it in the *game variable*. What three lines of code did the instructor provide to complete the previously mentioned?

|  |  |
| --- | --- |
| **Tip** | All prompts should be cyan, and all user input should be magenta. |

|  |  |  |
| --- | --- | --- |
| 🡺 | What command did you type to change the display color of the text to cyan? |  |
|  |  |  |
| 🡺 | What command did you enter to prompt the user for the number of rows? |  |
|  |  |  |
| 🡺 | What command did you type to change the display color to magenta? |  |
|  |  |  |
| 🡺 | What command did you enter to capture the user’s entry? |  |

Now before moving onto the next step go ahead and test your program by presenting the value of *game.Rows* in the console window after you get it from the user.

|  |  |
| --- | --- |
| **Tip** | Console.WriteLine can be given other things besides text, for example a number is valid. |

### 5.3 | Prompt the User and Capture the Number of Columns

This section and the following are very similar to the previous, therefore neither is being provided to you. First you need to change the prompt color to cyan. Then prompt the user for the number of columns (1-30). Before receiving the input, you must change the text color in the console to magenta, so their writing is magenta. Lastly store their input inside the *game.Columns* variable this time.

|  |  |
| --- | --- |
| **Tip** | Copying code you have already wrote can be useful strategy to speed up progress. |

Before moving on go ahead and check your progress by printing *game.Columns* to the console window to make sure your new code is working as expected.  
5.4 | Prompt the User and Capture the Cycle Time

Lastly, you need to get the *game.CycleTime* value from the user. This variable controls the update frequency of the simulation and is measured in **milliseconds**. As always start by updating the color to cyan and prompting them for the cycle time **in seconds**. After the prompt be sure to change the console’s text color to magenta. Lastly, when getting the user’s input as a number be sure to convert the seconds into milliseconds before storing in *game.CycleTime*.

|  |  |
| --- | --- |
| **Tip** | There are 1000 milliseconds in a second so simply convert via (*1000 \* seconds)*. |

Once again, go ahead and test your new code to make sure it is working correctly. Otherwise have you noticed that all prompts are cyan, and all user input is magenta? Maybe if you have some spare time try changing some of the colors to see what happens.

# Step 6 | Adding Update Logic

This step will provide the core logic we need for our game to work. Everything else up to this point has been input needed for this part to work.

* + Adding a procedure for updating (6.1)
  + Preparing the console window (6.2)
  + Adding a loop for rows (6.3)
  + Adding a loop for columns *“what about second loop?”* (6.4)
  + Presenting the cells in the console window (6.5)
  + Moving to a new line in the console window (6.6)
  + Updating game statistics (6.7)

*6.1* | In this section you are going to need to create a procedure that will be used by the *game variable*. Inside of this procedure is where all the update code will reside. What are the four lines of code the instructor has provided?

|  |  |  |
| --- | --- | --- |
| 🡺 | What command did you enter to name the procedure? |  |
|  |  |  |
| 🡺 | What code did you enter to mark the beginning of the procedure’s body? |  |
|  |  |  |
| 🡺 | What code did you enter to mark the end of the procedure? |  |
|  |  |  |
| 🡺 | What code did you enter to attach the procedure to the game variable? |  |

The remainder of this step will be taking place inside this procedure.

### 6.2 | Preparing the Console Window

With our empty procedure created, it is time to fill it with some logic. First you will need to clear the console window, so it is blank *(all black)*. This is because every time this procedure executes, we want it to replace the old content in the console window with new.

|  |  |  |
| --- | --- | --- |
| 🡺 | What command did you enter to clear the console window? |  |

### 6.3 | Adding a Loop for Rows

In this section you are going to add a loop that will visit each row of the game’s grid from top to bottom. Make sure you’re careful with the syntax on this one for it is quite unique. What does the loop we require look like?

|  |  |  |
| --- | --- | --- |
| 🡺 | What code did you enter to name the procedure? |  |
|  |  |  |
| 🡺 | What code did you enter to mark the beginning of the procedure’s body? |  |
|  |  |  |
| 🡺 | What code did you enter to mark the end of the procedure? |  |

### 6.4 | Adding a Loop for Columns *“what about second loop”*

Ready for another loop?!? Now it is time for you to add another loop that will visit each column just like how we visited each row in the previous section. This loop should be placed inside the existing loop for rows.

### 6.5 | Presenting the Cells in the Console Window

Now inside both loops you will be adding code to be executed for each cell. The reason behind the two loops is as follows; the first loop travels row by row, and for each row the sub-loop visits each column in that row. Hence you can visit every cell inside the grid of cells one by one top left -> bottom right.

|  |  |  |
| --- | --- | --- |
| 🡺 | What code did you enter to get a cell at a specific row & column? |  |
|  |  |  |
| 🡺 | What code did you enter to change the console’s text color based off whether the cell is alive or dead? |  |
|  |  |  |
| 🡺 | What code did you write to present the cell to the console window as text? |  |

### 6.6 | Moving to a New Line in the Console Window

This small step is responsible for making sure the cells don’t print all on the same line *(a single row)*. We want rows and columns and therefore, you need to tell the console window to move to the next row *(or line)* once we are visiting a new row after presenting the columns for that row.

|  |  |
| --- | --- |
| **Tip** | You have already been doing this so examine your existing code to find the answer! |

### 6.7 | Updating Game Statistics

For the last section of *Step 6* you will need to print the statistics of the game to the console window. The statistics text presented must be yellow and it will display the current cycle number as well as the number of alive cells. What three lines of code are responsible for displaying this information?

|  |  |  |
| --- | --- | --- |
| 🡺 | What command must you usher to change the console’s text color to yellow? |  |
|  |  |  |
| 🡺 | What code did you enter to present the *game.CycleCounter* value? |  |
|  |  |  |
| 🡺 | What code did you enter to present the *game.AliveCounter* value? |  |

# Step 7 | Starting / Stopping the Game

In this step you will be adding the final bits of logic to the program to control starting and stopping the simulation.

* + Present control information to user (7.1)
  + Hide the cursor and wait for the *enter* key to be pressed (7.2)
  + Add another procedure that runs when *ctrl + c* is pressed (7.3)
  + Add code that stops the game (7.4)
  + Add code that starts the game (7.5)

### 7.1 | Present Control Information

In this section you will present text to the user explaining how to start and stop the simulation *(on separate lines).* This text presented will need to be yellow. To start the simulation the user should press *enter* on their keyboard. Further, to stop the simulation, the user should press the key combo *ctrl + c*. What three lines of code are responsible for the previously mentioned?

|  |  |  |
| --- | --- | --- |
| 🡺 | What code did you enter to change the console window’s text to yellow? |  |
|  |  |  |
| 🡺 | What command did you enter to inform the user that pressing *ctrl + c* will stop the simulation? |  |
|  |  |  |
| 🡺 | What command did you enter to inform the user that pressing *enter* will start the simulation? |  |

### 7.2 | Hiding the Cursor and Waiting

In this section you will add code to hide the user’s cursor in the console window and then wait for user to press the *enter* key to start the simulation. What two lines of code are responsible for this?

|  |  |  |
| --- | --- | --- |
| 🡺 | What code did you enter to set the cursor’s visibility to invisible? |  |
|  |  |  |
| 🡺 | What command did you enter to make execution halt until the *enter* key is pressed? |  |

### 7.3 | Stop Simulation Procedure

Here you will add another procedure that is executed when *ctrl + c* is pressed. Unlike the previous procedure you wrote in *6.1* this one is a little different in appearance. What three lines of code will create this procedure?

|  |  |
| --- | --- |
| **Tip** | *Ctrl + C* is also known as a *Cancel Key* which can be used to stop running programs. |

|  |  |  |
| --- | --- | --- |
| 🡺 | What code did you write to attach a procedure to the *CancelKeyPress* or *ctrl + c*? |  |
|  |  |  |
| 🡺 | What code did you add to mark the beginning of your procedure? |  |
|  |  |  |
| 🡺 | What code did you enter to mark the end of your procedure? |  |

### 7.4 | Stopping the Game

In this section you will add code inside the procedure you just created. This code will stop the game and then inform the user of this in yellow text. What three lines of code must you put inside the procedure to make this happen?

|  |  |  |
| --- | --- | --- |
| 🡺 | What command did you usher to stop the game? |  |
|  |  |  |
| 🡺 | What code did you add to change the console’s text color to yellow? |  |
|  |  |  |
| 🡺 | What command did you write to present the user of stopping the game? |  |

### 7.5 | 🎉 Starting the Game 🎉

Finally, here is the last requirement before your program is completed! To finish things up you will need to tell the simulation to start and then make the program wait indefinitely. Don’t worry about understanding the 2nd line for it is out of the scope of today’s session. What two lines of code are responsible for the previous mentioned?

|  |  |  |
| --- | --- | --- |
| 🡺 | What command did you enter to start the game? |  |
|  |  |  |
| 🡺 | What code did you enter to make the program wait indefinitely? |  |

Now run your console app and you should be able to see the simulation after providing correct user input!

🎉 Congratulations you made it 🎉