# Practical Project: Hangman

Problems for practical projects for the "**Programming Fundamentals**" course from the official "Applied Programmer" curriculum.

Today we will make the console game "Hangman":

Diagram

Description automatically generated

[Hangman](https://en.wikipedia.org/wiki/Hangman_(game)) is an **old-school favourite**, a **word game** where the goal is simply to guess the **missing word**. You will be presented with a number of **blank spaces** representing the **missing letters** of the guessed word. Use your **keyboard** to type letters.

* If your **chosen letter** exists in the answer, then **all blank spaces** in the answer where that **letter appears will be revealed**.
* Be warned, every time you **type a wrong letter** you **lose a life** and the **hangman appears piece by piece**.
* **Solve** the puzzle **before the hangman dies**.

Text

Description automatically generated Text

Description automatically generated

## Create GitHub Repository

We already have a GitHub account created, so we're moving directly to creating a new repository.

Create a **new repository** from: <https://github.com/new>. Choose a **meaningful name**, e. g. "HangmanByUsername" add a **short description** and make your repo **public**:

Graphical user interface, text, application, email

Description automatically generated

|  |  |
| --- | --- |
| Icon  Description automatically generated | Please choose **your own original and unique name** for your project!  Your GitHub profile should be **unique**, not the same as your classmates.  You can follow this tutorial, but you can also **make changes** and **implement your project differently** from your classmates. |

Also, **add a** README.md file and .gitignore **for Visual Studio**, as shown below:

A screenshot of a computer

Description automatically generated

In Git projects the .gitignore **file** specifies which files from your repo are not part of the source code and should be ignored (**not uploaded in the GitHub repo**). Typically in **GitHub**, we upload in the repo **only the source code** and we don’t upload the compiled binaries and temp files.

Finally, **change the license** to "MIT" (which is the most widely used open-source license) or another license of choice, and click on the [Create] **button** to **create your repository**:

A screenshot of a computer

Description automatically generated

Now your **repository is created** and looks like this:

Graphical user interface, text, application, email

Description automatically generated

Now let's see how to **write the code** of our game.

## Write the Game's Code

Let’s create a game and play with it.

### Create a Visual Studio Project

First, we should **start Visual Studio** and **create a new C# console application**:

A screenshot of a computer

Description automatically generated

Then, **choose an appropriate name** and a **place to save the project**. You should also **check** the [Place solution and project in the same directory] box so that we do not have an additional folder for our files. Then, click on [Next]. On the next screen, choose [.NET 6 (Long-term support)] and click [Create]:

Our **project should be created** and should look like this:

Graphical user interface, text, application

Description automatically generated

Before we continue, let's change the name of our **main class** – Program.cs to something more **meaningful**. Do it like this:

Graphical user interface, application

Description automatically generated 🡪 Graphical user interface, text, application, email

Description automatically generated

Now let's start working on our code.

### Game Implementation Steps

We are going to start by **creating variables and constants** that will **hold the animations for the game**. In the **resources**, we **provide** you with a **document** that **contains all the strings** that you are going to need for **creating these animations**. We have to create:

* **Two string constants** that will hold the **win** and **loss** **animations**.
* **Two string arrays** that will hold the **wrong guesses frames** and the **death animation frames**.

Then we will create a **method** ReadWordsFromFile() that will **read all the words** we are going to use for the Hangman from the external file **words.txt** that we also **provide** you with.

After that, we will **initialize** the **game and** think of the **methods** we are goingto **need for it to work**.

Finally, we will **implement** **eight** **methods** which will **correspond** with **each other** in order for our **game to work**.

1. string GetRandomWord(string[] words) – the **method** will help us **get a random word**.
2. void DrawCurrentGameState(bool inputIsInvalid, int incorrectGuess, string guessedWord, List<char> playerUsedLetters) – the **method** will help us **draw** the **current** **state** of the **game** on the **console**.
3. void PlayGame(string word, string wordToGuess, int incorrectGuessCount, List<char> playerUsedLetters) – the **method** will run the **main logic of the game**.
4. bool CheckIfSymbolIsContained(string word, char playerLetter) – the **method** checks whether the **player's symbol** is **contained** in the **current** **word**.
5. string AddLetterToGuessWord(string word, char playerLetter, string wordToGuess) – the **method** will add the **player's letter** to the **current** **word** if the **word contains the letter**.
6. bool CheckIfPlayerWins(string wordToGuessChar) – the **method** will check if the **player** has **won the game**.
7. bool CheckIfPlayerLoses(int incorrectGuessCount) – the **method** will check if the **player** has **lost the game**.
8. void DrawDeathAnimation(string[] deathAnimation) – The **method** will help us **draw** the **death** **animation** when the **player** **loses**.

This is all we need to **finish this project**. Now let's start with the game animations.

### Create Animations for the Game

Start by opening the "**Project-Hangman-Game-Animations.docx**" **document** we **provide** you with in the resources. Now go back to your project in **VS** and create two **string constants** and name them "Win" and "Loss". Copy the **animation strings** from the **document** and paste themas **values** to our **string constants** "Win" and "Loss". Finally, they should look like this:

A picture containing text

Description automatically generated

Next, create two **string** **arrays** which will hold the game **animations**. Name the first array "wrongGuessesFrames", then copy the **animation strings** from the **document** and paste them **between** the array’s **brackets**. Finally, it should look like this:

Chart, box and whisker chart

Description automatically generated

A picture containing chart

Description automatically generated

A picture containing diagram

Description automatically generated

Name the second array "deathAnimationFrames", copy the **animation strings** from the **document** and paste them **between** the array’s **brackets**. You already know how to do that. The result is as follows:

A screenshot of a computer

Description automatically generated

**Read Words from an External File**

In the **resources**, we also provide you with a words.txt file that **contains** over **2500** **different** words. We will read all the **words** from this file (words.txt) and **insert** themintoan **array of strings** to use for our game. First, **copy the provided file** and **paste** it into your **project** **directory**. It should look like this:

Graphical user interface, application

Description automatically generated

Now go back to your **project** in **VS**. Create a **method** and name it ReadWordsFromFile(). In the **method**, we will read the **words** from the "**words.txt**" file and **return** them as an **array of strings**.

A picture containing text

Description automatically generated

In the **method** first, create a string variable. The **variable** will **hold** the **path** of our **current directory**. Use the following method to **get the current directory**:

|  |
| --- |
| Directory.GetCurrentDirectory() |

This **method** will help us access the **current** **directory** of our project e.g. "D:\My Projects\03.Project-Hangman\bin\Debug\net6.0" and **return it as a string**. It should look like this:





Next thing is to create another string **variable**. This variable will hold the path of our **project** **directory**. Use the below method to return the name of the **parent-parent-parent directory** (the one with our **words.txt** file):

|  |
| --- |
| Directory.GetParent().Parent.Parent.FullName |

This **method** will help us access the **project** **directory** that contains our **words.txt** file e.g. (**D:\My Projects\03.Project-Hangman**). It should look like this:





The next thing is to **access** the **words.txt** file **path**. To do this, **create** a **constant** that will **hold** the **name** of the **file**.



Now, **concatenate** the **project directory variable** with the **name** of the "**.txt**"(words.txt) file to **form** the **path** to the **file itself**. It should look like this:





The last new **method** we will use is to **read the text lines from the file we have**. It is the following:

|  |
| --- |
| File.ReadAllLines() |

Now create an **array of strings** that will **store** the **read text**. Each line **contains** **one** **word**, so these are the **words** we have for our **game**:



Finally, return the words array.



Now our first **method** is ready. We can call it and see if it **works properly**. Create an **array of strings** and **call** our newly created **method** for its **value**.



Now to check if everything works **properly**, let's **debug** our **program**.



After the program hits the breakpoint in the **words array** you should have **2999 words**.

Before we continue, set "Console.CursorVisible" to false because of the **drawing** we will do later on the **console.** Console.CursorVisible will hide the **white** **cursor** that **indicates** **which** **row** we are **currently** **writing on**, because we don't want it to be visible while we play the game.



You can learn more about some methods we used here:

* **Directory.GetCurrentDirectory()**:<https://docs.microsoft.com/enus/dotnet/api/system.io.directory.getcurrentdirectory?view=net-6.0>.
* **Directory.GetParent(string)**:<https://docs.microsoft.com/en-us/dotnet/api/system.io.directory.getparent?view=net-6.0>.
* File.ReadAllLines(path): <https://docs.microsoft.com/en-us/dotnet/api/system.io.file.readalllines?view=net-6.0>.

Let's **continue** with the **initialization of the game**.

### Initialize the Game

After the **setting** of the **cursor,** open a while **loop**. This **loop** will **run** the **game flow**.

A picture containing logo

Description automatically generated

In it, create a string **variable** that will hold our **random** **generated** **word**. To the variable **assing** a **method** "GetRandomWord()" and pass the **words** **array** as a **parameter**. This **method** will help us **generate** a **random** **word** **each time** the **game** **starts** **over**.



The **method** will be **underlined** in **red**. This is **because** it is not **implemented** **yet**. Don't worry – when we are ready with the initialization of the game we will **comment** on the code and **uncomment** it slowly with the **implementation** of the **methods**.

Next, create another string **variable** which will represent our **random** **generated** **word** but all the **letters** that the **word** **contains** will be **replaced** with a **symbol** of your choice (in order to be hidden from the player). In our case, we are going to use **underscores**. You can do this like that:



\*Note: You can **create** a **constant** for the **underscore** **symbol** because we are going to **use it later in our code**. You know how to do this.

Also, create an int **variable,** which will hold the player's **incorrect** **guesses** **count** and a List<char> which will hold **all letters** the player **has already entered**. List<char> is like a **resizable** **array** **of type** char. It also has **methods** that help us **add**, **delete**, **sort**, and **search items** easily. You can learn more about it here: <https://www.geeksforgeeks.org/c-sharp-list-class/>.

We will use these **variables** to **draw** on the **console**.



Here we need another method "DrawCurrentGameState**()**". This **method** needs **four** **parameters**:

1. bool inputIsValid – a **boolean** **variable** will tell the **method** **whether** the **player's** **input** is **valid** or **invalid**.
   1. **False** 🡪 valid input.
   2. **True** 🡪 invalid input.
2. int incorrectGuess – the **incorrect** **guesses** **count** variable.
3. string guessedWord – the **word** **to** **guess** variable.
4. List<char> playerUsedLetters – the **list** of **letters** **used** by the **player**.

Text

Description automatically generated with low confidence

The **method** will help us **draw** the **current** **state** of the **game** on the **console**. We should **invoke the method** and **pass the arguments** we have in the following way:



After the **first state** of the **game** is **drawn** we need to **set one more method** – PlayGame(). This **method** will also need **four** **parameters**:

1. string word – **randomly** **generated** word **variable**.
2. string wordToGuess – **word** to **guess** variable.
3. int incorrectGuessCount – the **incorrect** **guesses** **count** variable.
4. List<char> playerUsedLetters – the **list** of **letters** **used** by the **player**.

A picture containing text

Description automatically generated

The **method** will run the **main logic of the game**. We should **invoke the method** and **pass the arguments** we have in the following way:



After we are done with this, what is **left** is to ask the **player** if they **want** to **play** **again** or **quit** the **game**. You can **implement** this **logic** by yourself.

Graphical user interface, text, application

Description automatically generated

Now we have our **game** **initialized and we can comment** on the code so that we implement and test the methods we create one by one. To do this you can **highlight** the **code inside** the while loop and **press** [Ctrl + K + C].

A screenshot of a computer

Description automatically generated with medium confidence

We will **uncomment** **slowly** the **code** with the **implementation** of the **methods**. Let's start with the **first** **one** which will help us **generate** a **random** **word**.

#### Implement the GetRandomWord() Method

Create the method, name it "GetRandomWord()" and make it accept **a** string **array** (the words we have). The **method** will **choose** a **word** on a **random** **index** and **return** it. It should look like this:

A close up of a word

Description automatically generated

Create a variable of type "Random". You already know how to do that:



Now create a string **variable** and name it "word". This will be our **random** **generated** **word**. Use our Random **variable** to get a **word** on a **random** **index**. Finally, **return** the **word** with the command **.ToLower()**, this will **make** **sure** the **word** is in **lower-case** :



Now we have the **method** **ready**. Go back to the **game** **loop** and **uncomment** the **first** **line** and set a **breakpoint** tothe **method**.

Timeline

Description automatically generated

Timeline

Description automatically generated

**Each time** you press [F10], it should display a **different word**. This is all we need for our **method** to work. Now we can **uncomment** the **code** **until** our next **method**. Let's implement the next method.

#### Implement the DrawCurrentGameState() Method

Now we should create the **method** DrawCurrentGameState() which will help us **draw** the **different** **stages** of the **game** on the **console**. The **method** should **accept** the following **parameters**, which will help us know **which game stage** we are currently at:

|  |
| --- |
| **(bool inputIsInvalid, int incorrectGuess, string guessedWord, List<char> playerUsedLetters)** |

It should look like this:

Text

Description automatically generated

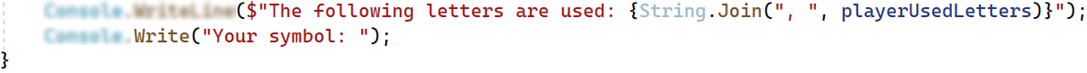
In the **method**, we should clear the **console** first because we want to **start** the **drawing** of the **current** **stage** from the **beginning** of the **console**. On the next line write on the **console** the **array** "wrongGuessesFrames" and as an **index** of the **array** pass the incorrectGuessCount variable:



On the next **two** **lines**, write on the **console** the **guessed** **word** and how many **symbols** the player has to **guess**. It should look like this:



Next, write on the console **which letters** the **player has entered**.



Now we have to **check** if the **player's input is invalid**. If the player's **input is invalid**, it means that the player has **entered** more or less **than one character**, so **write** a **message** on the **console** to **explain** that the player only **needs** to **enter** **one character**.

Logo, company name

Description automatically generated

Finally, **write** on the **console** a **place** where the **player** will type his **symbol**.



This is all we need for our method to work. Let’s uncomment **the method** to see if it **works properly**. **Read** one **line** **from** the **console** **because** the **method** won't stop **iterating** and we can't see if it **works correctly**.

Graphical user interface, text, application, email

Description automatically generated

Finally, the **result on the console** should be as follows:

A screen shot of a computer

Description automatically generated

Graphical user interface, text

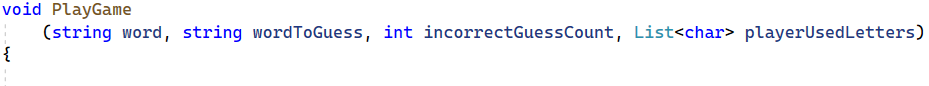
Description automatically generated

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

#### Implement the PlayGame() Method

Create the **method** PlayGame(). This **method** will **execute** the **main** **logic** of our **game**. The **method** will **read** the **player's** **input** **letter**, **check** if the **letter** is **contained** in our **random** **generated** **word** and will **replace** the **player letter** in the **hidden word** if the **word contains** it.

It will accept the **random word**, **the random hidden word with underscores**, the **number of wrong guesses** the player made, and the **list of letters the player used**.



In the **method** first, open a while**-**loop which will **iterate** until the **player guesses the word** or respectively **does not** **guess** it.

Text

Description automatically generated with medium confidence

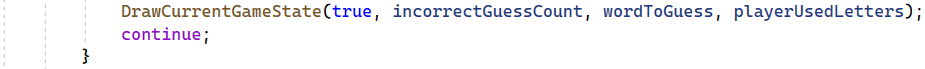
In the **loop**, **read** the **player's input** and make it **lower-cased**, this will **prevent you** from accidentaly typing **upper-cased letter** ,then check if the **input** **length** is **different than one**.

Картина, която съдържа текст

Описанието е генерирано автоматично

If the **length** is **different than one**, it means that the **player** has **entered** more or less than one **symbol**, so we need to **draw** on the **console** the **current game state** using our **method** and **continue** with the **next iteration**.

Pass true as the **first parameter** of the **method**. If true is **passed**, the **method** will **detect** that the **player** has **entered** **incorrect** **characters** and **display various messages** on the **console**.



If the check **passes**, **parse** the **player** **input** to char and **add** it to the **list of used letters**.

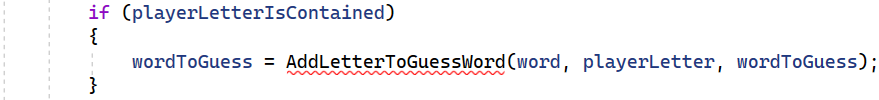


The next step is to **check** if the **player's** **letter** is **contained** in the **word**. **Create** a **method** CheckIfSymbolIsContained() for this **purpose** and **assign** it to a **boolean** **variable**. The **method** will **accept** the **word** and the **player's letter**.



\*Note: Again here we will **implement** the **methods** **needed** at the **end**.

If the **player's letter** is **contained**, we will create another method AddLetterToGuessWord() which will help us **add** a **letter** to the **word in the place it is contained**. The method will accept the **word**, the **player's** **letter** and the **hidden** **word**. Finally, **assign** the method to our **variable** wordToGuess.



In the other case when the player has not guessed a correct letter, **just** **increase** the **incorrect** **guesses** **count** by **one**. You know how to do this:

A picture containing text

Description automatically generated

Then we need to **call** the **method** for **drawing** the **current** **game** **state** with the **corresponding** **parameters** again, but this time **pass** false as the **first** parameter. By passing false, the **method** will know that the **player** **input** is **valid**.



It remains to **check** whether the player **has won** or **lost the game**. Let's start with the **winning** **case**. We will create a method CheckIfPlayerWins() which will **check** if the **player** **has** **won** the **game**. The **method** will **receive** the **hidden** **word** and will **return** true if the **player has won** and false if **not**. **Assign** it to a boolean variable.



If the player **wins** **clear** the **console**, write on the **console** the **winning** **screen** **constant** and a **message** that will hold the **guessed word**. Finally, **break** the **loop**.

Text

Description automatically generated with low confidence

Now do the same for the **losing** **case**, create a method CheckIfPlayerLose() which will help us to **know** if the **player** has **lost** the **game**. The **method** will **receive** the **player's incorrect guesses count** and will **return** true if the **player** has **lost** and false if not. **Assign** it to a **boolean** variable.



If the **player** **loses,** first set the **cursor** **position** to (0, 0), we do this to **escape** the **console** **blinking** while we **render** the **death** **animation**.

Text

Description automatically generated

After that, we will **create** a method DrawDeathAnimation() which will **render** our **death** **animation** **frames** **array**. The **method** will **accept** the **death** **animation** **frames** **array.**



After the **method** finishes **rendering, clear** the **console** and write on the **console** the **loss** **screen** **constant** and a **message** that will hold the **guessed word**. Finally, **break** the **loop**.

Graphical user interface, text, application

Description automatically generated

This is all we need for our **method** to **work**. It is the following:

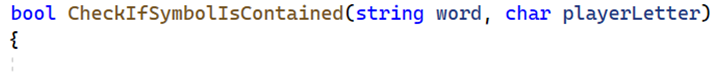
Graphical user interface, text, application, email

Description automatically generated

Now we just have to **initialize** the **methods** that it needs. Note that you should **comment the above code** in order to **test the implemented methods** one by one.

#### Implement the CheckIfSymbolIsContained() Method

Create a **method** and name it CheckIfSymbolIsContained(). This **method** will get the **word** and the **player's letter**, check if the **letter** is **contained** in the **word the user has to guess** and **return** true or false.



In the **method**, check if the **word** **does** **not** **contain** the **letter**. You can use the Contains() **method** to do this. If the check **passes**, **return** falseotherwise **return** true**.**

Graphical user interface, text, application

Description automatically generated with medium confidence

Test the method by yourself:

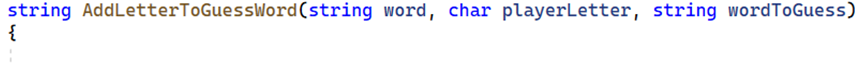




This is all we need for this **method** to work. Let's continue with the next one.

#### Implement the AddLetterToGuessWord() Method

Create a **method** and **name** it AddLetterToGuessWord(). The **method** will **receive** the **word**, the **player's** **letter** and the **hidden word,** will **replace** the **letter** onthe **index** itis **contained** in and **return** the **newly** **created** **word**.



In the **method** first, create a **char array** and **assign** to it the **hidden word**. Use the String.ToCharArray() **method** to **convert** the **word** to **char array**.



Now we need to go **through** **all** the **letters** in the **word to guess** with a for loop to check where in the word the guessed letter is contained.

A picture containing text

Description automatically generated

When the **player's letter matches** the **letter** of the **word** at the **current** **index**, **assign** the **player's letter** to the given place in the "wordToGuessCharArr" **variable**. It should look like this:

Text

Description automatically generated with low confidence

Finally, return a **new string** from the "wordToGuessCharArr" **char array**:



Test the method by yourself:

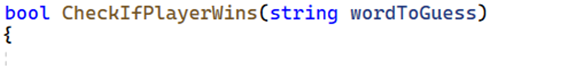




This is all we need for our **method** to work.

#### Implement the CheckIfPlayerWins() Method

Create a **method** and name it CheckIfPlayerWins(). The **method** will **receive** the **word** that the **player has to guess** and will help us to **check** if the **player** has **guessed** **the** **word**. We know the player **has won the game** when the word **does** **not** contain **underscores**.



In the **method**, we have to **check** if the **word** **contains** an **underscore**.

If the word **still contains** an **underscore** that means that the **player** has more **letters to guess** and we **should** **return** false.

Text

Description automatically generated with low confidence

In the other case, we should **return** true.



Test the method by yourself:

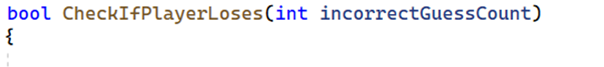




This is all we need to **finish** this **method**. We can continue with the next method.

#### Implement the CheckIfPlayerLoses() Method

Create a **method** and **name** it CheckIfPlayerLoses(). The **method** will **receive** an **integer** that **contains** the **player's** **incorrect** **guesses count** and will help us to know if the **player** **has lost the game**.



First, we should **create** a **constant** that **holds** the **max allowed incorrect guesses**. The **max incorrect guesses** are **equal** to the **length** of the wrongGuessesFrames **array** which **means 6**.



Note that the **constant** should be **created on the right place** in the code, not in the method itself.

Now when we have the **constant** go back to the **method** and inside **check** if the **player's incorrect guesses** **count** is **equal** to our constant **max allowed incorrect guesses.** If it is **return** true otherwise **return** false.

Text

Description automatically generated with medium confidence

Test the method by yourself:

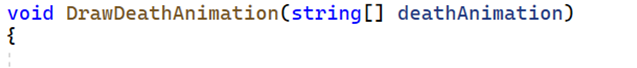




After we are done with this **method** we can **continue** with the **last one for this application**.

#### Implement the DrawDeathAnimation() Method

Create the method and name it "DrawDeathAnimation()". This **method** will help us render our **death animation** **array**. The **method** should receive **one argument** – **string array** with the **death animations**. It should look like this:



In the **method** first, open a for loop that will **iterate** to the **length** of the **death animation array**.

Text

Description automatically generated

In the for loop, write on the **console** the **death animation array** with index "i". You should know how to do that:



Now we have to use the "Thread.Sleep()" **method** which comes from **using** "System.Threading". This **method** will help us **pause** the for loop for a given **period**, in our case **200** **milliseconds**, and slowly **visualize** all **animations in the array**. Otherwise, our **animations will be rendered too fast,** and we won't be able to **see them**. It should look like this:



Finally, **set the cursor position to (0, 0)**. This will help us **draw** each animation **from the top of the console** and avoid **console blinking**.

Text

Description automatically generated with medium confidence

Now If everything is **fine** and we **don't** have any **errors** we can **go back** to the **game code** and **uncomment** the PlayGame() **method** to **test** if it **works properly**.

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated Text

Description automatically generated

We can see that if we **lost** the **game** the **death** **animation** is **rendered** and the **game** **restart**, and if we've **won** the **game**, it just **restarts**. This is **because** we have **more** **code** to **uncomment**. Go back to your **game** **initialization** and **uncomment** the **last** **part** of the **code**.

A computer screen shot of a computer code

Description automatically generated

After **uncommenting** the **last** **part** of the **code**, you should be **able** to see the **win**/**loss** **screen** at the **end** of the **game**.

Text

Description automatically generated Text

Description automatically generated

This is **all** we **need** for our **game** to **work** **properly**. Now let's **upload** it to **GitHub**.

## Upload Your Project to Github

We already know how to clone our repository by using **Git** **Bash** or **TortoiseGit**.

### Use TortoiseGit (Option 1)

Use **Git** **clone** for cloning with TortoiseGit. Go to the desired directory, **right-click** on a blank space anywhere in the folder and click [Git Clone**]**.

Graphical user interface, application

Description automatically generated

Now go to our newly created **repository** and copy the repository’s **URL** – you should already know how to do this. The last thing that we should do is to open TortoiseGit, paste the **URL** and click [OK].Graphical user interface, application, email

Description automatically generated

Your files from your **GitHub** repo will be downloaded to a **sub-folder** called as your project in GitHub, "**HangmanByPeter**" in our case.

Graphical user interface, application

Description automatically generated

The next thing to do is to **add** the **project** to our **cloned** **repository**. You can move your **C# source code** files from your **old project folder** to your **new repo folder**. You can use "**Cut & Paste**". It should look like this:Graphical user interface, text, application

Description automatically generated

Now **upload** the changes from our working project folder to **GitHub**.

We can use TortoiseGit's [GitCommit…]. Go to your project’s folder, **right-click** on a blank space anywhere in the folder and click [Git Commit -> "main"…].

Graphical user interface, application

Description automatically generated

Add an **appropriate** message and click [Add] so you don’t miss any files, finally click [Commit].

Graphical user interface, application

Description automatically generated

After that click [Push] and then [OK]:

Graphical user interface, text, application

Description automatically generated 🡪

Graphical user interface, application

Description automatically generated

This is all you need to **upload your project source code** to your **GitHub repository** using TortoiseGit.

### Use Git Bash (Option 2)

**Alternatively**, use **Git Bash** to **commit** and **push** your local changes to the **repo**.

Go to the desired **directory**, right-click on a blank space **anywhere** in the folder and select **[Git Bash Here]** to open the Git command line console. If the **[Git Bash Here]** menu is missing, you should first install Git.

Graphical user interface, application

Description automatically generated

Type the **"**gitclone**"** command followed by the link to your **repository**:

|  |
| --- |
| git clone {Repository URL} |

This command is for cloning with **Git Bash**, paste your **repository** **URL** after the command.

Text

Description automatically generated

Your files from your **GitHub** repo will be downloaded to a **sub-folder** called as your project in **GitHub**, "**HangmanByPeter**" in our case.

Graphical user interface, application

Description automatically generated

The next thing to do is to **add** your **project files** to your **cloned** **repository folder**. It should look like this:

Graphical user interface, text, application

Description automatically generated

Now we are ready to upload our changes from the **Git Bash clone**. Go to the desired **folder**, right-click on a blank space anywhere in the folder, select "GitBashHere" and run the following **commands**.

Type the following command:

|  |
| --- |
| git status |

The **git status** command displays the state of the working directory and the **staging area**.

A screenshot of a computer program

Description automatically generated

Now type:

|  |
| --- |
| git add . |

This command **adds** all modified files.

Next type:

|  |
| --- |
| git commit -m "Your message here" |

This command**commits** your changes. We also should **add** an appropriate **message**.

Second to the last type.

|  |
| --- |
| git pull |

This command **updates** your local **repository**.

Now the last thing that we should do is to **push** our changes by using the command:

|  |
| --- |
| git push |

This command **pushes** your changes to our local **repository**.

A screenshot of a computer

Description automatically generated with medium confidence

This is all you need to **update** your **repository** withGit Bash.

## \* 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.

### Guess the Full Word

Consider a way to **add an option** for the player to **guess the full word**.

### Add Rendering Animation for the Win

You can **add a rendering animation** for the **winning stage of the game**.

### Suggest Help to Player

Implement functionality to **help the player** if they want. For example, you can **reveal a letter from the word** for them.

### Create GUI App

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

### Additional Ideas

* You can think of a more interesting way to **draw the game animations**.
* 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!

|  |  |
| --- | --- |
| Icon  Description automatically generated | It is very important to **commit frequently** your code to GitHub. This way you create a **rich commit history** for your project and your GitHub contribution graph is growing:  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:

Graphical user interface, text, application, email

Description automatically generated

### Documentation Sections

Add **information** about your project in your README.md file: 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)

### Use Markdown

Note that the GitHub README.md file is written in the **Markdown language**. Markdown combines text and special formatting tags to describe formatted text documents.

### 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 it!**

Graphical user interface, text, application, email

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:



You can add information about the **inputs** and **outputs** of the project:

Graphical user interface

Description automatically generated with medium confidence

### Your Solution

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

### Link to the Source Code

Add a **link** to your **source code** as follows:



### 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 "Hangman" game:

Graphical user interface

Description automatically generated

## 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.

We have two **options** to **choose** from, first **option** is to use **Mono C#**. **Mono C#** is not **updated** yet and still works with **.NET 5.0**. This **means** we should **remake** our **code** to **.NET 5.0** to **work** in **Mono C#**. The **second** option is to **create** a **C# replit project**. The **C# replit project** is working with **.NET 6.0** and we can **simply** **copy** and **paste** our **project** **files** into it. The only **difference** in the **final** **result** is that the **Mono C# is working a little faster** than the **C# Replit project**.

### Option 1 – Mono C#

You **can't** **copy** and **paste** your **code** in your **Mono** **C#** **replit** **project** because our **project** is **written** on .**NET 6.0**, so your **option** is to **create** a **C#** **console** **project** on **.NET 5.0 locally in VS** first and **copy** your **project** **files** **to it**. Then **errors** will **pop** up and **you** **should** **resolve** **them**. If you **manage** to **resolve** the **errors**, you can **copy** and **paste** **your** .NET 5.0 code into the **Mono C# Replit project**.

### Option 2 – C#

Your **second** **option** is to **create** a **C# replit project**. This **way** you **only** **have** to **add** your **project** **files** to the **Replit** **project**. Keep in **mind** that **C# replit projects** run the **application** **slower** than **Mono C#.**

Whichever option you choose, you only need to **make one change** to the **code for the game to work properly**. Find the string[]words variable and **change** the **path** that we give to the "File.ReadAllLines()" **method** to contain only the file name – "words.txt". This is **enough** for **Replit** to **find** the file. It should look like this:



Choose **an** **option** and **create** your **Replit** **project** by **yourself** so you can **share** your app with your friends.

## Add Replit Link to Your README.md

Now add a "**one-click live demo**" of your project from your **GitHub** project documentation. You can do it as follows:

Graphical user interface

Description automatically generated with low confidence

You can take a **screenshot** from Replit.com and **paste it** into the GitHub documentation editor directly with **[Ctrl+V]**.

This is what it should look like after the changes in your README.md documentation:

Graphical user interface, application

Description automatically generated

Now we have completed our **console game** and we have a new **project** in our GitHub portfolio.