Assessment – Gamertag Project

This assessment includes the following:

1. **Clarify the task** and understand the project requirements,
2. **Identify** the expected specifications, standards and guidelines for the project
3. **Create a console-based application** that loads from file a list of gamertags and presents the information on screen in a variety of ways
4. **Create and conduct simple tests** and confirm code meets design brief
5. **Documenting actions** carried out and **results of tests** performed

***Complete this document and submit as part of your final assessment.***

Contents

[Task Description 3](#_Toc90366285)

[Expected Output 3](#_Toc90366286)

[Identify Standards and Guidelines 4](#_Toc90366287)

[Step 1 – Answer each statement by selecting True/False 4](#_Toc90366288)

[Step 2 – Identify one other Programming Standard/Guideline 5](#_Toc90366289)

[Getting Started 6](#_Toc90366290)

[Step 1 – Download the AIE template project 6](#_Toc90366291)

[Step 2 – Unzip the file to your chosen working folder 6](#_Toc90366292)

[Step 3 – Open the Visual Studio solution 6](#_Toc90366293)

[Console-based Program 7](#_Toc90366294)

[Step 1 - Create a Gamertags class 7](#_Toc90366295)

[Step 2 - Loading the “Gamertags.txt” file via the Gamertags class 8](#_Toc90366296)

[Step 3 – Printing all available gamertags 10](#_Toc90366297)

[Step 4 – Create an instance of the Gamertag class 11](#_Toc90366298)

[Step 5 – Test your current progress 12](#_Toc90366299)

[Step 6 – Filter and Display gamertags ending with a number 13](#_Toc90366300)

[Step 7 - Filter and Display gamertags NOT starting with a “letter or number” 14](#_Toc90366301)

[Step 8 – Display a Welcome Message 15](#_Toc90366302)

[Step 9 – Processing user input 16](#_Toc90366303)

[Step 10 (Optional) – Create other *filters* for gamertags 16](#_Toc90366304)

[Step 11 (Optional) – Getting creative 16](#_Toc90366305)

[Debugging Exercise 17](#_Toc90366306)

[Step 1 - Answer the following questions about debugger usage 17](#_Toc90366307)

[Final Testing 18](#_Toc90366308)

[Step 1 – Perform the following tests and record the results 18](#_Toc90366309)

[Step 2 – Fix any errors in your program before submitting your assignment 18](#_Toc90366310)

### 

## Task Description

In this assignment you will:

* Create a **console application** that
* **Loads from file** a *list of Gamertags,* and
* **Displays information about the Gamertags** to the console in several ways as described in the “*Console-based Gamertag program”* section below

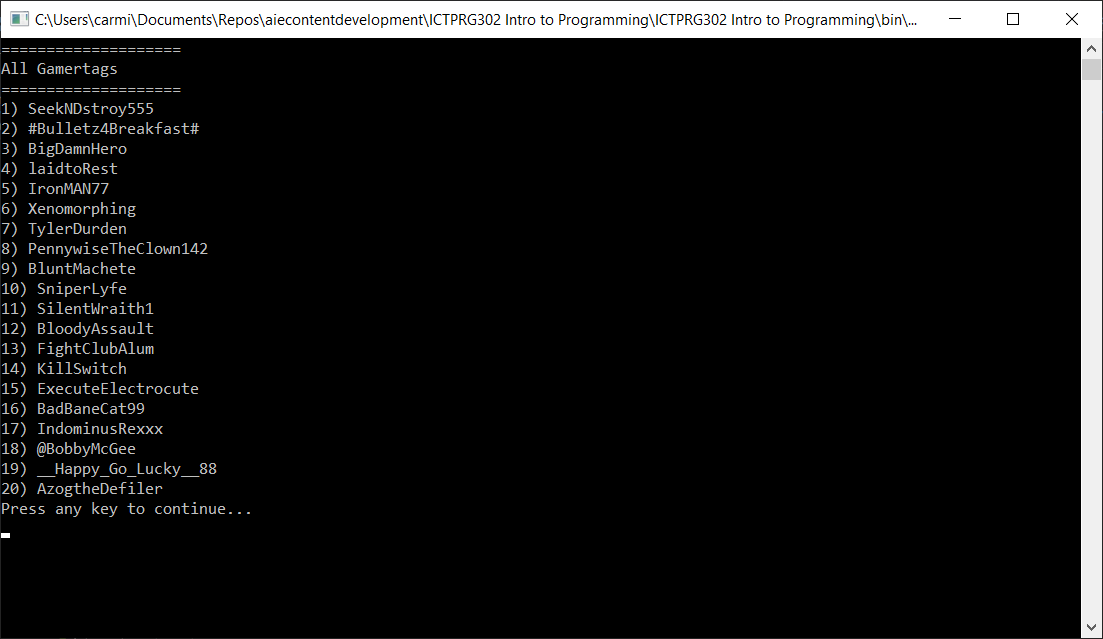
The program will be:

* Written in the **C# programming language**
* Using the **Visual Studio IDE** (Integrated Development Environment)

Read through these instructions to understand what’s required from your application and what expected output is to be presented.

## Expected Output

The following screenshot is an example of some expected output from your completed program.



## Identify Standards and Guidelines

Programming Standards, Conventions and Guidelines aim to provide programmers with a recommended, consistent set of rules to facilitate code maintenance, increase readability and reduce the opportunity for the bugs. There are several available on the internet by large companies and foundations. Often teams will create a custom set of guidelines for their specific project(s).

### Step 1 – Answer each statement by selecting True/False

|  |  |  |
| --- | --- | --- |
|  | Which of the following statements are True? | True/False? |
| 1 | The following *Capitilization Guidelines,* which form part of the *Microsoft Framework Design Guidelines:*   * Recommend *simple & consistent rules* for using case that * Make identifiers for types, members, and parameters *easy to read*   <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/capitalization-conventions> | TRUE / FALSE |
| 2 | The following *MSDN General Naming Convent*ions recommends:   * Favouring *brevity over readability* when naming variables, classes and properties. Implying that short, unreadable names are better than longer names that are readable and easily understandable * Using of *hungarian notation* for identifier names * Using *abbreviations* whenever possible   <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions> | TRUE / FALSE |
| 3 | The following *MSDN C# Standards & Guidelines* aim to:   * *Create a consistent look* to the code * Enable readers to *understand the code* more quickly * Facilitate *copying, changing, and maintaining* the code * Demonstrate *C# best practices*   <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions> | TRUE / FALSE |
| 4 | The following *MSDN Layout Conventions* aim to:   * Encourage programmers to *express their individual style* * Make it *difficult to read* the code * Make the code *harder to maintain*   <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions#layout-conventions> | TRUE / FALSE |
| 5 | The following MSDN Field Design recommends:   * Using *constant* fields for values that will never change   <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/field> | TRUE / FALSE |

### Step 2 – Identify one other Programming Standard/Guideline

1. Using the following URL as a starting point:
   1. <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/>
2. *Select one section*, then identify a recommended guideline contained in it.
3. Document the recommendations below

|  |
| --- |
| **Copy/Paste the URL** of your selected MSDN Programming Standard/Guideline |
| *Your answer here…*  *e.g.* <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-classes-structs-and-interfaces> |

|  |
| --- |
| **Describe one positive recommendation** from the selected URL |
| *Your answer here…*  *e.g.* ✔️ DO name classes and structs with nouns or noun phrases, using PascalCasing. |

|  |
| --- |
| **Describe one negative recommendation** guideline from the selected URL |
| *Your answer here…*  *e.g.* ❌ DO NOT give class names a prefix (e.g., "C"). |

## Getting Started

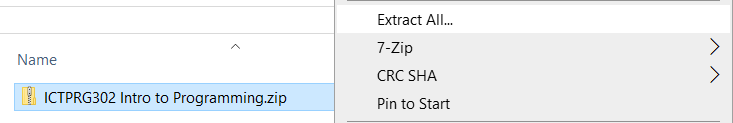
A *starting Visual Studio project* is provided on AIE Canvas. It includes the starting files necessary to complete this assignment.

### Step 1 – Download the AIE template project

*Download the starting project* named “**ICTPRG302 Intro to Programming.zip**” from Canvas to your computer.

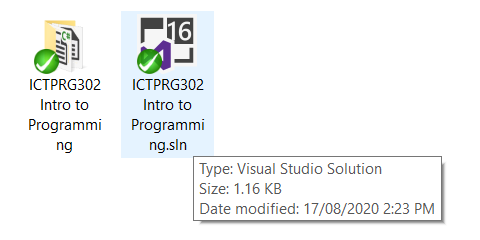
### Step 2 – Unzip the file to your chosen working folder

Unzip the file to your working folder of choice.

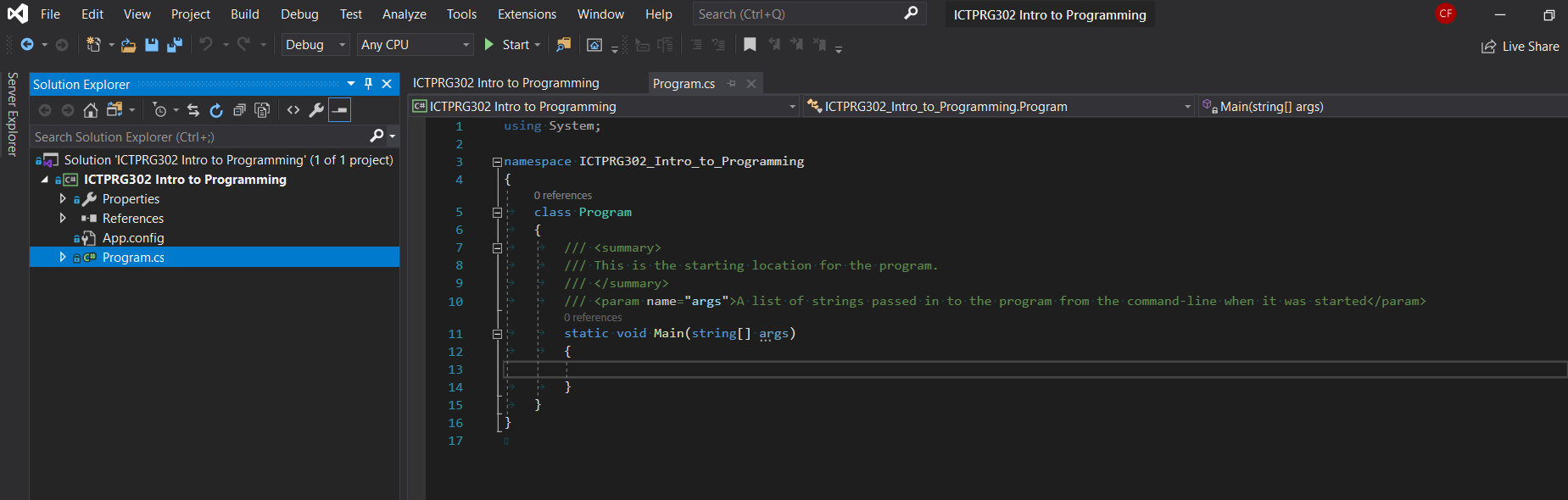


### Step 3 – Open the Visual Studio solution

Several files and folders will be unzipped to your selected location, including the Visual Studio solution file, called “**ICTPRG302 Intro to Programming.sln”**

****

*Double-click* the “**ICTPRG302 Intro to Programming.sln”** file to open it in Visual Studio. You should be presented with something like this:



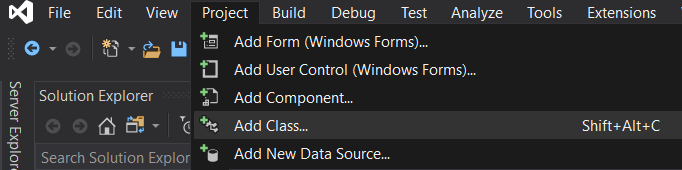
## Console-based Program

### Step 1 - Create a Gamertags class

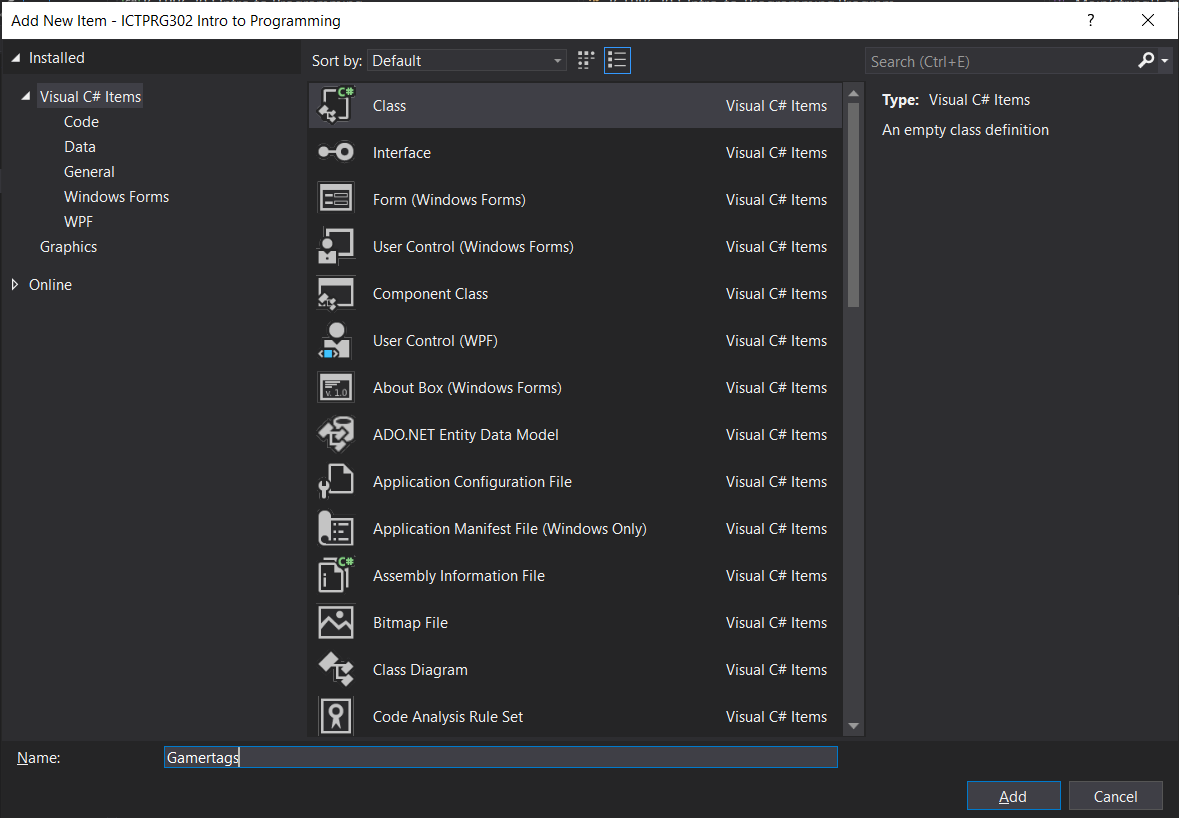
We will create a new class (a *user-defined datatype*) to hold the *related data and functions* required to manage the list of gamertags.

The most important data this class will contain is the list of Gamertags. This will be stored in the class as an *array of strings*.

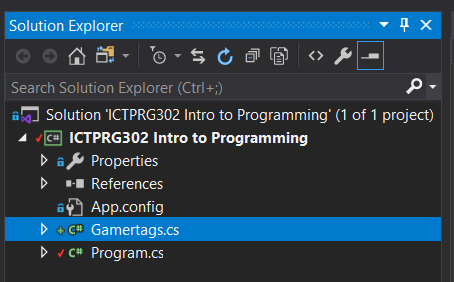
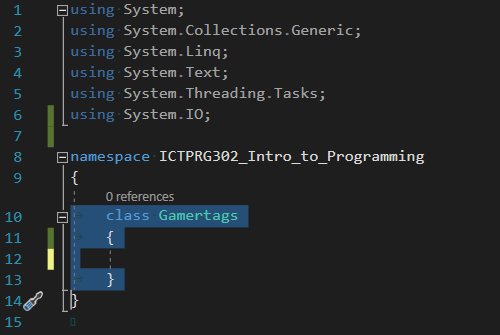
1. Select the **Project->Add Class** menu item



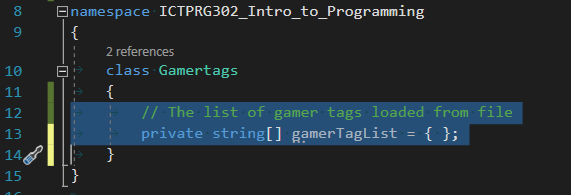
1. Select a *Basic C# Class* type. Name the new class Gamertags.



1. Notice that a new file has been added to the project called Gamertags.cs and it’s immediately opened in the code editor tab. This is where we add our data/functions for this class.

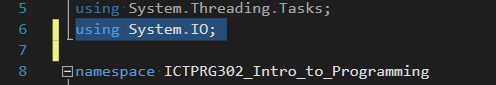
1. Add a new private variable that will later be used to hold our list of gamer tags once they’re loaded from file. Optionally add a comment to help describe the class variable. The private keyword restricts access to data (other options public/protected).



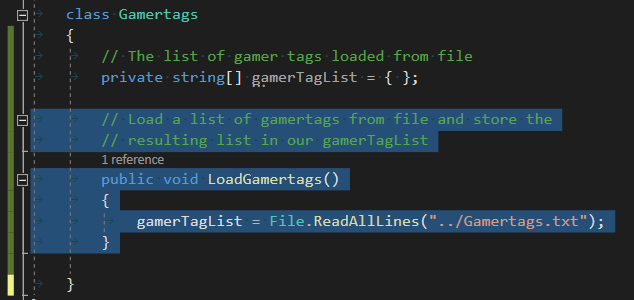
### Step 2 - Loading the “Gamertags.txt” file via the Gamertags class

Functions *add helpful operations to our classes* that we can call/run/execute from elsewhere in our code. They contain a sequence of operations to process (in order). This can be as simple as a single instruction, or something more complex.

1. Include the following usingclause at the top of your file to allow the compiler to find the File class in a namespace called System.IO

**

1. Add a function called LoadGamertags to this class as follows:

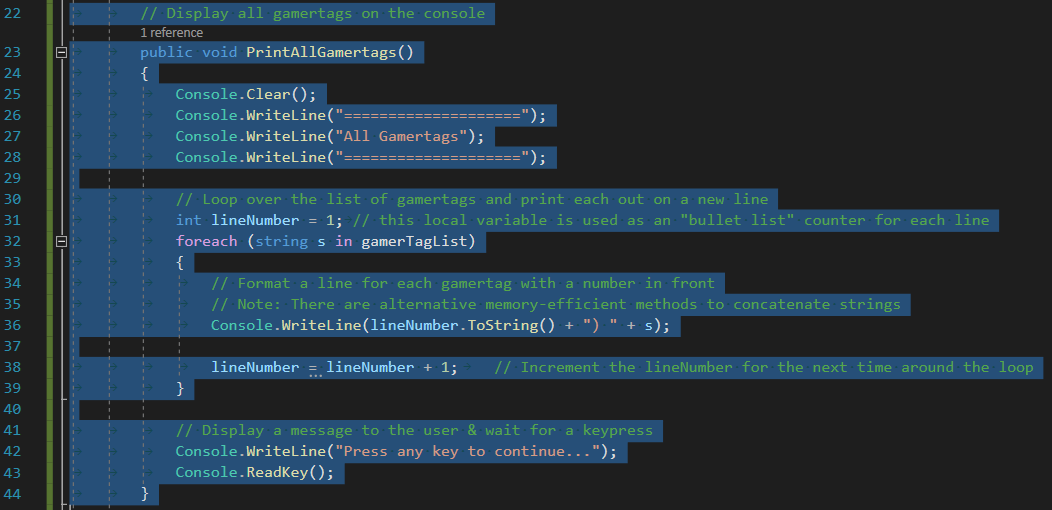


This will load a file called “Gamertags.txt” from disk and stores the data in the gamerTagList variable. The “../” asks the compiler to look for the file in the parent folder from where it starts up, which happens to be the “ICTPRG302 Intro to Programming\bin” folder.

### Step 3 – Printing all available gamertags

We’ll add another function to loop over the list of gamertags and print each to the console one line at a time. We’ll print a header at the start. We’ll also wait for the user to press a key before continuing the program.

1. Add a function called PrintAllGamertags to this class as follows:

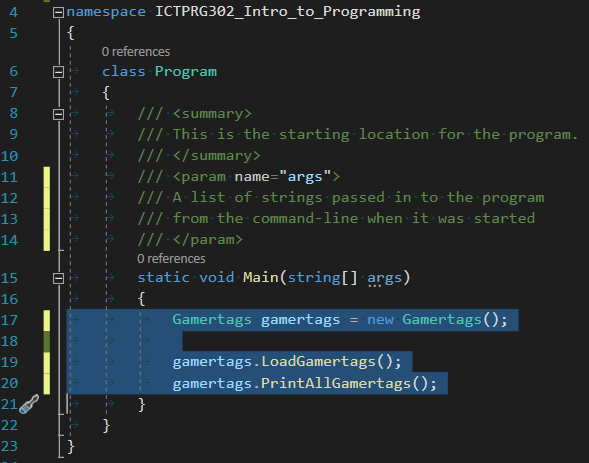


This function prints a 3-line header, loops over all the gamertags in the gamerTagList and prints each to a new line in the Console. Finally, it prints a message to the user and waits for the user to press any key before continuing.

### Step 4 – Create an instance of the Gamertag class

Up to now, the class we’ve created hasn’t been “used” yet, and our program will not execute any functions we’ve written. This is because a class can be thought of as a blueprint. We need to *create an instance* of it (like making an object from any blueprint) and then use the instance in our program.

In the Program.cs file add the following code to create a new instance of the Gamertag class and store it in a local variable called gamertags.



After creating an instance of the Gamertags class, our Main() function can call the public functions we created. We’ll call the functions in a sequence that makes sense.

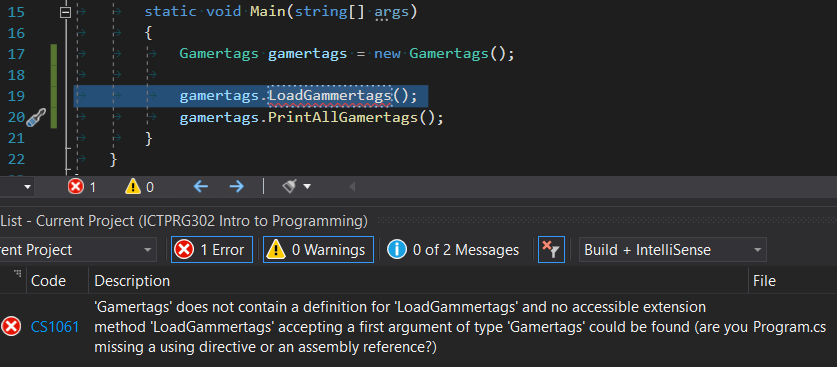
*Firstly*, we want to load the gamertags from file.

*Secondly*, we want to display the loaded gamertags to the Console.

Note that the computer follows our commands in sequence, like instructions in a recipe, step-by-step.

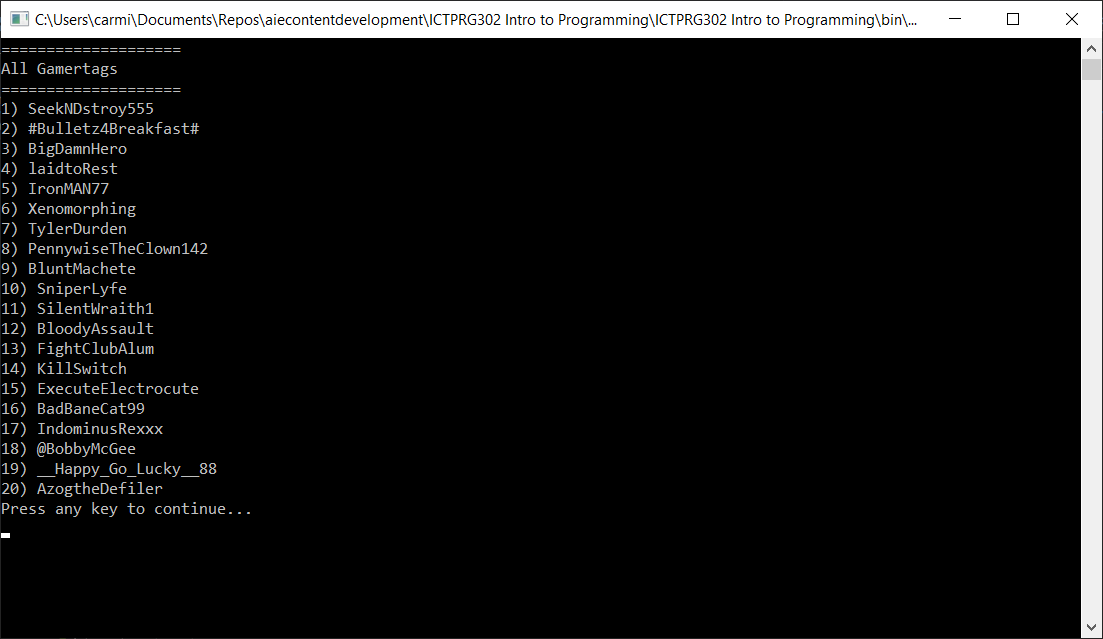
### Step 5 – Test your current progress

*Build and run your program* by pressing the Start  button in the Visual Studio toolbar. If there are *syntax errors* the compiler will display a message. For example, if you make a typo, you may see something like:



Double-click the error message to bring the error in to focus in the editor. Then make any necessary changes before trying again.

After all syntax errors have been fixed, Start your program. You should see the following output:



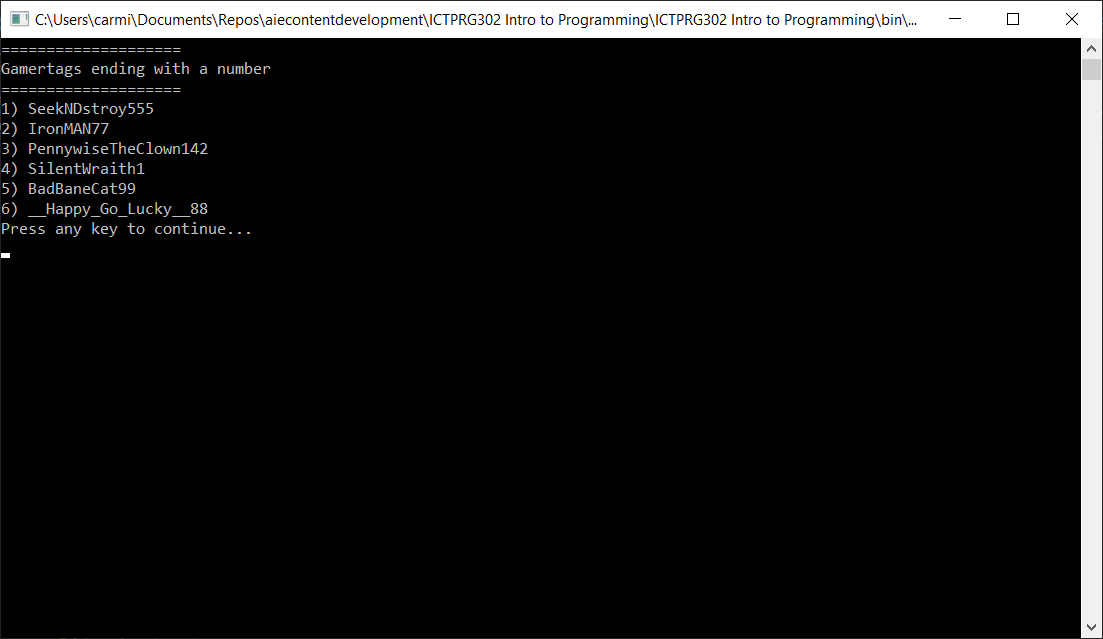
### Step 6 – Filter and Display gamertags ending with a number

Add the following function to the Gamertags class that filters gamertags ending with a number and ignore all the others.



Note the “if statement” being used inside the “foreach loop”. We test each gamertag in the gamerTagList to ensure it has *at least one character* and the *last character* is a number. If both tests are valid then the if-body will be processed, and the program will print the gamertag.

You should see the following output:



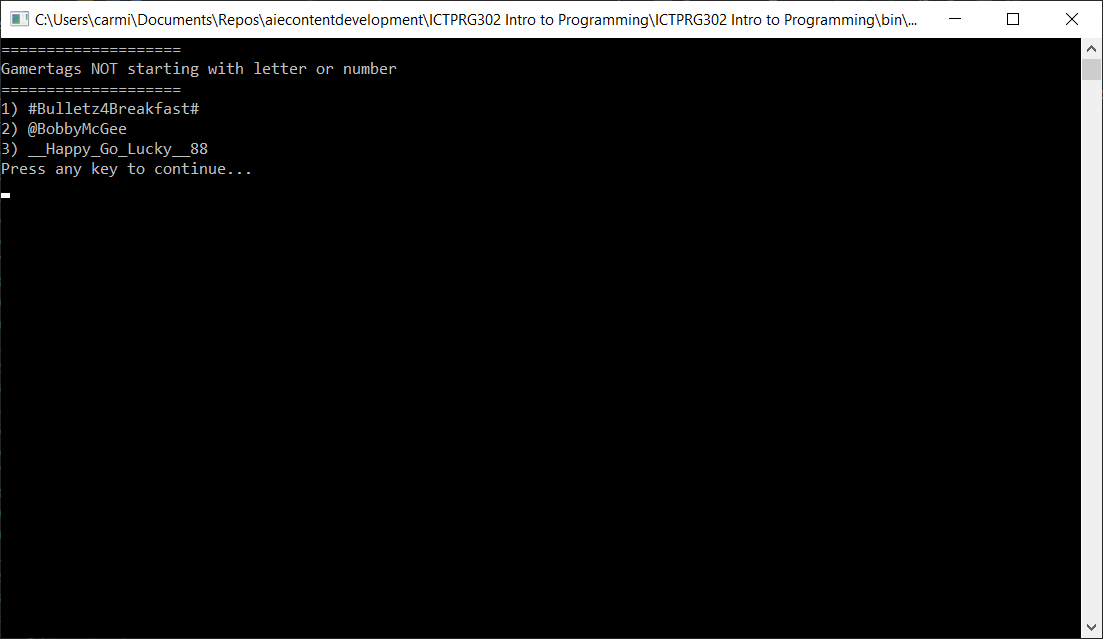
### Step 7 - Filter and Display gamertags NOT starting with a “letter or number”

Add a function to the Gamertags class such that it will only display gamertags NOT starting with a letter or number.

Research the C# functions Char.IsLetterOrDigit() for more assistance:

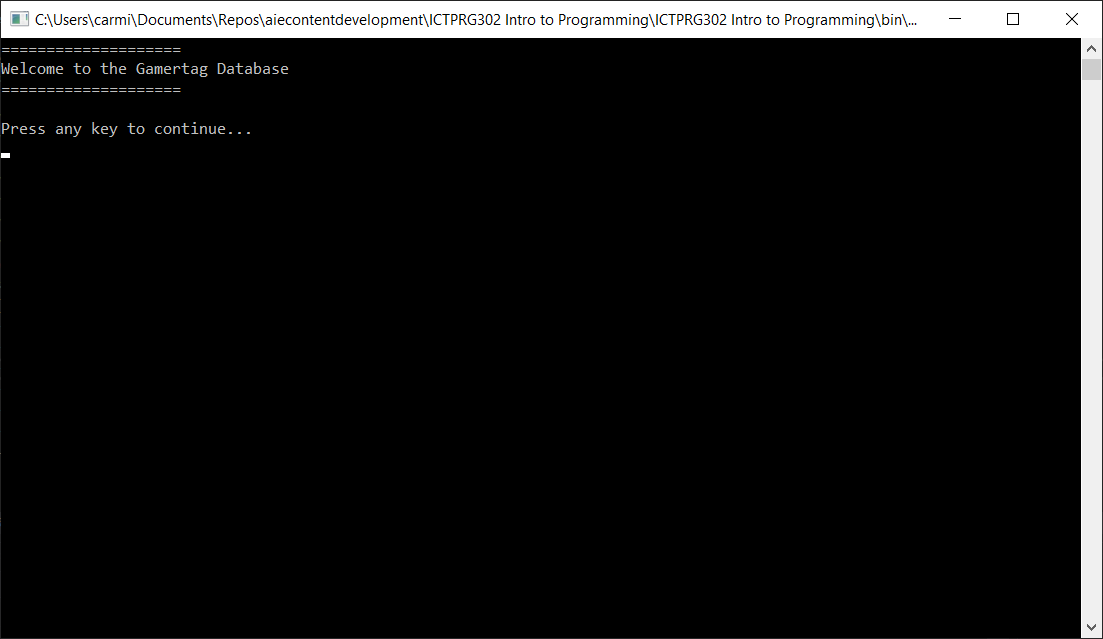
* <https://docs.microsoft.com/en-us/dotnet/api/system.char.isletterordigit?view=netcore-3.1>

You should see output like the following:



### Step 8 – Display a Welcome Message

Using the previous steps and practice, create a new function “ShowWelcomeMessage()” that your program uses to display a formatted welcome screen similar to the following (feel free to use some creativity when designing your own) . Wait for the user to press a key before continuing with the rest of the program:



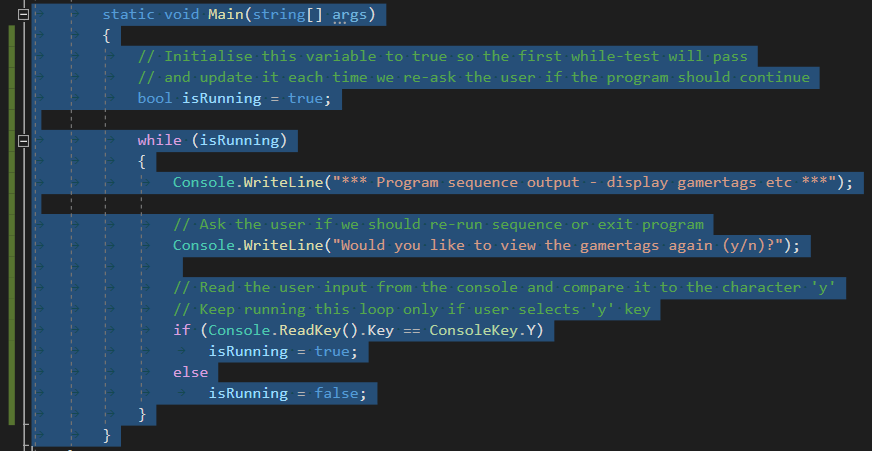
### Step 9 – Processing user input

At present your program runs through its sequence once and terminates. What if your user wants to view the output again without having to re-run the program?

Update your Main() function with a while-loop that:

1. Runs the sequence and output gamertags as previously
2. Asks the user if they would like to view the output again
3. Process the user’s input to *rerun the sequence* or *quit the program*

Here’s an example of a while-loop that asks the user whether to re-run the program:



### Step 10 (Optional) – Create other *filters* for gamertags

Extend your basic program to include other filters for the list of gamertags.

Examples include:

* Print all Gamertags that include only *“letters or numbers”*
* Print all Gamertags in “uppercase”
* Print the “first 5” Gamertags, ignoring the rest

### Step 11 (Optional) – Getting creative

Think about ways that you can extend your basic program to include more user-interactivity.

Examples include:

* *Adding a menu* that responds to user-input. Allow the user to choose what to display.
* *Adding new gamertags* to the list, by asking the user for them to input a gamertag
* *Saving the gamertags to file* when the program ends (see File.WriteAllLines())

## Debugging Exercise

The debugger is a powerful integrated tool in the Visual Studio Environment. It’s used to help programmers *find and fix logic errors* in their programs. These errors are colloquially called *bugs*.

### Step 1 - Answer the following questions about debugger usage

|  |
| --- |
| What are **breakpoints** used for? |
| *Your answer here…* |

|  |
| --- |
| What Visual Studio **windows/tab(s)** are typically used to interrogate the values of a variable during a debugging session? |
| *Your answer here…* |

|  |
| --- |
| What is “**stepping**” through code? Why do we do it? How do we “step” through code whilst debugging? |
| *Your answer here…* |

|  |
| --- |
| Explain **one situation during development** of your application where the debugger was helpful? What was the problem you faced? How was the debugger helpful? |
| *Your answer here…* |

## Final Testing

To verify the correct operation of your program, you must **create, perform and document tests**. The tests will assist in confirming that your program *satisfies the original task description*.

### Step 1 – Perform the following tests and record the results

### Step 2 – Fix any errors in your program before submitting your assignment

|  |  |
| --- | --- |
| Confirm that your program outputs the following gamertags when “Printing ALL Gamertags” | **TEST PASSED?** |
| SeekNDstroy555  #Bulletz4Breakfast#  BigDamnHero  laidtoRest  IronMAN77  Xenomorphing  TylerDurden  PennywiseTheClown142  BluntMachete  SniperLyfe  SilentWraith1  BloodyAssault  FightClubAlum  KillSwitch  ExecuteElectrocute  BadBaneCat99  IndominusRexxx  @BobbyMcGee  \_\_Happy\_Go\_Lucky\_\_88  AzogtheDefiler | YES / NO |

|  |  |
| --- | --- |
| Confirm that your program displays the following when “Printing Gamertags Ending with a number” | **TEST PASSED?** |
| SeekNDstroy555  IronMAN77  PennywiseTheClown142  SilentWraith1  BadBaneCat99  \_\_Happy\_Go\_Lucky\_\_88 | YES / NO |

|  |  |
| --- | --- |
| Confirm that your program displays the following when “Printing Gamertags NOT starting with a letter or number” | **TEST PASSED?** |
| #Bulletz4Breakfast#  @BobbyMcGee  \_\_Happy\_Go\_Lucky\_\_88 | YES / NO |