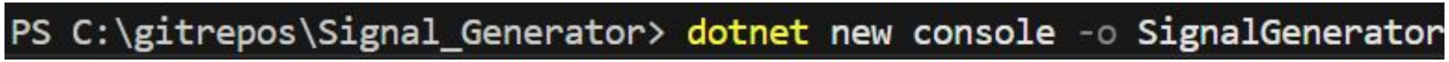
# Signal Generator(C# Emulator)

Altaf Abdulla

## Objective

- To demonstrate OOP knowledge through C# in designing a signal generator
    - Setting up VSCode
    - Creating  Classes and Objects
    - Creating a Dictionary and assigning a constructor
    - Creating the base class and inheritance
    - How to compile and execute

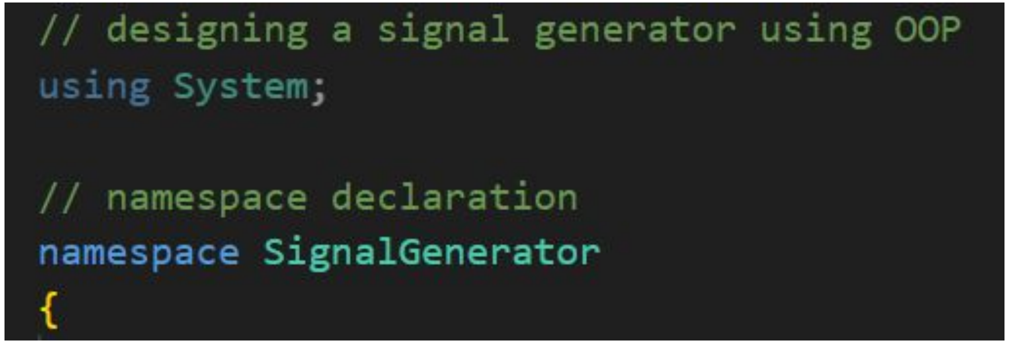
## Setting Up VSCode



```
PS C:\gitrepos\Signal_Generator> dotnet new console -o SignalGenerator
```

## Setting Up VSCode

1. First established new C# console on VSCode folder using:
   a. dotnet new console -o SignalGenerator
2. After console opens, navigate to Program.cs and open the file
3. Call the following commands to import a namespace with preset classes and methods, and create another namespace to design program specific classes
   a. using System;
   b. namespace SignalGenerator {}



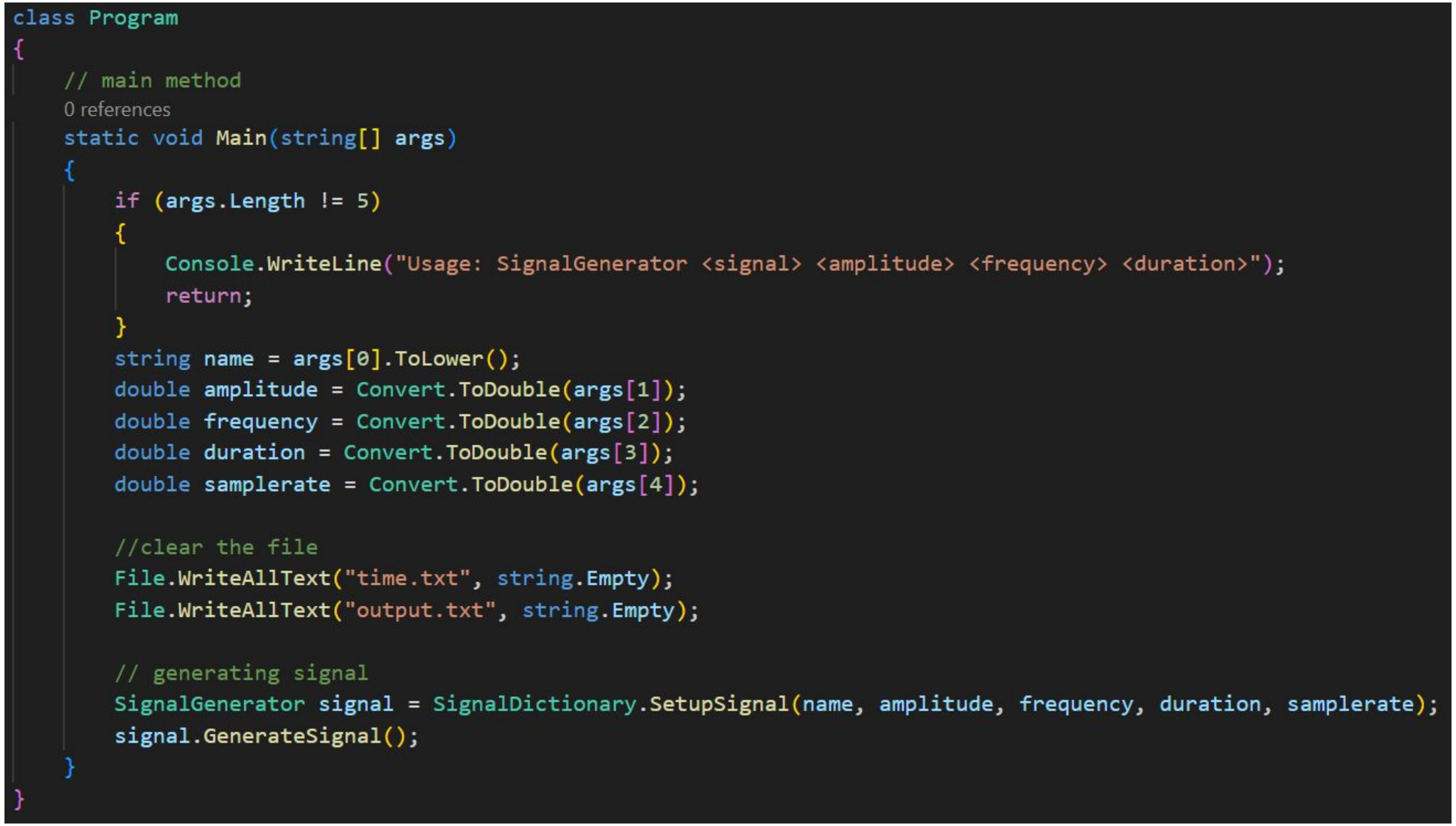
```
// designing a signal generator using OOP
using System;

// namespace declaration
namespace SignalGenerator
{
```

## Creating Classes and Objects

## Creating Entry Class

- Created Program class as Entry class, serving as entry point
  - Encapsulates Main method which is where execution starts
  - Main takes in string array of arguments, which can be given when executing
  - Then each array value is taken and assigned to respective variables
  - Output files which may contain values from previous executions are cleared
  - An object called signal is created, using the SignalDictionary Class
    - Separate class to map signal names found in arg[0] to corresponding constructor functions
  - A method is executed when signal.GenerateSignal()
    - Calling the GenerateSignal method found in the SignalGenerator class, which will accomplish different operations based on signal specified



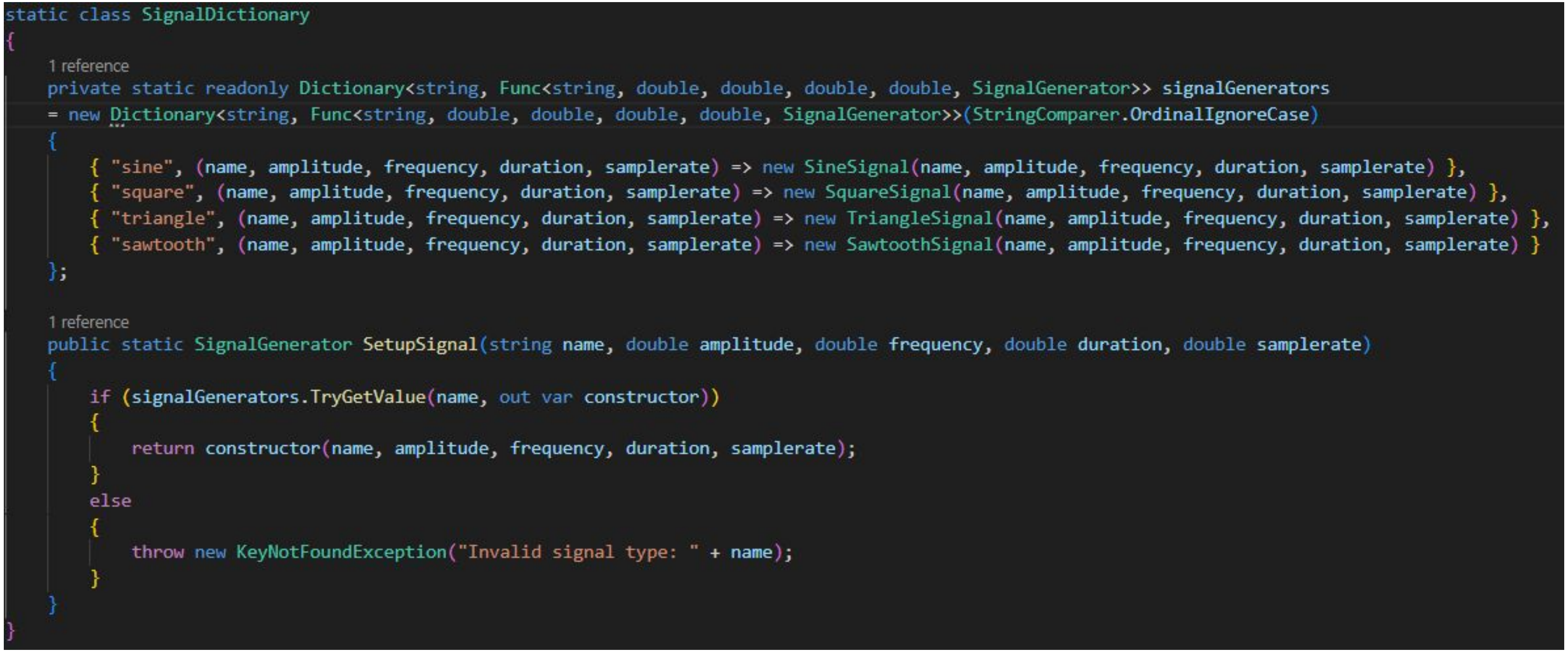
```
class Program
{
    // main method
    0 references
    static void Main(string[] args)
    {
        if (args.Length != 5)
        {
            Console.WriteLine("Usage: SignalGenerator <signal> <amplitude> <frequency> <duration>");
            return;
        }
        string name = args[0].ToLower();
        double amplitude = Convert.ToDouble(args[1]);
        double frequency = Convert.ToDouble(args[2]);
        double duration = Convert.ToDouble(args[3]);
        double samplerate = Convert.ToDouble(args[4]);

        //clear the file
        File.WriteAllText("time.txt", string.Empty);
        File.WriteAllText("output.txt", string.Empty);

        // generating signal
        SignalGenerator signal = SignalDictionary.SetupSignal(name, amplitude, frequency, duration, samplerate);
        signal.GenerateSignal();
    }
}
```

Program Class

## Dictionary Setup

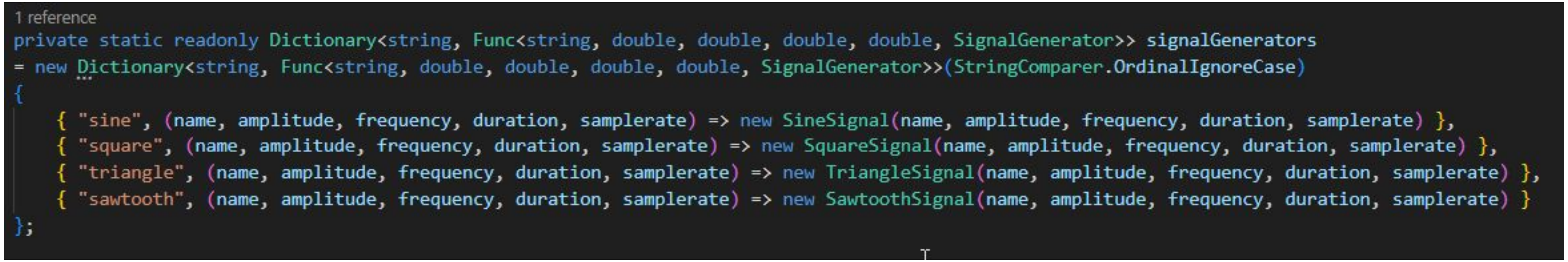


```
static class SignalDictionary
{
    1 reference
    private static readonly Dictionary<string, Func<string, double, double, double, double, SignalGenerator>> signalGenerators
    = new Dictionary<string, Func<string, double, double, double, double, SignalGenerator>>(StringComparer.OrdinalIgnoreCase)
    {
        { "sine", (name, amplitude, frequency, duration, samplerate) => new SineSignal(name, amplitude, frequency, duration, samplerate) },
        { "square", (name, amplitude, frequency, duration, samplerate) => new SquareSignal(name, amplitude, frequency, duration, samplerate) },
        { "triangle", (name, amplitude, frequency, duration, samplerate) => new TriangleSignal(name, amplitude, frequency, duration, samplerate) },
        { "sawtooth", (name, amplitude, frequency, duration, samplerate) => new SawtoothSignal(name, amplitude, frequency, duration, samplerate) }
    };

    1 reference
    public static SignalGenerator SetupSignal(string name, double amplitude, double frequency, double duration, double samplerate)
    {
        if (signalGenerators.TryGetValue(name, out var constructor))
        {
            return constructor(name, amplitude, frequency, duration, samplerate);
        }
        else
        {
            throw new KeyNotFoundException("Invalid signal type: " + name);
        }
    }
}
```

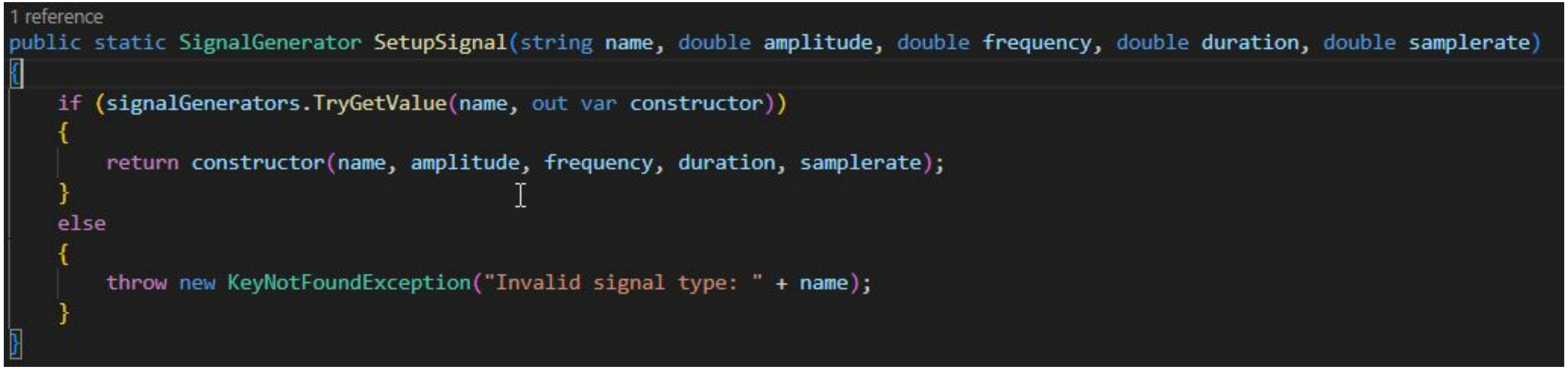
SignalDictionary Class - Used to Allow for Scalability and Ease of Use

- Dictionary Method
    - Declares a Dictionary named signalGenerators using a string as a keyword.
    - Uses Func<...> as method reference to take in the values from
    - Returns a SignalGenerator object
    - Based on the keyword, a specific constructor is formed, as per the dictionary definitions



```
1 reference
private static readonly Dictionary<string, Func<string, double, double, double, double, SignalGenerator>> signalGenerators
= new Dictionary<string, Func<string, double, double, double, double, SignalGenerator>>(StringComparer.OrdinalIgnoreCase)
{
    { "sine", (name, amplitude, frequency, duration, samplerate) => new SineSignal(name, amplitude, frequency, duration, samplerate) },
    { "square", (name, amplitude, frequency, duration, samplerate) => new SquareSignal(name, amplitude, frequency, duration, samplerate) },
    { "triangle", (name, amplitude, frequency, duration, samplerate) => new TriangleSignal(name, amplitude, frequency, duration, samplerate) },
    { "sawtooth", (name, amplitude, frequency, duration, samplerate) => new SawtoothSignal(name, amplitude, frequency, duration, samplerate) }
};
```

- Setting up Signal using Dictionary - Returning to Main to setup “signal”
  - Check to see if name is keyword listed in the Dictionary signalGenerators
  - If so, returns the constructor of the keyword to Main, where method was called
  - If not, an exception occurs, and program ends

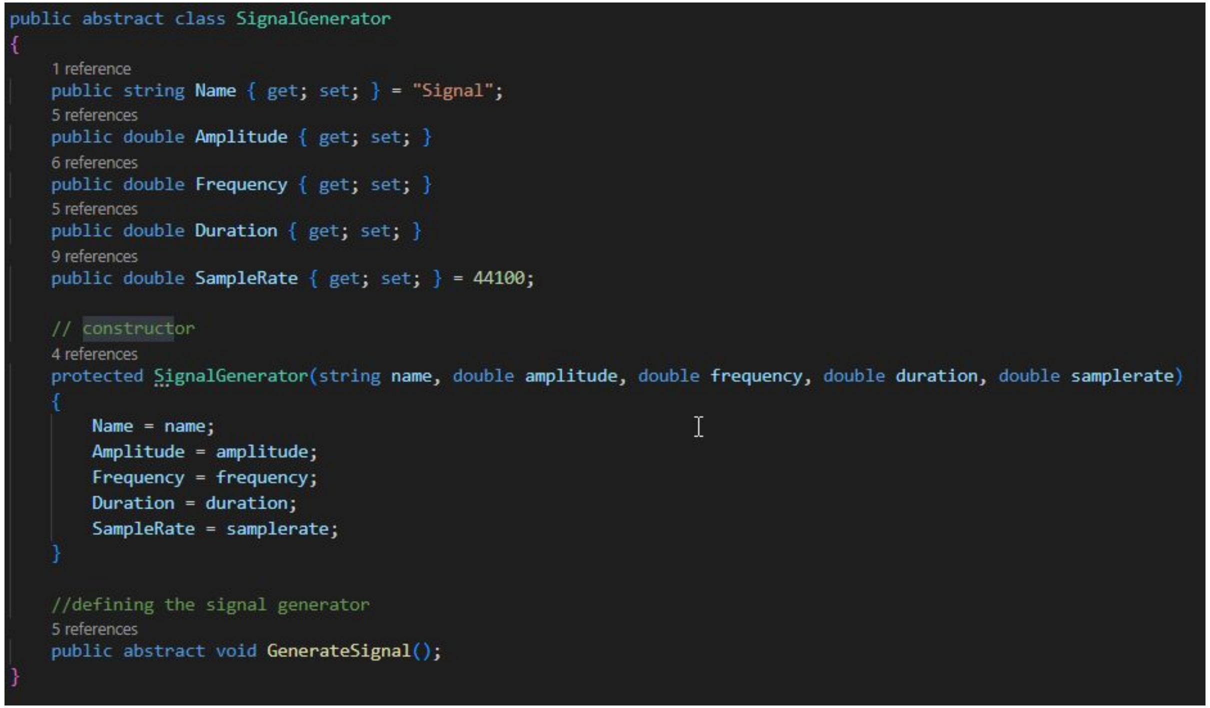


```
1 reference
public static SignalGenerator SetupSignal(string name, double amplitude, double frequency, double duration, double samplerate)
{
    if (signalGenerators.TryGetValue(name, out var constructor))
    {
        return constructor(name, amplitude, frequency, duration, samplerate);
    }
    else
    {
        throw new KeyNotFoundException("Invalid signal type: " + name);
    }
}
```

## Inheritance and Child Classes

## Usage of “abstract” Keyword

- Use abstract in base class and method to define common interface for all child classes.
- Class must be abstract as well to prevent instantiation
  - Meant to serve as base templates, not to be usable alone
- Why use protected for constructor?
  - Ensures required parameters are passed, while only letting child classes call constructor



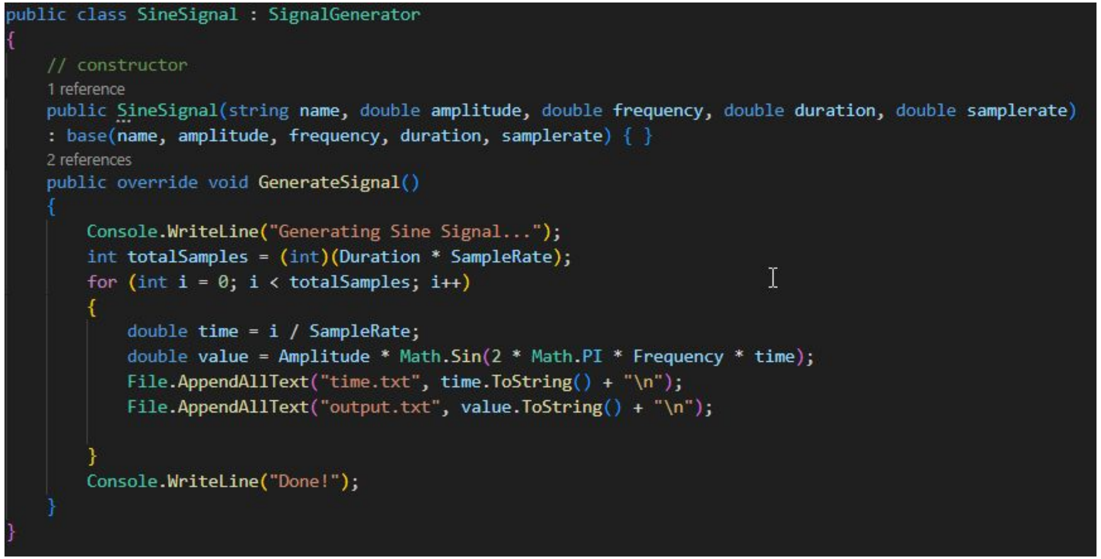
```
public abstract class SignalGenerator
{
    1 reference
    public string Name { get; set; } = "Signal";
    5 references
    public double Amplitude { get; set; }
    6 references
    public double Frequency { get; set; }
    5 references
    public double Duration { get; set; }
    9 references
    public double SampleRate { get; set; } = 44100;

    // constructor
    4 references
    protected SignalGenerator(string name, double amplitude, double frequency, double duration, double samplerate)
    {
        Name = name;
        Amplitude = amplitude;
        Frequency = frequency;
        Duration = duration;
        SampleRate = samplerate;
    }

    //defining the signal generator
    5 references
    public abstract void GenerateSignal();
}
```

## Child Classes - Example

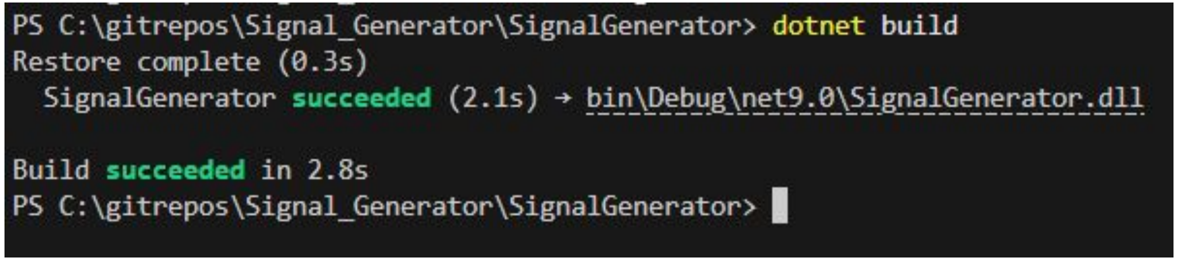
- Sets up constructor with values received from Main method.
  - Specific format calls the constructor of the base class and passes values into SineSignal for use
- Uses override keyword to write over GenerateSignal defined in base class, defining a SineSignal specific version of the GenerateSignal method
  - Allows to create waveform-specific methods for generating signal



```csharp
public class SineSignal : SignalGenerator
{
    // constructor
    1 reference
    public SineSignal(string name, double amplitude, double frequency, double duration, double samplerate)
    : base(name, amplitude, frequency, duration, samplerate) { }
    2 references
    public override void GenerateSignal()
    {
        Console.WriteLine("Generating Sine Signal...");
        int totalSamples = (int)(Duration * SampleRate);
        for (int i = 0; i < totalSamples; i++)
        {
            double time = i / SampleRate;
            double value = Amplitude * Math.Sin(2 * Math.PI * Frequency * time);
            File.AppendAllText("time.txt", time.ToString() + "\n");
            File.AppendAllText("output.txt", value.ToString() + "\n");

        }
        Console.WriteLine("Done!");
    }
}
```

## Compilation and Execution

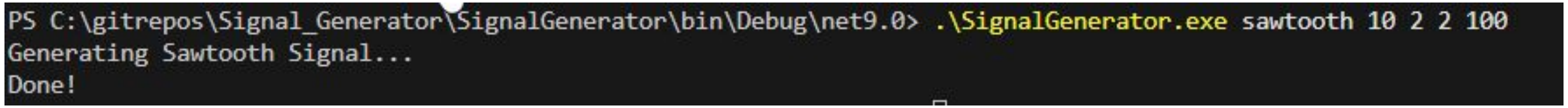


```
PS C:\gitrepos\Signal_Generator\SignalGenerator> dotnet build
Restore complete (0.3s)
  SignalGenerator succeeded (2.1s) → bin\Debug\net9.0\SignalGenerator.dll

Build succeeded in 2.8s
PS C:\gitrepos\Signal_Generator\SignalGenerator> 
```

## To Compile and Execute

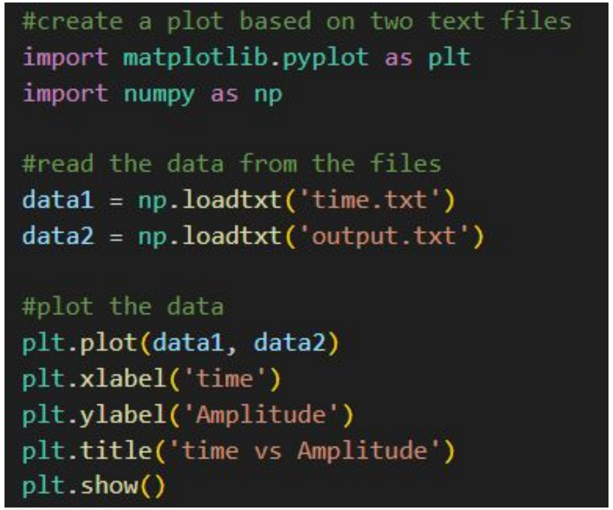
1. Write dotnet build to compile
   a. This will compile the code and build a .exe file that can be executed
2. Navigate to the folder containing the .exe file and run, passing arguments
3. The code will run and generate two .txt files, one for time signals and another for the output values



```
PS C:\gitrepos\Signal_Generator\SignalGenerator\bin\Debug\net9.0> .\SignalGenerator.exe sawtooth 10 2 2 100
Generating Sawtooth Signal...
Done!
```

## Using Python to Display Waveform

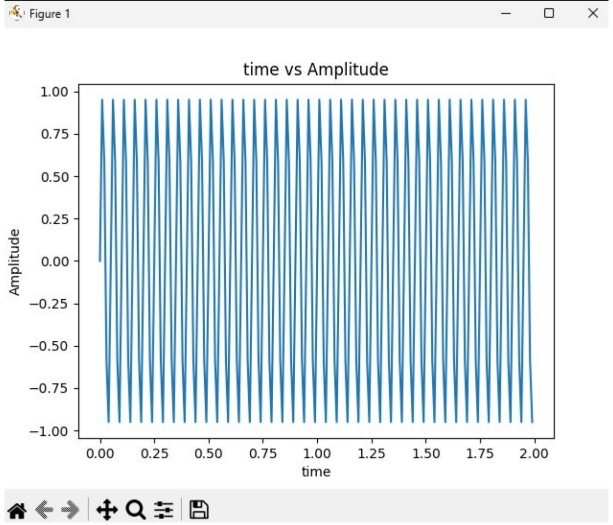
- Used Python's matplotlib and numpy libraries to read the x and y values from time and output .txt files generated from Program.cs to display a waveform.
- This approach was chosen because the current C# program is a simple console application.
- Next Steps:
  - Expand the C# programs to generate more complex types of signals(ie. Audio and Radio Signals)
  - Explore and implement native tools for graphing within C# itself



```python
#create a plot based on two text files
import matplotlib.pyplot as plt
import numpy as np

#read the data from the files
data1 = np.loadtxt('time.txt')
data2 = np.loadtxt('output.txt')

#plot the data
plt.plot(data1, data2)
plt.xlabel('time')
plt.ylabel('Amplitude')
plt.title('time vs Amplitude')
plt.show()
```



Github Link

Thank You!