A close up of a sign

Description automatically generated

Programming Project - Unit 3

Block#

Lewis Miller | Candidate Number: 2086

Computer Science | H446-03 Programming Project

Bournemouth School | Centre Number: 55119

# CONTENTS

CONTENTS 1

ANALYSIS 3

Identifying The Problem 3

Identifying Why The Solution Should Be Approached Through Computational Means 3

Identifying Stakeholders And Their Needs 4

Stakeholder 1: Mr S. Perry – Computer Science Teacher: 5

Stakeholder 2: Mr J. Fowler– Computer Science Student: 5

Overall Stakeholder Requirements 6

Similar Products And Existing Solutions 7

MicroBit {and IDE} 7

Microsoft Visual Studio Code 23

Scratch {and IDE} 39

Requirements Specification 52

Essential Features 52

Requirements Specification For The Program 54

System Software And Hardware Requirements 57

Limitations Of The Program 58

GUI based app development 58

Dedicated ability to share programs online 58

Support for multiple languages 58

Measureable Success Criteria 59

DESIGN 61

The Components Of The Problem 61

Programming Techniques 61

Saving and Opening Files 62

Design and Development of The User’s Own Program 62

Output of The Program 62

Debugging 62

Designing a solution 62

Language 62

Something Else // TODO 63

DEVELOPMENT 64

Overview 64

The UI 64

The Debugger 64

The Designer 71

The Blocks 71

BaseBlock 71

Chains 71

Functionality 71

Running A Program 71

Saving Your Work 71

REFERENCES 72

# ANALYSIS

## Identifying The Problem

In a society that is rapidly advancing through the technological era, it is becoming more and more imperative for younger students to gain an understanding of not only how computers work, but also how they can be used and programmed. As such I will be developing a program designed to help younger students not only to become engaged in programming but also to develop thinking skills that can be helpful for completing tasks later in life.

Because the problem is primarily focused on how school students in lower years interact with and understand computers and computational methods, a suitable approach could prioritise a “hands-on” approach to the subject, aiming to encourage pupils to gain an understanding through trial and error. This would allow them to gain a deeper insight into how real computer programs run in real life. To do this, students could be provided with their own IDE or similar program, which itself would have to be programmed. These aspects make the program amenable to a computational approach as providing a rich learning environment integrated with the ability to produce a working example of software is often difficult to do with plain pen and paper.

In order to help students to engage in programming, I will develop a program that provides an interesting and interactive way of programming. The implementation I intend on using will prevent simple syntax errors and such from occurring by using a block-based interface. Each block will have a specific functionality, and they will connect in such a way that it will be easy to produce a functioning program. The first instance of the program will rely on a command-line interface, but in future I would like to add a graphical element.

## Identifying Why The Solution Should Be Approached Through Computational Means

|  |  |
| --- | --- |
| Feature |  |
| User Interface | Designing code by hand is a time-consuming and arduous procedure. By abstracting it into a program based on code “blocks”, the design of code becomes simpler and more intuitive. This creates the opportunity to make coding more accessible to a wider range of ages and abilities. |
| Code Generation | Generating functional code manually is particularly slow and difficult. You would have to write the binary out manually and feed it into the computer’s RAM. Nowadays we use programs to do this for us, and we write the code in a language that resembles English. This English-like language works for most experienced developers but is hard to learn for people who have never used it before. By providing a visual aid to show how the code joins together, computational techniques can be improved with little need for the use of programming languages. |
| Teaching Assistance | Different people learn in different ways. The program could provide an explanation of what each component does and how to use it when needed, saving time for the teacher who would otherwise have to explain it themselves. |
| File Handling | To save your work and restore it on a PC, the solution would have to be able to access and edit the file system, otherwise all data could be lost when they exit the application. |
| Input | The user would have to input how they want the code to be designed, and when they want it to run. Designing the code using a GUI allows the changes to the code to be hidden from the user, minimising the overall complexity they are affected by. |
| Processing | The problem requires that code can be generated by the user and then run by the program. The act of designing the code would require many processes to ensure that it builds correctly when needed. This running program would have to be able to take the user’s input, process it, and then output a result. |
| Outputs | The program would output a fully functional CLI based application for the user. This is more simple than compiling code manually, as the program does it all for you. This is suitable for a computational approach as the output must be run on a computer. |
| Feedback | Errors are often hard to find without assistance. A computational method could provide a debugging tool or assistant to help isolate and fix errors in code by listening to the outputted program for errors, then show where they occur when the user returns to the main application. |

## Identifying Stakeholders And Their Needs

From what I can tell, there are two primary groups that will use this software: Computing Teachers and their Students. As such the program will be an educational assistant and should aim to help teaching students about computing.

* Teachers
  + Effective teaching aid
  + Easy to show a class how to use
  + Engages students with a variety of features
* Students
  + Easy to learn, understand and use
  + Allows for developmental experimentation with computational methods

### Stakeholder 1: Mr S. Perry – Computer Science Teacher:

Mr Perry is a Computer Science teacher at Bournemouth School. The software must suit his needs as he will be using it regularly in class, and any disruptions caused by it will be immediately noticeable. Mr Perry also wants the program to help show students how high-level languages function.

#### Requirements

|  |  |
| --- | --- |
| Requirement | Justification |
| Stability | Any crashes will significantly impact lesson time, so the program should not crash during use. |
| Simple to explain | Mr. Perry does not want to waste lesson time showing students how to use it, wasting time that could be spent better on seemingly minor things that are difficult to intuit when they should be easy. |
| Provides understanding of computing practices | Resources that allow computing practices to be learnt are greatly valued as they do not tie the student down to a single way of programming. |
| Can show students the equivalent code in a more advanced language | Being able to recognise the way statements flow together and connect in other languages will help build up programming skills, making students better programmers overall |

#### How They Will Use the Program

Mr Perry intends on using the program to aid his teaching of computer science. He intends on providing it to some of his students who are having a harder time understanding some concepts.

#### Why Is It Suitable for Their Needs?

Allowing students access to a basic program that only showcases limited concepts, they are not exposed to more than they can understand at once. By giving them basic functionality, they spend more time learning the simple concepts than fiddling with GUIs and objects. This ultimately results in better understanding of the basic skills

### Stakeholder 2: Mr J. Fowler– Computer Science Student:

Jack {age 14} is an aspiring computer science student at Bournemouth School. He is particularly interested in the programming aspect of computing but needs a way to better understand programming techniques. He has a hard time understanding several fundamental concepts and is looking to improve his knowledge on them.

#### Requirements

|  |  |
| --- | --- |
| Requirement | Justification |
| Simple to use | Students shouldn’t be spending large amounts of time learning to use the program instead of using it. |
| “Engaging enough to stop me from falling asleep” | Wasted lesson time is bad for education; if the student is distracted, they aren’t learning. |
| “A large breadth of functions” | Students should have the opportunity to expand on their learning in inventive ways. |
| “Having a quick way to check that my program works” | Immediate access debugging allows the user to rapidly isolate and correct errors, increasing development rates and improving debugging skills. |
| “It shouldn’t break when I’m using it” | “I don’t want to lose all of my work for something out of my control” |
| “I need to learn something from using it” | “What’s the point in doing something in school if it doesn’t teach me anything?” |

#### How They Will Use the Program

Jack will use this program to help build up his programming skills before he is ready to use a fully-fledged language such as C# or Java. He is likely to use it both at home and in school,

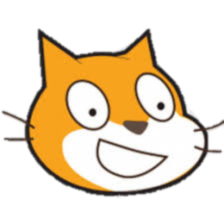
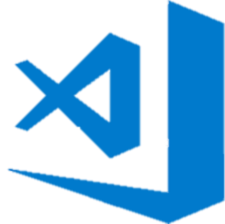
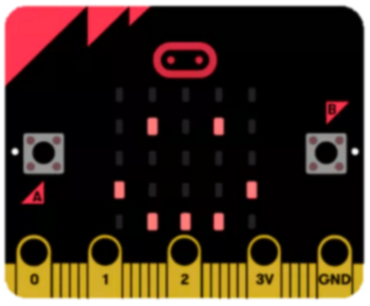
#### Why is it suitable for their needs?

By providing a simple language that demonstrates the simple concepts like Input 🡪 Process 🡪 Output or basic Programming Logic, as well as an IDE for it, Jack can be shown how to make his own programs. He can then use the language to experiment with his own skills and improve until he reaches a level compatible with high-level languages such as C#, VB or JavaScript.

## Overall Stakeholder Requirements

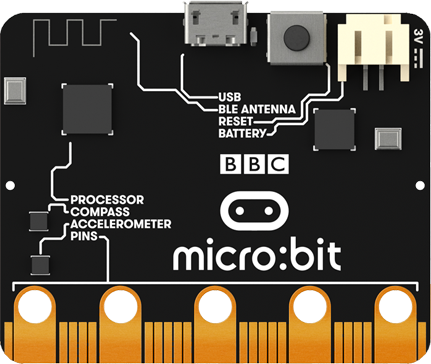
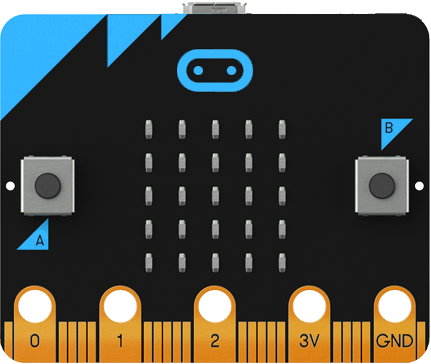
* Stable
* Simple to explain, understand and use
* Provides understanding of computing practices
* Can show the code in a full language
* Engages the user
* A large variety of functionality
* Can help with debugging
* Can be installed both at school and at home

## Similar Products And Existing Solutions

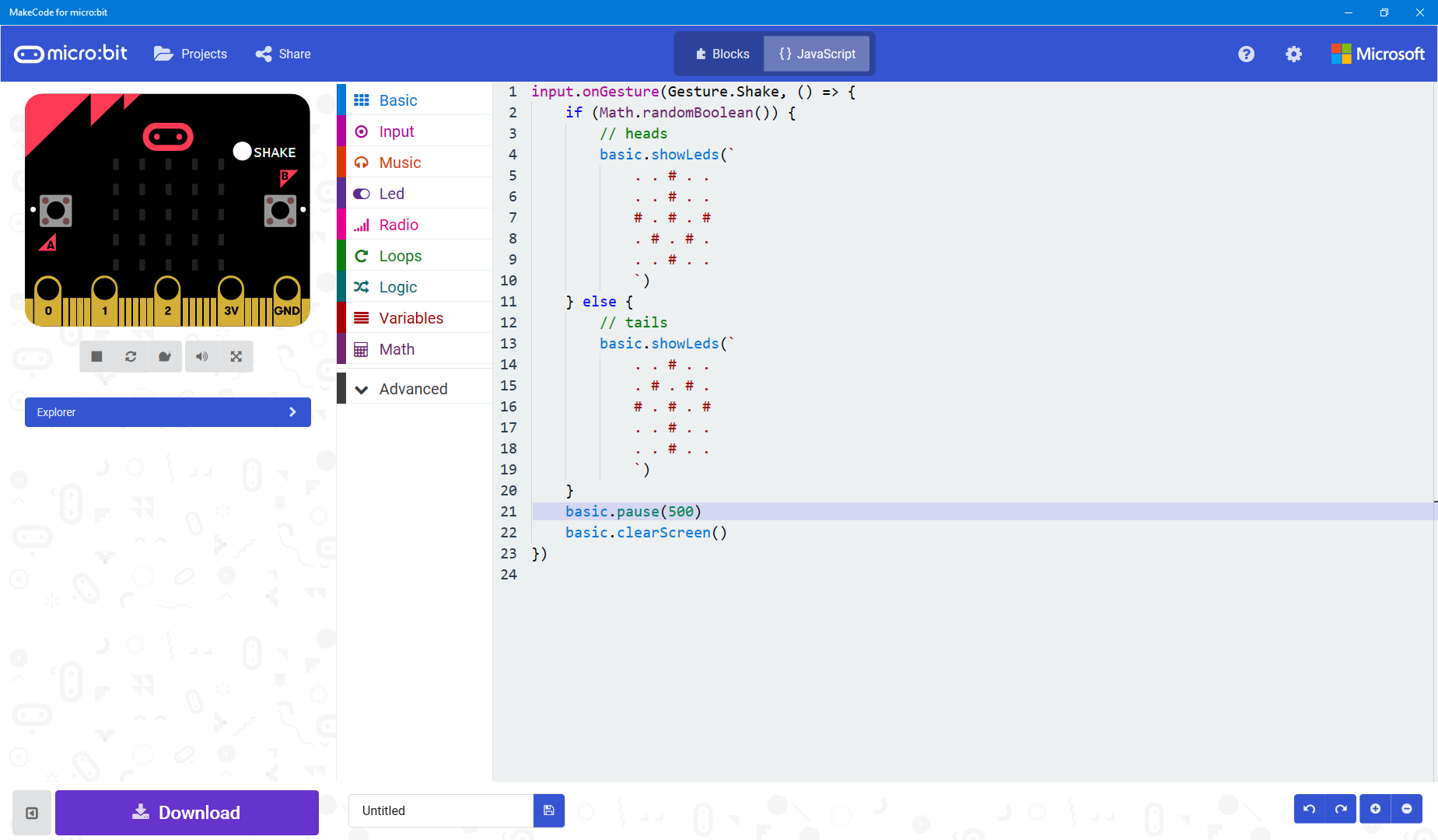


### MicroBit {and IDE}

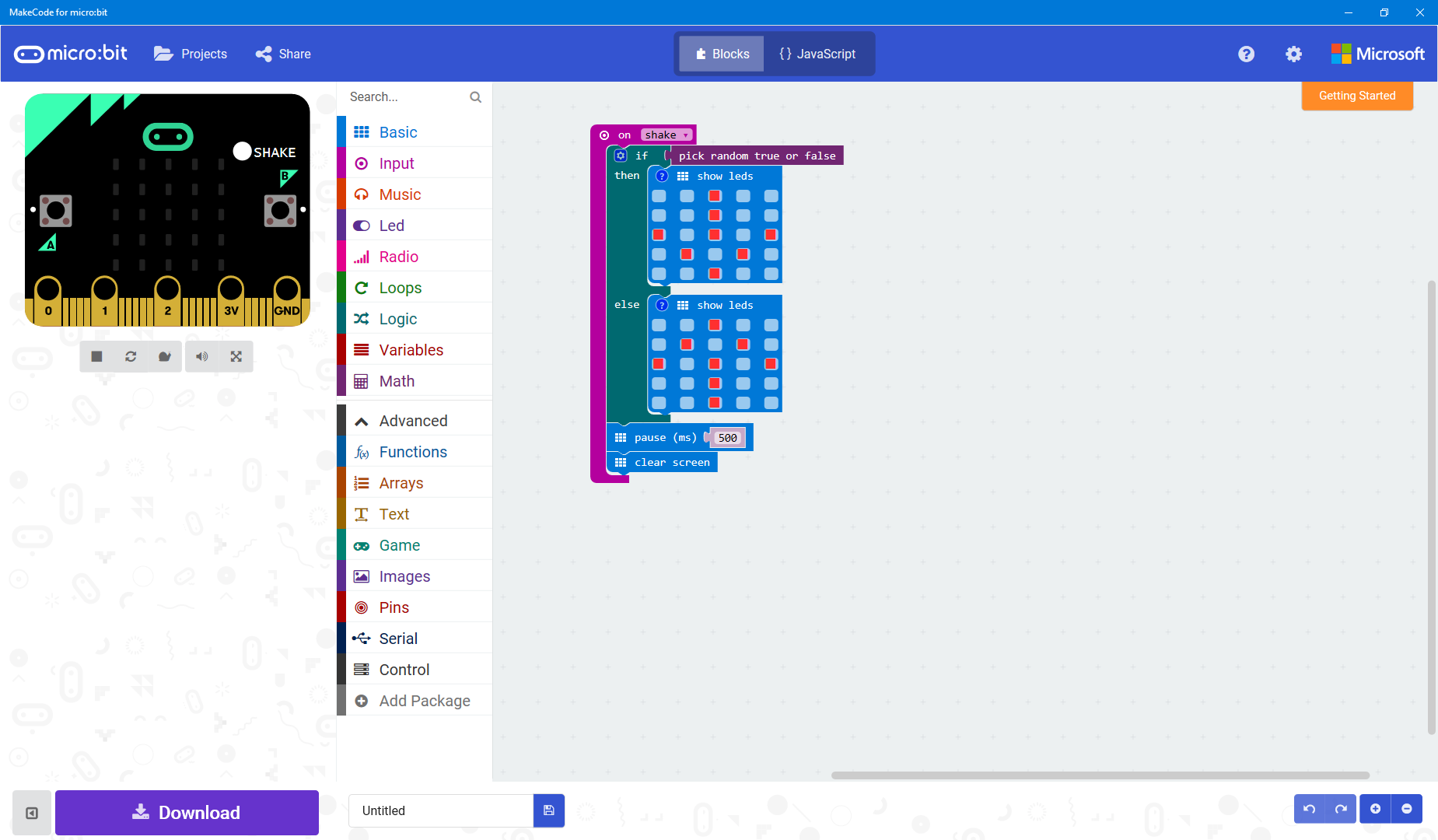
The MicroBit, produced by the MicroBit Educational Foundation {MEF}, is a piece of educational hardware designed to get young students to be interested in programming. The MicroBit itself is a handheld electronic device that can be programmed using Python or JavaScript {JS} using an online editor or an offline editor {I will be using the offline editor}.



The MicroBit hardware has two PTM switches and a 25-LED screen on the front, 20 pin connections at the bottom, a micro-USB port, a Bluetooth/Radio antenna, a compass, an accelerometer and a surprisingly powerful processor.

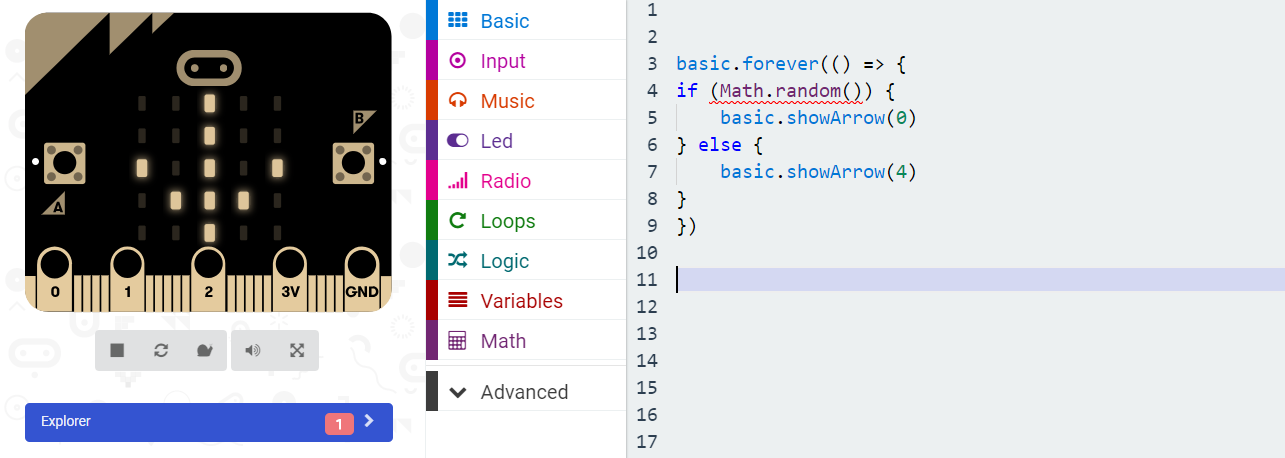


The MicroBit JS code editor {MakeCode for MicroBit Windows App}

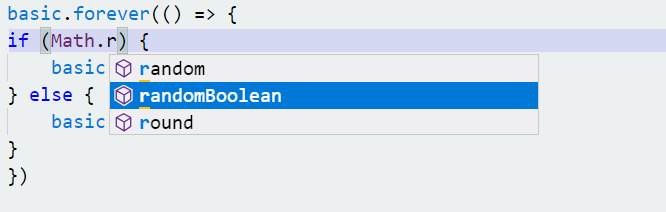


The MicroBit block editor for the same code

The JS editor allows you to edit MicroBit code in both a block-based format and a text-based format. This makes it possible to switch between the two, similar to how Small Basic allows its user to migrate to Visual Basic. Unlike the Small Basic migrator, however, the JS editor allows you to switch between text and blocks seamlessly. The JS editor also displays a MicroBit on the top left of the screen, allowing you to test your code without having to wait to download it to a MicroBit.

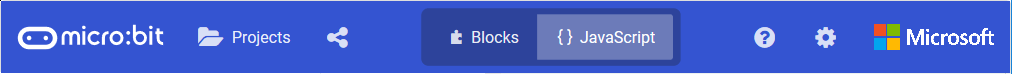


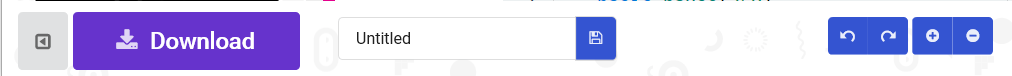
If your code contains an error, the MicroBit will appear greyed-out and a notification informing you where your code doesn’t work will be shown. The MicroBit also appears greyed-out when the code is compiling



This editor also makes use of IntelliSense, as Microsoft helped produce the software.

#### Toolbars





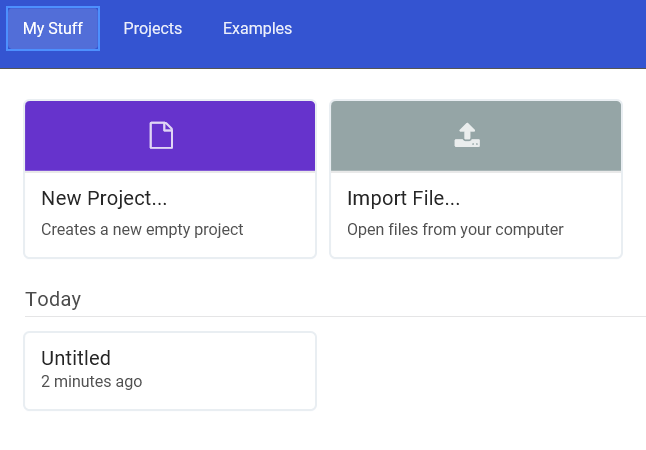
The MicroBit top and bottom toolbars, respectively

The Top Toolbar appears above the editor workspace. It allows you to:

* Switch between the JavaScript {JS} editor and the Block editor,
* Create, open and share projects
* Access help documentation
* View and change settings

The ability to switch between the two editors allows seamless transitions from between the two editor modes, discussed further [HERE](#_Dual-Language_Support).

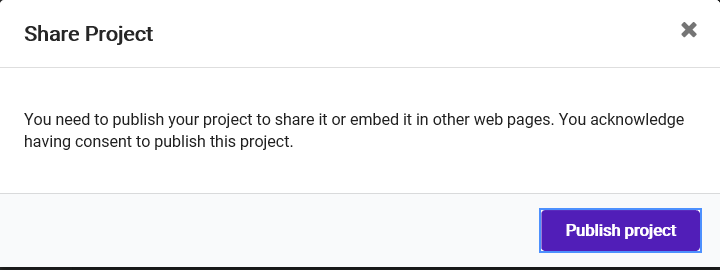
##### Open



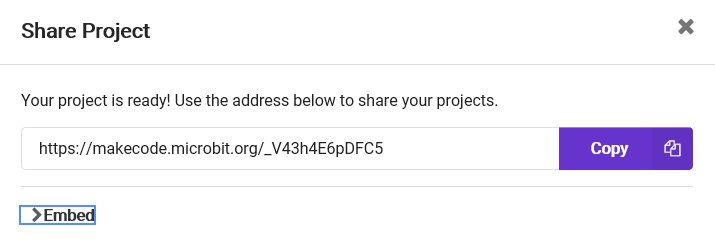
The “Projects” menu dialogue

The “Projects” button opens a menu, where you are prompted to import a file, create a new project, open a recently opened project or view projects others have made as a starting point for your own application.

##### Share



The editor asks before uploading your data



A random link is generated to the program



The option to embed your code in HTML is also provided through Copy/Paste-able sections that provide the code already formatted for HTML use

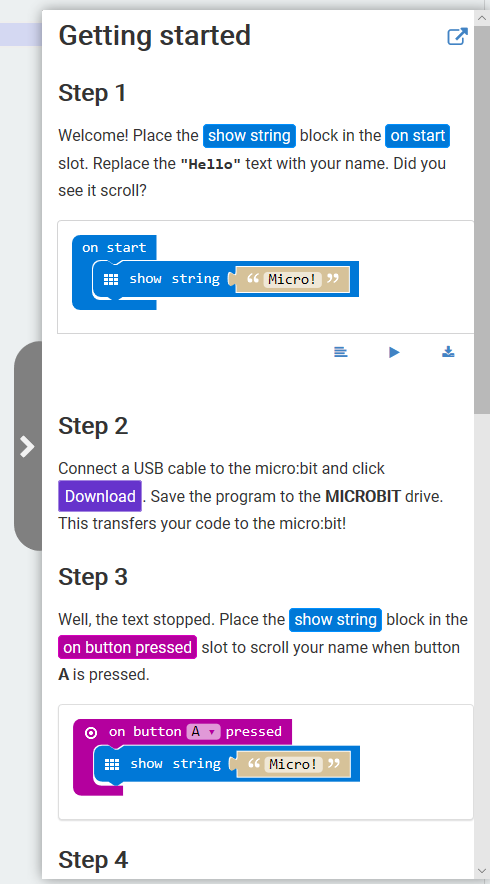
This uploads the project to MicroBit’s own site, where it can then be accessed using a link generated by their servers.

##### Help



Clicking the (?) icon opens the Help menu

The help menu provides links to additional resources. Many of the items open the side menu, a panel on the right of the screen that shows help information.



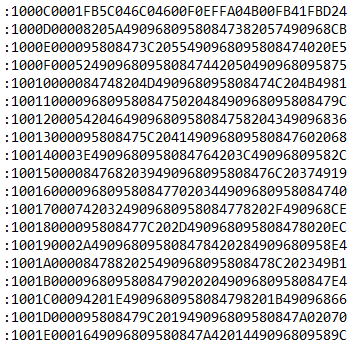
* Support
  + Provides a link to MicroBit’s own support [website](https://support.microbit.org/support/home)
* Getting Started
  + Opens the side menu to the Getting Started page
* Projects
  + Opens the side menu and suggests some projects to look at for inexperienced users
* Reference
  + Opens the side menu to a page that explains the purpose of the different types of commands available
* Blocks
  + Opens the side menu to explain how the blocks editor works {similar to Scratch}
* JavaScript
  + Opens the side menu to explain basic JS principles and ideas
* Hardware
  + Opens the side menu to explain the different parts of the MicroBit itself, and how they can be used
* Buy
  + Provides a link to [MicroBit’s website](https://microbit.org/resellers/) where users can purchase a device
  + As the MicroBit is a piece of external hardware, it must be purchased for use in person

##### Save



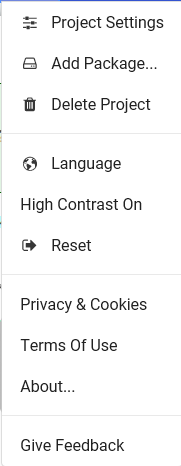
Downloading the project is not the same as saving it

You can save the project at any time by clicking the “Save” button on the bottom toolbar. This will generate a “\*.hex” file, which can then be downloaded to a MicroBit for use {Clicking download will attempt to find a connected MicroBit and automatically download the “\*.hex” file to it}. The “\*.hex” file is actually a compiled hexadecimal file and contains all the necessary information to run the program and rebuild the editor environment when reopened on a different computer.



An excerpt from a project “\*.hex” file

#### Settings

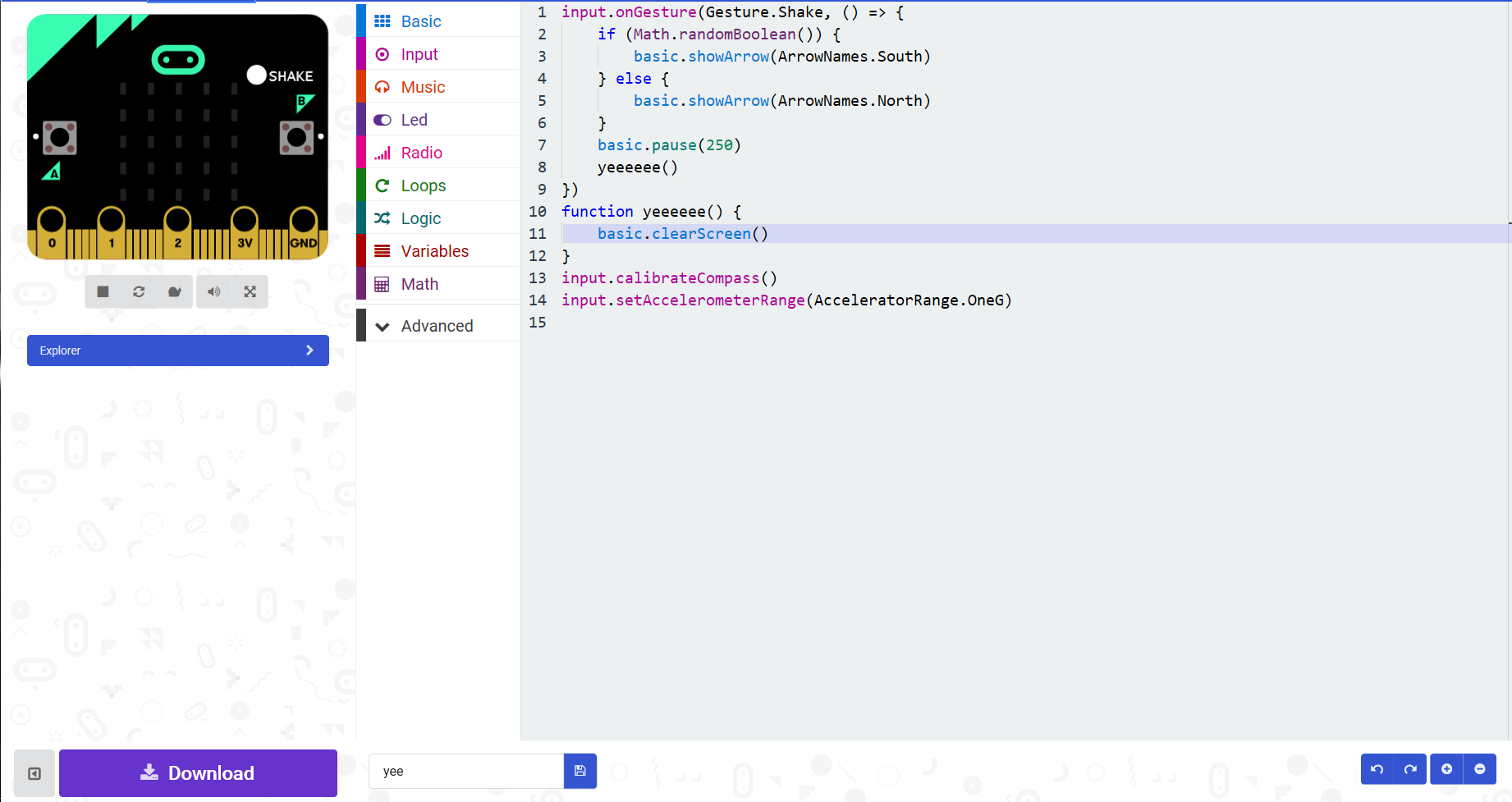


Clicking the gear icon opens this menu

The settings menu allows the user to edit project and global settings, as well as view related legal information about the program

* Project settings
  + Allows the user to view settings applicable to your project, such as the name.
    - The editor does allow you to edit your own settings file, but this is not recommended
* Add package
  + Allows the user to add packages to the project to augment its functionality, such as Bluetooth or Device compatibility
* Delete project
  + Allows the user to completely delete the project and related files
* Language
  + Allows the user to change the display language (Defaults to the System language if available)
* High Contrast
  + Allows the user to switch between a light and a dark editor theme to help people with vision impairments
* Reset
  + Restores any changed settings to their default values
* Privacy and cookies / Terms of use / About
  + Provide access to legal information about the program
* Give feedback
  + Allows the user to report errors and suggest improvements to the MicroBit team

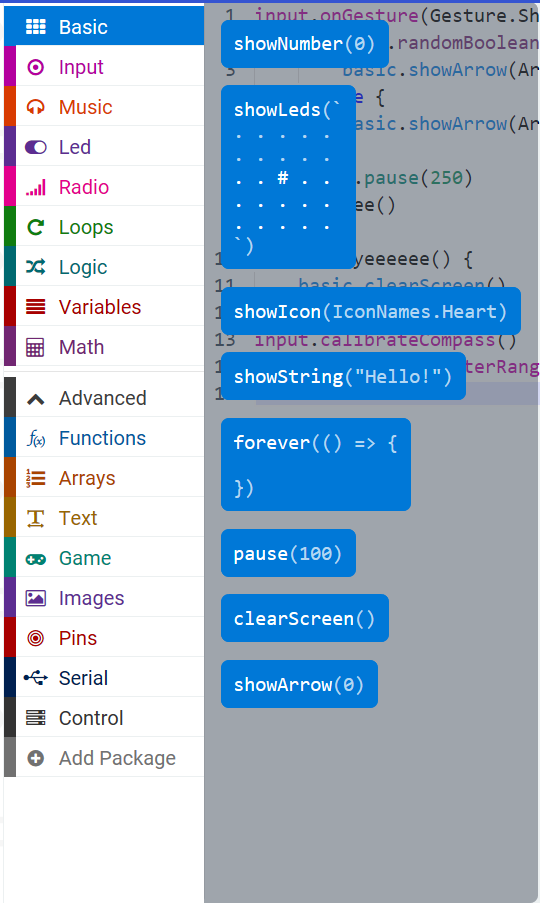
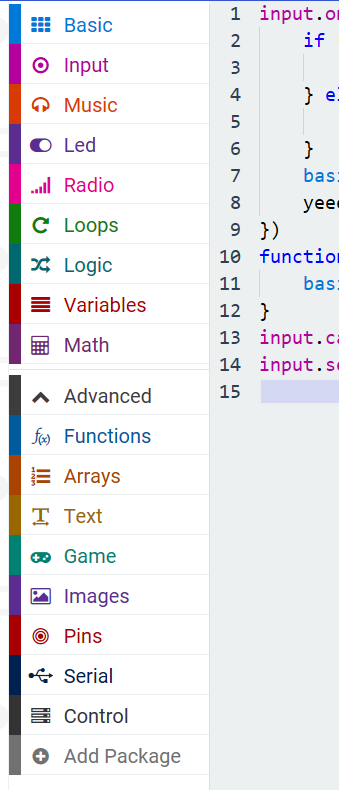
#### Workspace



The MicroBit JS editor workspace features {Left 🡪 Right, Top 🡪 Bottom}:

1. A real-time MicroBit example with buttons to allow the following functions:
   1. Start / Stop execution
   2. Reload code
   3. Slow down execution
   4. Mute audio
   5. Fill screen with display
2. Toolbox with code block categorised by function
3. The main text editor space with text highlighting and error detection
4. Hierarchy explorer allowing you to browse the different parts of the project
5. Bottom toolbar with buttons that allow the user to
   1. Show / Hide the example display
   2. Download / Save and Name your work
   3. Undo / Redo your last action

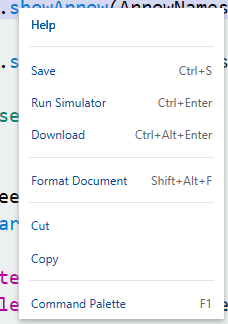
##### Toolbox



The toolbox opens an overlay when clicked, inserts a code snippet when an item is clicked

In this editor, the toolbox works like a collection of code snippets, each categorised based on what they do and how they are used. When clicked, the code snippet is inserted at the point the cursor is currently at. This is very useful as it means that users do not have to memorise the library names that are associated with JavaScript and can simply have the snippets inserted for them.

##### Right-Click Context Menu

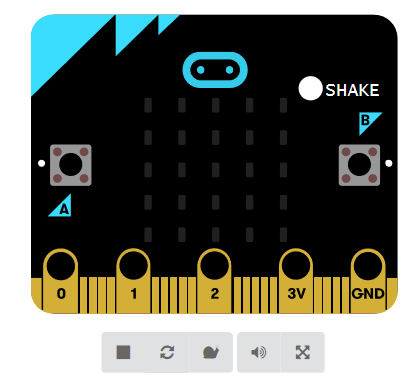


The context menu when

Right clicking an item of text brings up the context menu. From here you can perform actions such as saving, running and downloading the program. You can also “Format Document”, which allows you to clean up your code easily. Cut and Copy can also be accessed from here.

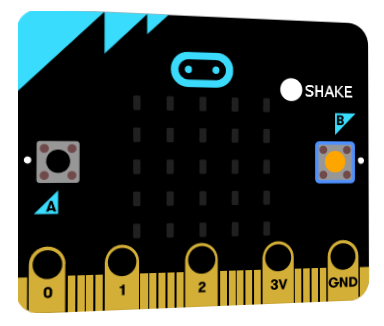
The command palette option is discussed further later in this document {[HERE](#_Command_Palette)}.

##### Real Time Execution



MicroBit Preview

When a change is made, the MicroBit preview reloads with the new code. This allows code to be tested immediately after changes are made. All the inputs on the MicroBit are shown, and can be clicked to activate them



The currently clicked input is highlighted, and the MicroBit rotates slightly to face the mouse

This allows bugs to be found quickly and easily without having to download the code to a MicroBit to test it, saving valuable time. It also means that the user doesn’t have to possess a MicroBit to use one, but it is recommended.



The buttons beneath the MicroBit Preview

These buttons allow you to control the way the MicroBit preview executes code. The Stop button stops code execution at the current line / block and can be clicked again to resume it. The Reload icon will force the preview to recompile and restart the code. The “Snail” causes the preview to enter a Slow Mode, where each line is executed at a rate that allows execution to be watched (about 250ms between each line).



The line currently being executed is highlighted when Slow Mode is active, and the time taken for each line to be run also increases

##### Dual-Language Support



These buttons allow you to switch between the Block and JS editors at will

The editor supports both JavaScript and the Scratch-Like language used to program a MicroBit. This functionality can be understood better by using the file hierarchy explorer beneath the MicroBit Preview.



The explorer bar: collapsed



The explorer bar: opened to view the files in the project

The program keeps two copies of the file, one that controls the block editor and one that controls the JS editor {Note that the JS editor uses TypeScript, a strongly typed variant of the dynamically typed JS}. The “main.blocks” file contains all the information needed to load the blocks editor and can’t be viewed outside of the editor. The “main.ts” is a plaintext file and contains exactly what is written in the JS editor. The “README.md” can be edited to provide a built-in description for the project. “pxt.json” is equivalent to the settings file and cannot be edited directly. Opening “pxt.json” opens the same window as clicking “Project Settings” in the Gear Menu. The dark blue folders represent source code and modules.

The Dual-Language feature works because the code is recompiled after any changes, and when the editor view is changed from one file to another. The currently opened file is parsed and converted into both executable code and the other data format, such as: JS 🡪 Blocks and Executable Code; Blocks 🡪 JS and Executable Code.

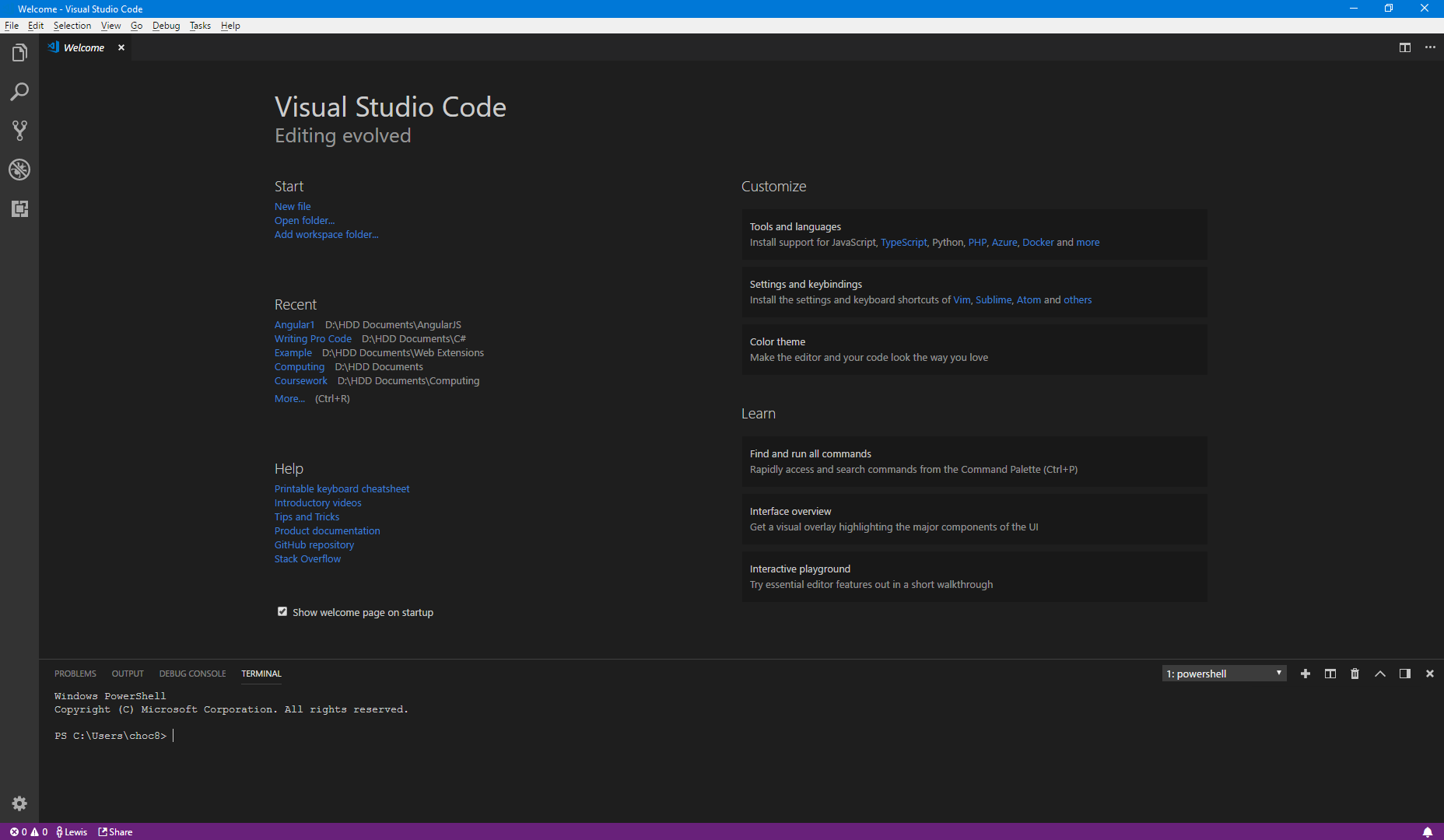
#### Summary of Features

|  |  |  |
| --- | --- | --- |
| Feature | Will I Keep It | Why? |
| Toolbars | Yes | Having direct access to functions is a useful feature for users, instead of having to learn and use a large number of keyboard shortcuts straight away |
| Help menu | Yes | Having easily accessed help is a useful feature for any user to have. A description of each block and what it does would aid users who are unsure of how to use them |
| Undo/Redo keys | No | While useful, keeping track of the user’s last actions is likely to prove to be quite difficult, and goes beyond the scope of my application |
| Edit settings | Yes | Should there be any meaningful changes they can make, users should be able to edit settings to improve and edit their environment to their choosing |
| Context menu | Yes | Context menus are useful because they provide immediate use and functionality without cluttering up the UI |
| Dual Language support | No {Mostly} | Editing and parsing in two languages would be much harder to code than just one and would likely cause me to be unable to finish the program. The “{Mostly}” aspect is because I would quite like it to be possible to show the user the equivalent code in C#, just not edit it |
| Real time execution | No {Probably} | This depends very much on how I implement my solution. Should I try to compile the code into its own application, then this feature would be a lot harder to achieve than if I were to run the code as part of the main program |
| Toolbox | Yes | Being able to access all the pieces that build up the program is a necessity when they cannot be typed in manually |
| Projects menu | Yes | Being able to view all your latest projects and open them immediately is a useful feature as it allows quick access to projects instead of having to browse for them yourself |
| Sharing and web hosting | No | This would require a server, website and another set of algorithms in several languages, which is far beyond the scope of my project |
| Legal information | No | As the application is unlikely to store personal user data / be used in industry / upload information to the internet, it is unlikely that it will need to contain any legal information |
| Copy / Paste | Yes | Being able to take code that you have already written and reuse it saves on development time |
| Package management | No | Being able to write extensions and additional packages would require technical skills beyond those of a learner programmer |
| Feedback | No | While a useful feature for reporting bugs, as my stakeholders will be able to contact me directly, incorporating such a feature is unnecessary |
| Zoom In/Out keys | Yes | This allows people with vision impairments to use the program better as they can zoom in to gain a better focus on what they are looking at |

The MicroBit IDE is excellent for giving instant feedback about your code, as it can highlight errors and suggest code snippets as well as recompile your code as you work on it. However, many of its features are more advanced than what my project requires

### Microsoft Visual Studio Code

Visual Studio Code {VSC} is a professional level IDE by Microsoft and is the IDE I use to develop most of my non- “.NET” applications. It is designed to integrate seamlessly with Microsoft Team Services or Git, can support many programming languages, and can have additional features added using the NuGet package manager. This all comes together to create an excellent IDE that can be used for many purposes.



Visual Studio Code when started without a root directory. My preference is the dark theme, but the editor’s appearance is fully customisable

Visual Studio Code has all the expected features of an IDE. It is capable of loading and saving most plaintext files, as well as viewing many common other file formats such as GIF, JPEG or PNG. It can even analyse your code and suggest minor improvements such as extracting an item into a class or method or using an alternate data type to better suit your needs. VSC also includes the IntelliSense feature present in the MicroBit JS editor, but instead of focusing on just one language it can support many different ones. VSC also provides many debugging tools to assist development.

An important aspect of VSC is that it can be opened within any directory on your PC and display all the child directories and files within them by right-clicking in/on a folder and selecting “Open with Code” or using the terminal command “code . (or a folder name)”. Unlike its counterpart Visual Studio {which I use for all “.NET” programming”}, VSC is relatively resource-light, making it suitable for use on even the most basic of machines.

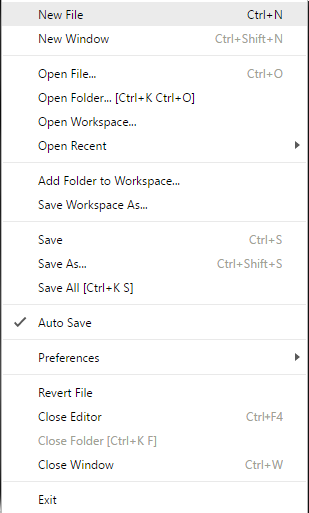
#### Menu Bar



The VSC menu bar

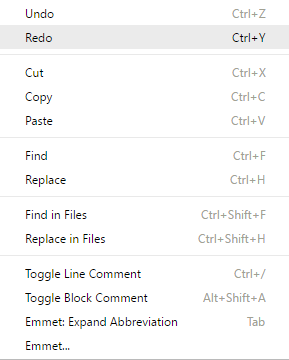
VSC’s menu bar is where much of VSC’s functionality can be accessed from. Each menu item categorises the commands available to the user and makes it easy to find the command they need. If a command within the sub-menus has a keyboard shortcut associated to it, such as Save All {CTRL + K, Ctrl + S}, that shortcut is displayed next to the command.

##### File



This menu displays the basic functions associated with the creation and loading of files and folders. It also displays different saving options such as Save As, Save, and Save All {Saves all open tabs}. This menu contains links to VSC’s many settings, under the Preferences sub-menu.

##### Edit

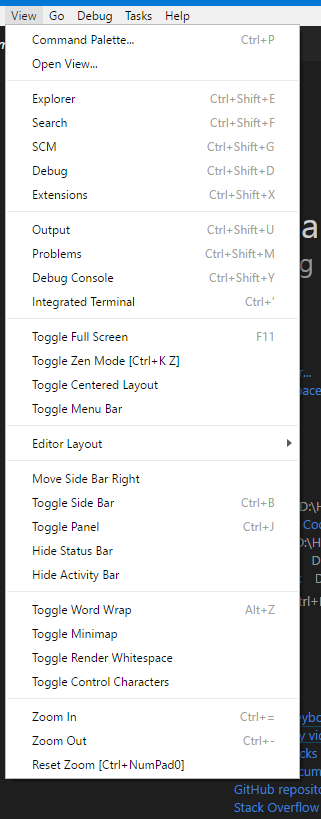


The Edit menu

The Edit menu contains commands such as Undo/Redo, Cut/Copy/Paste, Find/Replace and Comment Line. These commands are used most commonly as keyboard shortcuts however, so the menu is mostly there to help remind the user of them.

[Emmet](https://emmet.io/), the tool shown at the bottom of the menu, is used in HTML for increasing the rate of code generation. It uses abbreviations to help developers not have to type out the full HTML item name or type repeated items such as lists. With some minor configurations, Emmet can be used in many other languages, but there may be some temperamental aspects to it. There is a lot more to Emmet than this, all of which can be found at <https://emmet.io/>.

##### View



The View menu contains some of the most important items for VSC users. This menu allows for the rapid changing of the editor environment to help maximise productivity. The first pair of items open the command palette, the first one leaves an empty palette, the second with the keyword “view” inputted for you. The second set allow you to change the chosen Sidebar menu, explained in the [Editor Environment](#_Sidebar) section. The third set allow you to change what is displayed at the bottom of the window {I normally have mine set to the terminal for ease of access}, again explained further in the Editor Environment section. The next three sets control how the editor is displayed and allow you to toggle the purple status bar at the bottom of the window, change how text is displayed, and change the zoom level of the editor and text.



The status bar icons



Whitespace can be rendered with a pale dot, while tabs are represented with a dash

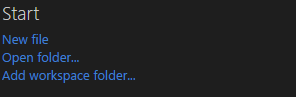
#### Welcome Page



My VSC Welcome page on 06/08/2018

The welcome page {by default} opens every time you start a new instance of VSC. This page shows your recent projects, as well as allows you to open them or other files if they are not listed. It also provides useful links to create new files and folders, view help pages, and edit some common settings.

##### Section – Start



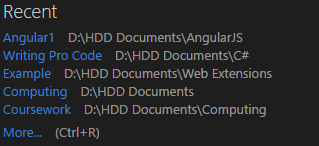
The Start section

This section shows the most basic options for creating a new file or project. It provides the options to:

* Create a new file
* Open VSC to a folder
* Create a new project folder

These are the most common start points for projects, so having them on the start page is going to be useful to many developers.

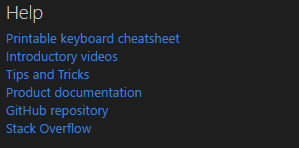
##### Section – Recent Items



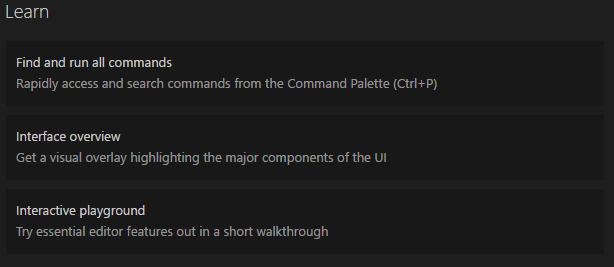
My Recent Section – Note the

This section of the Welcome Page shows recently opened files and folders. By clicking on these you can open VSC to the selected folder and continue working where you left off. VSC saves your workspace configuration when you exit, so you can resume work seamlessly even after a long period of time without working on the file or project.

##### Section – Help and Learn



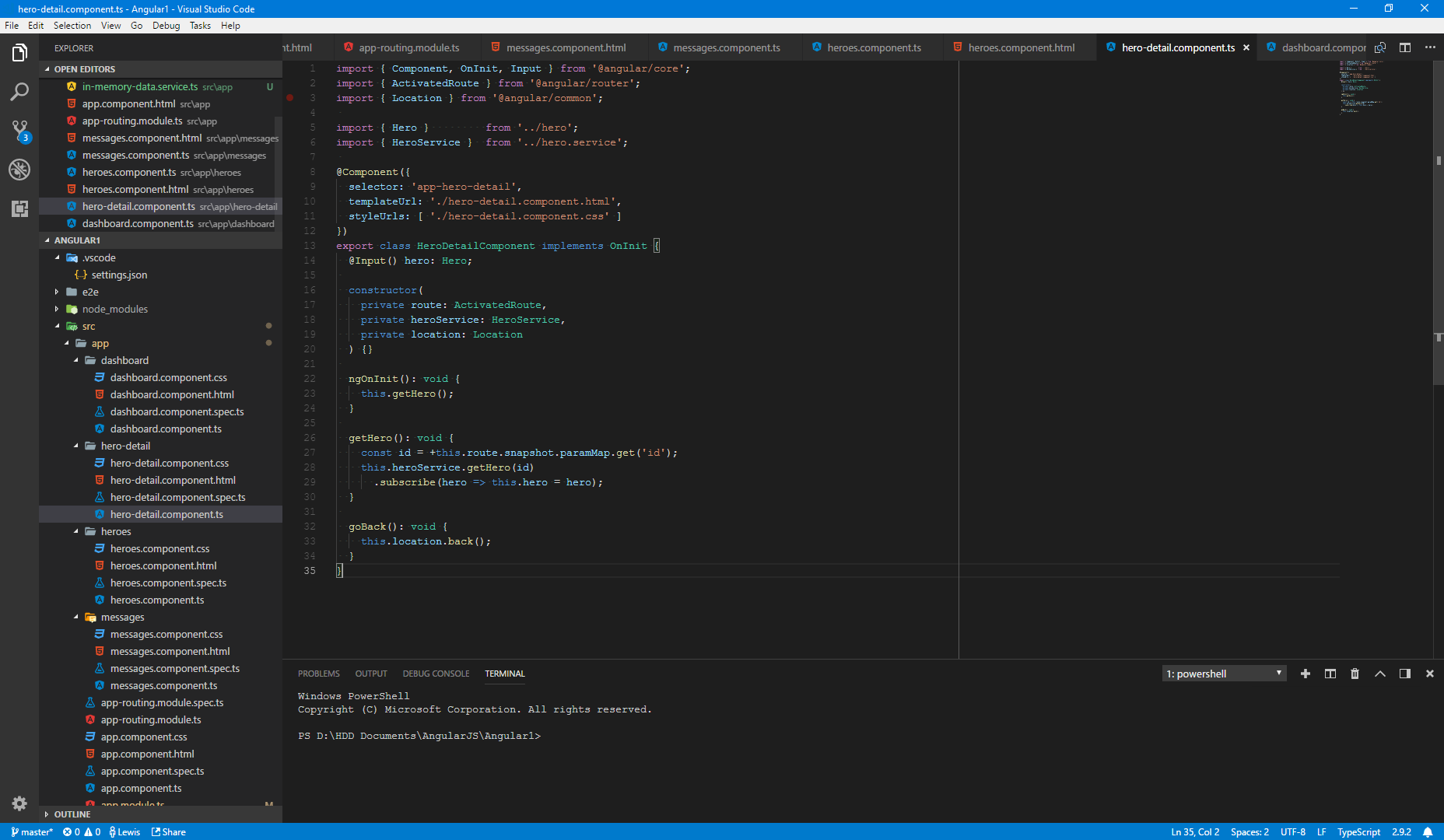
The Help Section



The Learn Section

These two elements provide an easy way to learn more about VSC and what functions and features it has. The hyperlinks in help

#### Editor Environment



VSC within a folder and a set of files opened

##### Text Editor

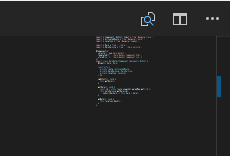


The text editor with multiple tabs open

When editing programs, the quality of the text editor is always one of the most important things to consider. The functions provided by VSC’s editor are numerous. It includes the same IntelliSense that runs in the MicroBit editor and allows you to jump to errors in your code by highlighting them in the scrollbar. The editor also displays a small view of your code that appears in the top right of the window to give a brief overview of your code.

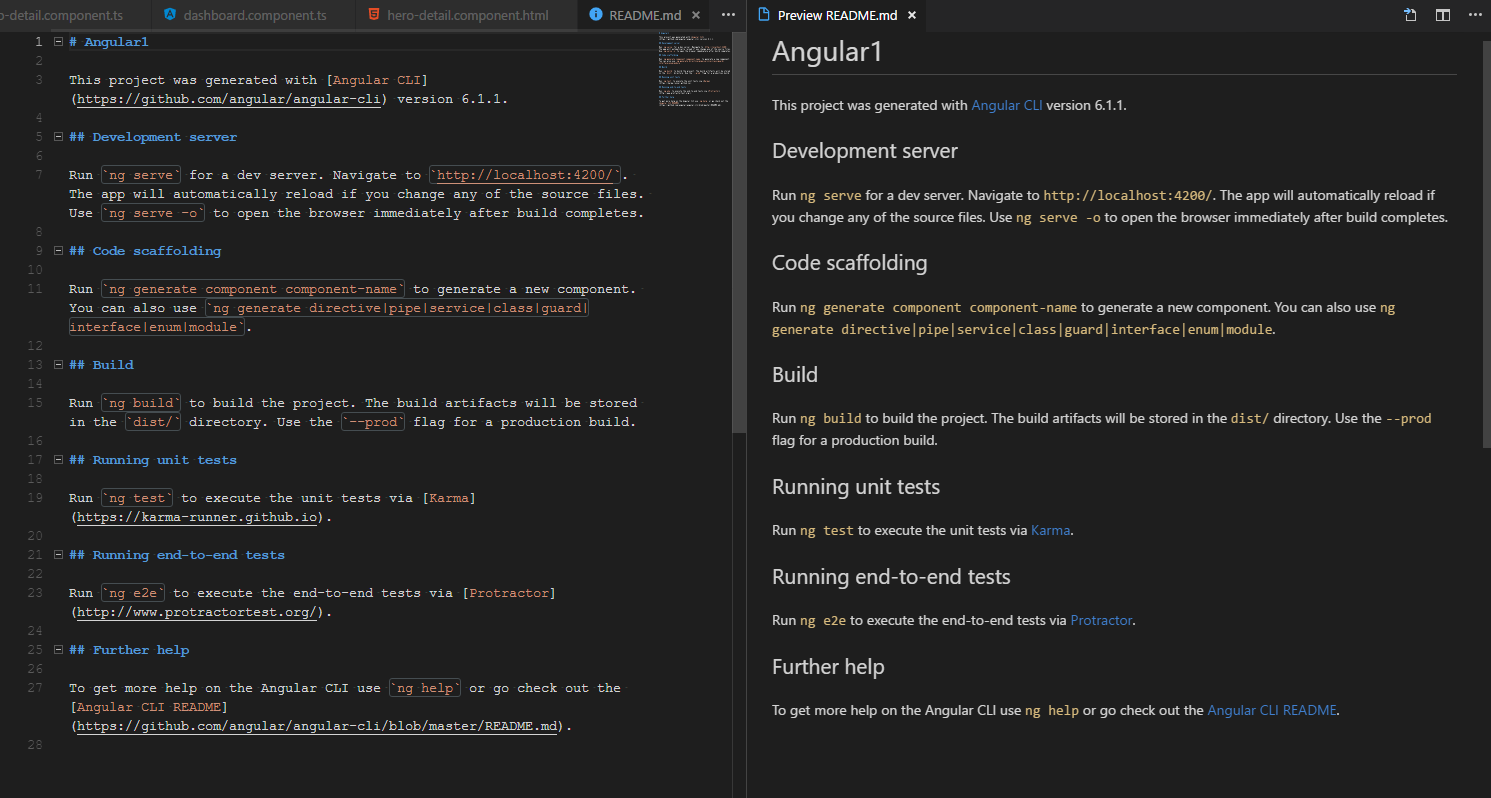


The red bars on the scrollbar show where errors have been detected



The code’s preview. The small blue bar is on the scrollbar, and shows where the Git tool has detected a change

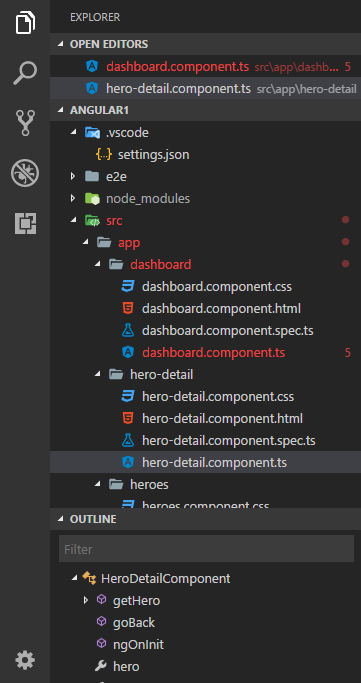
VSC’s editor can also display a preview of certain files, such as MarkDown files {.md}, allowing you to see what effect your changes may have upon the document.



A MarkDown preview in VSC

Visual Studio Code’s editor allows you to work on multiple documents at a time using tabs at the top of the page. Clicking a tab will open the relevant document and hide the previous one from view.

##### Sidebar



The Sidebar in Explorer view

The sidebar is a very important part of VSC. It has 5 main modes to be in.

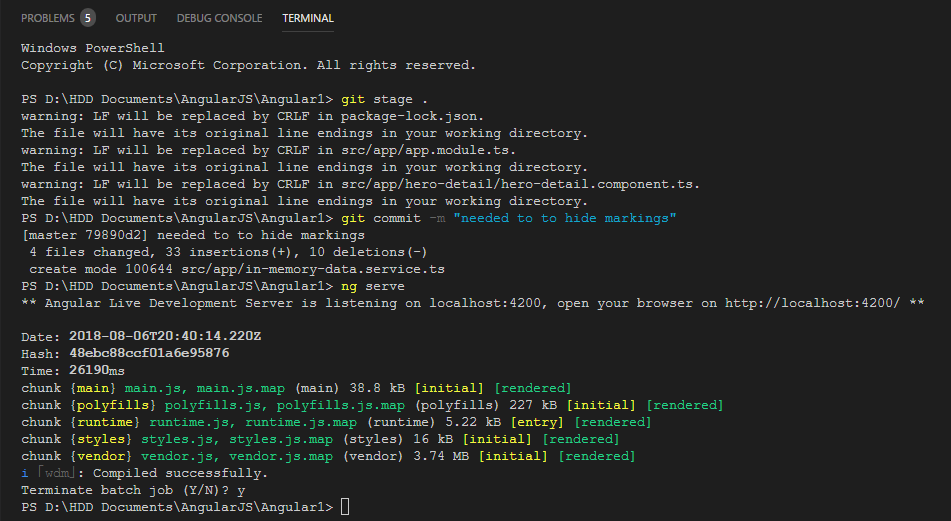
* Explorer
  + Shows the currently open editors
  + Shows the directory hierarchy
  + Shows all the objects (in object orientated code) in the file
* Search
  + Allows for Find and Replace functionality. Results are listed in alphabetical order and can be jumped to by clicking on them
  + The search function can find any instances of an item in all the files in a directory and replace them immediately
* Source control
  + Allows records to be kept of who changed what and when
  + VSC’s source control integrations mean that it is very easy to use Git and Microsoft Teams, improving collaborative productivity
* Debugging
  + VSC allows you to configure different ways of debugging a program
    - These often take the form of running different parts of the program separately for unit testing
  + When a program is running diagnostic data is displayed. VSC with no extensions can only fully debug NodeJS applications, but the NuGet service allows more configurations to be downloaded
    - Memory usage
    - Call Stack
    - Variable Contents
* Extensions
  + From here, VSC allows you to use NuGet to obtain extensions to the editor
  + These extensions can be language-specific or global

At the bottom of the sidebar is a gear icon. Clicking this will immediately take you to VSC’s settings editor. Unlike most applications, VSC allows you to view the raw Settings.JSON file and edit it directly. This can cause errors, which is why most applications hide their configuration files from users.

##### Integrated Terminal

CTRL + ‘

Visual Studio Code allows for direct integration with the built-in terminal on your OS, be it Linux’s Bash, MacOS’ shell or Windows’ CMD or PowerShell CLIs {I use PowerShell whenever I need to use the CLI}.



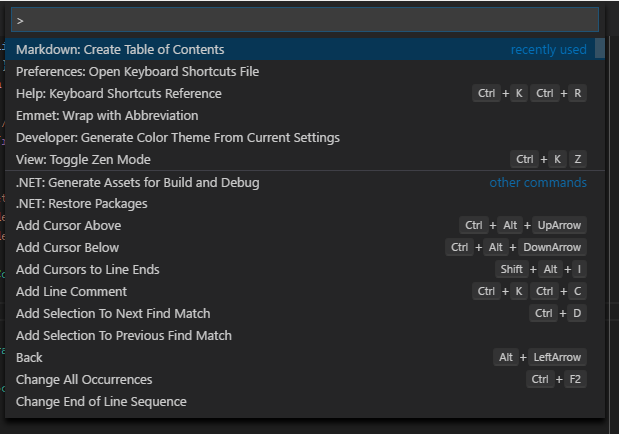
The terminal, running Windows PowerShell. In this instance I have committed some changes to a Git Repository and run {and terminated} an Angular web app.

The terminal panel also contains the Output and Debug Console, as well as the list of errors found by any linting {code parsing and grammar checking} tools installed on VSC. The Debug Console is most often used in conjunction with NodeJS applications. The Output terminal can be used for these applications as well, but some other applications {such as the “.NET” command line framework} can use this too.

Having a terminal present is always a useful tool, as it allows you to edit files and folders and run tasks without having to switch consoles. VSC also allows you to run multiple terminals within the same instance, meaning that if a CLI is forced to stop taking input due to a running process, you can simply switch to a new one while keeping the old one running.

##### Command Palette

CTRL + P



The command palette displays recently used commands, as well as suggested commands

An incredibly helpful feature of VSC is the ability to view all the commands currently available to me, and filter through them as necessary. This increases productivity as a developer does not have to sift through all the menus and selections they would otherwise have to and can instead access a command directly if they do not know the shortcut.

#### Extensibility and NuGet



Source: [NuGet GitHub Repository](https://raw.githubusercontent.com/NuGet/Home/dev/resources/nuget.png)

NuGet is Microsoft’s package management framework. It uses online hosting to distribute extensions and features to Microsoft products. It is much of this extensibility that makes VSC such a powerful editor, as many different extensions can be used for different tasks

#### Summary of features

|  |  |  |
| --- | --- | --- |
| Feature | Will I Keep It | Why? |
| Menu Bar | Yes | Having a central menu bar where all the functions can be accessed from is an inherently useful feature, which is why many apps have this feature |
| Text Editor | No | The program will use a graphical editor, so having a text editor would be redundant |
| Sidebar | No | If needed, having a sidebar to view different parts of the project would be very useful, but only if it could be integrated correctly |
| IntelliSense or Equivalent | No | The most common use for IntelliSense is the correction and suggestion of spelling and grammar. If I was creating a text editor, then I would consider adding it, but it has little use otherwise |
| Integrated Terminal | Yes | Having an integrated terminal would allow the user to edit the program then switch to using it seamlessly, without needing to switch window or start a new program |
| Tabbed Editing | Yes | This would allow a user to work on multiple programs at the same time, perhaps to copy work from an old one into a new one, the build on it |
| Full Directory View | No | This is a feature that is useful for projects with large directory tress, but is unnecessary in a small project similar to what the program will generate |
| Command Palette | Yes | Being able to access and search all commands, similar to the main menu bar, is a very useful feature. Keeping this would allow users to find the command needed with ease |
| Extensions and Extension Management | No | Adding extensions to a small dedicated program would ultimately result in excessive complication to the program for limited feature availability |
| Welcome Page | Yes | Having a welcome page as a starting point for users who have not yet opened a project would be good as it gives a set of commands to guide users into starting or continuing work on a program |
| Search | No | As each file will be made up of blocks, not words, having a search feature would likely make the program harder to navigate, instead of making it easier |
| Debugging Views | Yes | Having integrated debugging views that show every available tool when required is a useful tool |
| Code Preview | No | Generating an image of what the entire file looks like is nice to look at, but simply forces the program to use more system memory than it already is. This may result in a degradation of performance in larger files |
| Source Control Integration | No | Source control software, such as Git or VS Team Services, is primarily used in collaborative works or large projects. As each program is more likely to be small, Source Control integration becomes a more redundant feature than expected |
| Object Hierarchy View | No | While C# is an inherently object-orientated language, actively using objects in a simple program is likely to be more complex than most first-time users will be able to understand |
| Help Pages | Yes | The program is meant to help introduce users to programming, and as such should do as much as possible to help them understand the program |
| Direct access to settings file | No | Directly configuring your own settings can result in errors and crashes, directly affecting the user experience. By masking the settings, you can protect new users from making irreversible changes |

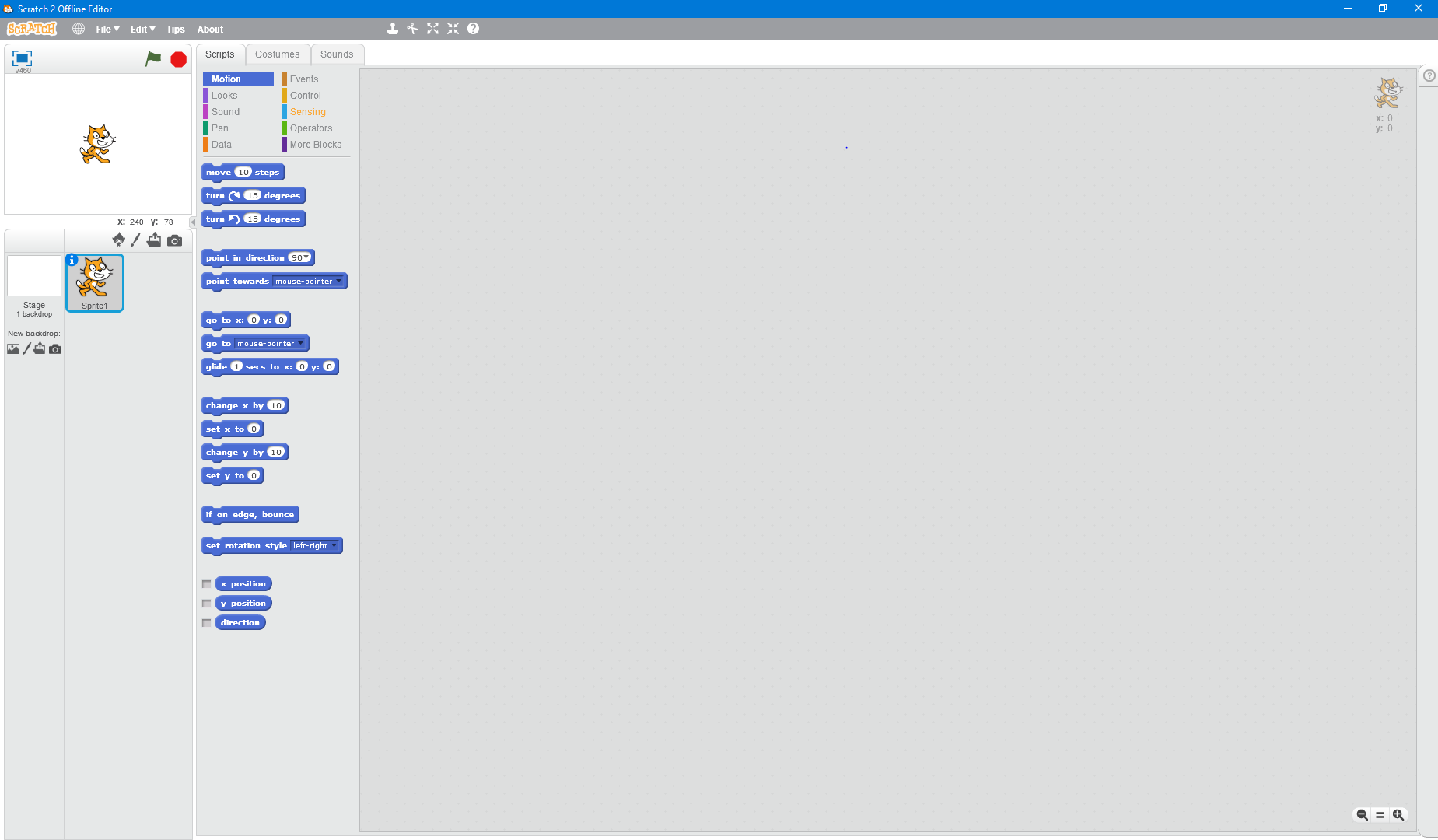
Visual Studio Code is a fully-fledged IDE designed with **all** developers in mind. As such, many of its features are much more advanced than suit my needs. In particular some of the extensible aspects and some of the code parsing tools are much more than what a program created for a younger and more inexperienced student would need.

### Scratch {and IDE}

Scratch is a visually orientated programming format produced by the Massachusetts Institute of Technology. It provides a blank canvas and a “sprite” when first opened, along with a large number of “blocks” which each have their own individual functionality. As simple as Scratch is on the surface, the variety of tools available for use allow for an incredible range of design possibilities. Because Scratch treats “sprites” as individual objects means that programming their behaviours is easy as you do not have to consider how they interact unless you want to while also allowing for an understanding of simple object orientation. The IDE is simple, but bright and colourful with a sans-serif font to allow for easy reading by younger students.

#### The Interface

The Scratch interface is clean and free from distractions, with colour icons signalling where functions are available. These icons stand out because of the lack of colour in much of the rest of the application. The main set of tools are found at the top menu of the application. These include the standard File and Edit options, plus a set of mouse tools to interact with the application.



The Scratch IDE, with a sprite and background

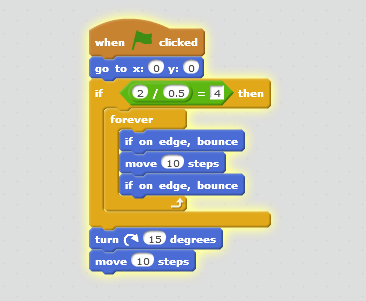
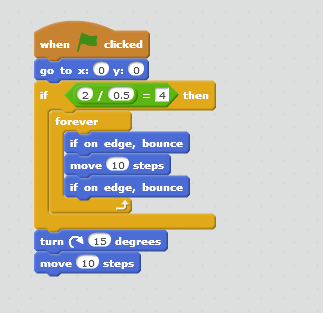


The Scratch menu bar, with dropdown menus and mouse tools.



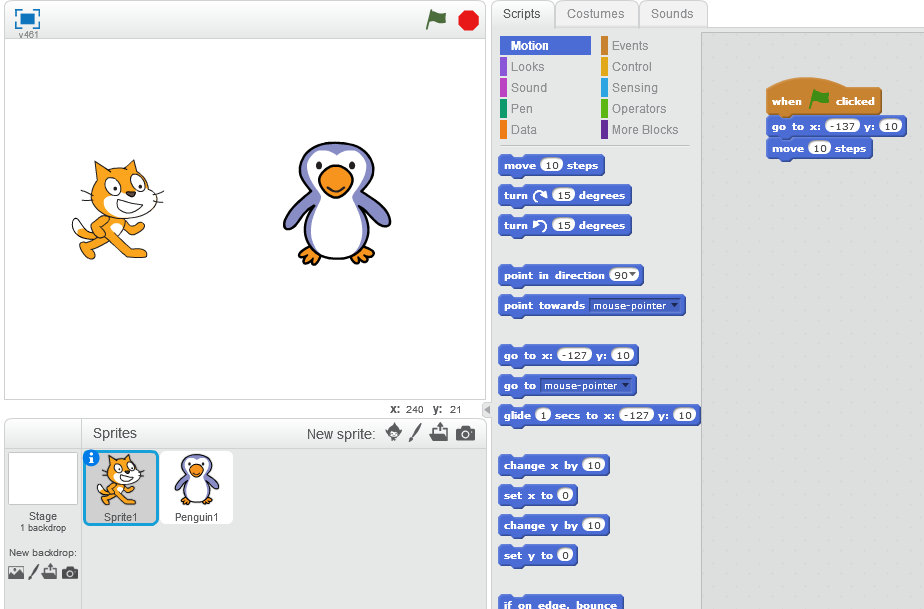
The mouse tools from left to right: Stamp (Copy sprite with first click, paste with each subsequent click), Cut (Cut sprite with first click, paste with each subsequent click), Expand and Shrink (Change size of sprite on click), Help (Click on an item to get help about it)

Scratch does not provide any debugging tools aside from highlighting the currently executing blocks. This means that if the program stops working, the only description of the issue is where code was last executing.

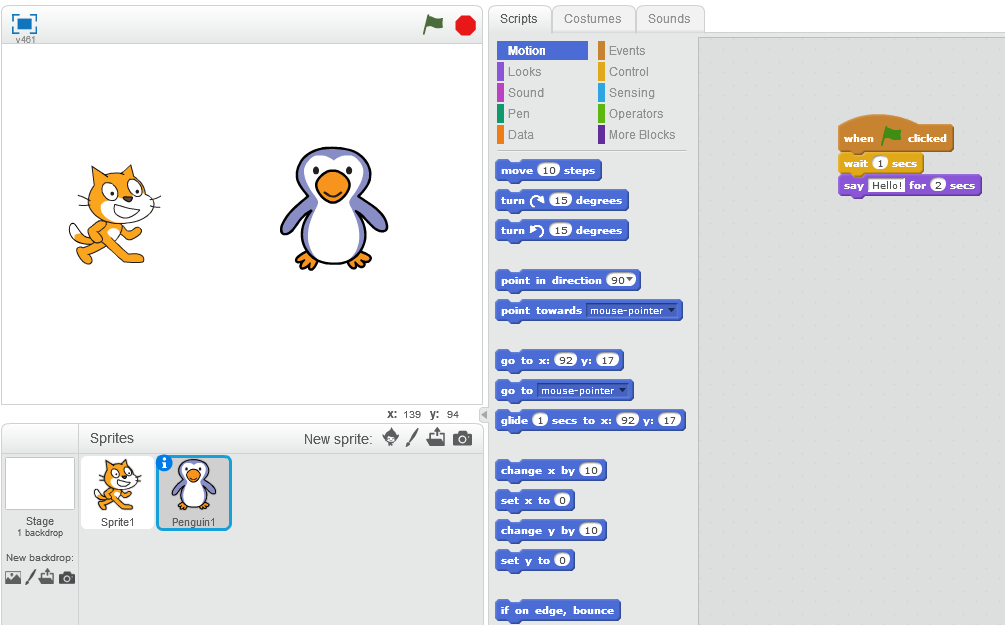


The editor highlights currently executing code.

Scratch relies on Sprites to show the functions applied to it. Each sprite can be treated as an object, with different sprites having different scripts attached to them.

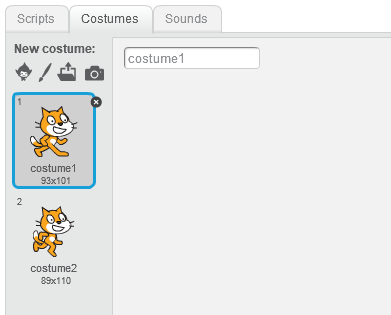


The scripts for Sprite1 are shown in the workspace



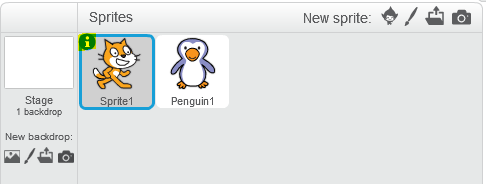
After selecting it by either clicking the sprite or clicking the sprite icon, the scripts for Penguin1 are shown.

Sprites can have “costumes” and sounds assigned to them. Costumes allow sprites to change their appearance during the execution of a program. The user can switch between Scripts, Costumes and Sounds using the tabs at the top of the workspace.

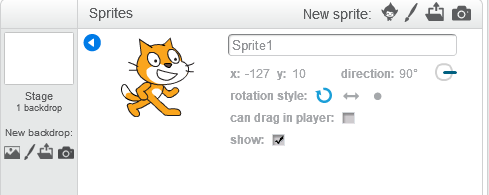


The “Costumes” tab of the workspace

Each sprite also has an “Info” view available, which can be viewed by clicking the “I” in the sprite icon. This view shows information such as how the sprite rotates, its current location/rotation and if it is visible or not. The sprite’s name can also be changed in this view.



The “I” is highlighted in this image



The info view provides additional information about a sprite

The scratch interface also has several other features. The size of the blocks can be changed to make them suitable for the user using the magnification buttons at the bottom right of the workspace, and program execution can be started and stopped using the icons at the top right of the “stage”. The current states of different variables can also be displayed onscreen during execution.



The magnification buttons control the zoom level of the blocks



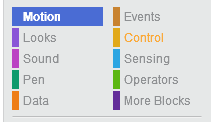
The green flag and red stop sign clearly indicate that they start and stop the program



The variables can be shown onscreen using the checkboxes, as shown

#### The Toolbox

Scratch uses building blocks to create the programs. These blocks are split into 10 different categories, each one with a different type of functionality:



The toolbox categories

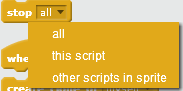
These blocks are accessed by clicking on them in the toolbox and dragging them out to where you want them. They then snap to a nearby available location when close enough.

##### Motion

Motion blocks control how the on-screen sprite appears to move, including commands to “step”, with each step referring to an increment of 1 unit on the built-in grid {}. You can also command the sprite to move from its current location to a given XY location, rotate, or move from its current location by a given X or Y value. These blocks are an example of Process blocks, performing actions when called

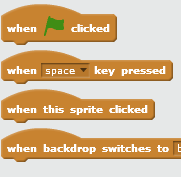
##### Events and Control

Events and Control blocks are used for control flow. The Control blocks include If/Else blocks, For/While loop blocks and Stop blocks.



Control blocks. Note that the “stop [all]” block can be used in different situations

Event blocks allow a sprite to react to a given event, often in the form of a Message or when the program starts. These blocks include the On Start block, the On Key Pressed block and the On Click block.



These blocks make up some of the most used blocks, along with the next category, as they are used to control the processes the program takes when given an input.

#### How Scratch Runs the Code UPDATED

Scratch is an interpreted language. It runs each code block sequentially, and only parses them when needed. This type of language allows code to be changed and re-run almost immediately, as there is no phase before code runs.

#### Summary of Features

|  |  |  |
| --- | --- | --- |
| Feature | Will I Keep It | Why? |
| Toolbox | Yes | The Toolbox provides a centralized location for all the blocks available to the user. It is always present in the relevant areas and is intuitive to use |
| Tips Panel | No | The Tips Panel allows the user to get help immediately for what they need, improving workflow and aiding understanding of the program. The ability to click on an item for help is also very useful, however ultimately it adds nothing to the primary program |
| Code can be run at any time | Yes | This is a useful debugging tool as you can run code immediately to find errors and act upon them quickly |
| Info viewer | No | If applicable, having a separate view to show properties of an element is very useful as it would save screen space and allow users to view extra information with ease, but it is not necessary for my purposes as I am likely not going to be allowing users to create a GUI based program |
| The Stage | No | As my application will be command-line based, it is unlikely that I will need a version of the stage as it is best suited for graphical applications |
| Snapping blocks | Yes | The blocks can snap to other nearby blocks where available. This functionality is very useful as blocks that cannot fit together will not, preventing some types of errors |
| Save / Load a project from a file | Yes | This functionality allows work to be stored, shared and restored with ease |
| Support multiple languages | No | Scratch can be used in over 40 languages from across the world, and is used in over 150 countries [[source]](https://scratch.mit.edu/about), however my program is unlikely to reach such a broad audience |
| Mouse Tools | No | Scratch’s mouse tools are designed to help with developing the graphical output of the program, not the logic behind it, so keeping these would be pointless |
| Copy/Paste blocks | Yes | The ability to copy and paste blocks to repeat a line of code multiple times easily, such as “delay {5} seconds” between several parts of the code is likely to be useful for many users |
| Display variable data when needed | Yes | This may be good for debugging purposes, as the variable contents can be shown when needed to help check if the program does what it is meant to do |
| Interpreted Language | No | While interpreters improve short term performance by not requiring the code to take the time to be compiled, they take longer overall for files that are run repeatedly and require that the interpreter be installed on any device that uses the program. An executable file, while taking longer to compile, can be run on most windows platforms and is portable. |

Scratch is an ideal introduction to coding. It provides a clear and distraction-free interface and simple functionality but allows users to develop code to as high a level as they desire.

## Requirements Specification

### Essential Features

|  |  |  |
| --- | --- | --- |
| Requirement | Description / Implementation | Justification |
| Easy to Read and Understand | Large, colourful interfaces with contrasting text colours to maximise readability. Sans-Serif fonts could also be used | Young students are not always able to read as competently as their elder counterparts, as such the design should reflect this |
| Intuitive Usage | Tooltips could pop up after the mouse is hovered over the block, or a button could be clicked to show a help box. A full help page should also exist | Explaining new concepts can be quite difficult, so the program should be easy to use, with help available if necessary |
| Stability | Because students will be encouraged to code their own applications, errors could be handled within the program (instead of crashing the entire program) on a block-by-block basis | A program that crashes often is not usable in any environment, and as such the program should be as stable as possible. |
| Load and Save To / From A File | The program should allow users to open files they have already begun working on, and save their current progress | If they cannot save their work, they have been doing would have to be re-done next time the program is used, cutting into lesson time |
| Accurate Representation of Programming Ideas | The program should make use of if-else and logical operators. It should also introduce the user to Types such as String, Int and Decimal. | As the program is designed for use by novice programmers, it should ensure that they are introduced to important aspects of computing correctly. |
| Toolbars / Menu Bars | A horizontal menu bar at the top of the window, with sub-menus containing additional functionality that cannot be accessed directly onscreen | Toolbars have been commonly used in GUI-Based applications since the early 1980s [1][2][3]. Most users are used to them and can intuit the functionality provided through the sub-menu names |
| Help Menus | These could be accessed from a side panel or an overlay that appears when a context menu item or relevant button is clicked | Providing assistive features to users allows them to develop at their own pace, and makes it less likely for them to require additional assistance |
| Settings Editing | If required, users could personalize their workspace by changing settings such as text size, colours or even minor functionality aspects through a GUI editor page | This allows users to set up a workspace that’s best for them. (It is easily implemented in .NET as the framework provides a built-in settings file with the application) |
| Context Menus | When an area of the editor GUI is right-clicked, a context menu should appear to provide additional commands, such as: Copy/Paste, Help or Delete | This allows the GUI to be less cluttered as the buttons to activate these commands are hidden until needed, instead of always being present |
| Toolbox | A panel that contains categorised blocks, each of which can be dragged into the main workspace | This provides a centralised space to retrieve items and simplifies the UI because of this |
| Welcome Page | A page that appears when the application is opened. It should provide links to recently opened files, and allow the user to open/create a file | This gives the user a basic idea of what the program can do, as well as allows them to rapidly resume working on previously developed applications |
| Copy / Paste | Right clicking a block and selecting `Copy` should add the item to the Windows clipboard (Note: this could be done using JSON?). Right clicking an empty space and selecting `Paste` should produce an exact replica block at that location | This allows the user to copy code snippets to the clipboard and then use them in other programs, or the same program |
| Zoom In / Out | Buttons located either at the bottom of the window or in one of the toolbars should change the font scaling | Users with vision impairments would need a larger text size to allow them to use the program easily |
| Integrated Terminal | When a program is built and run, the input, output and error streams should be read and written to by the main program, and the child window should not be visible | Being able to run a separate terminal process within the application would allow the user to interact with their finished program without having to leave the application |
| Command Palette | By either selecting a sub-menu item from a toolbar or using a keyboard combination, the user should be able to search for commands available in any sub-menu | Having a simple list of commands that is immediately available is great because it allows users to access those commands quickly when needed, but doesn’t restrict new users from accessing those commands |
| Debugging Views | When a user program is running, the main application should be able to monitor and display feedback as to the state of the application, reporting errors where necessary | This view would allow users to see when and where their programs crashed or exited, as well as send inputs to them. This allows all debugging to be handled within the main application, so the user doesn’t have to switch contexts |
| Code Can Be Run at Any Time | A button or keyboard combination could be pressed that compiles the code, switches the application to the debugging mode and runs the newly created program | This makes testing quick and easy, as well as allows users to review immediate changes to their code |
| Snapping Blocks | The Blocks of the program should be able to “snap” together to when they overlap correctly | This “Snapping” makes it easier for blocks to connect, as otherwise they’d have to be aligned “just right” before functioning |
| Display Variable Data When Needed | The user could enable a “Debugging mode” in the output application that prints the contents of variables, or the main program could read them from a separate but continuously updated file. | This allows users to observe the contents of their variables to ensure that they are working properly |

[1] – [Web Designer Depot](https://www.webdesignerdepot.com/2009/03/operating-system-interface-design-between-1981-2009/), [2] – [Wikipedia](https://en.wikipedia.org/wiki/History_of_the_graphical_user_interface), [3] – [Harding University](https://view.officeapps.live.com/op/view.aspx?src=https://www.harding.edu/fmccown/gui/history-gui.pptx): All three sources corroborate that the first GUIs were programmed in the early 1980s.

### Requirements Specification for The Program

#### Program Requirements

* Separate Design and Debugging Views
  + This allows for the User to easily recognise when code is running, as well as declutters the workspace and improves performance by reducing simultaneous thread operations
* Can be distributed across a network without sharing user data to other users
  + GDPR prevents user data from being shared without their permission

##### Design View

* Workspace panel in centre of window
  + Users should have a large area on screen where they can “build” their programs. This could simply be a large panel that is blank when first opened. This open space will be where the student spends most of their time with the program
* Blocks
  + All Blocks
  + Process blocks
    - These blocks are blocks that equate to a C# keyword or .NET functions such as if-else or System.Threading.Threads.Sleep(x)
    - These blocks can have data block inputs, but cannot be defined by the user
  + Data Blocks
    - Data blocks are equivalent to data literals
      * Types include
        + String
        + Decimal
        + Boolean
        + String []
    - They cannot be used by themselves, but act as an input to a process or function block
    - Variable blocks
      * These are like normal data blocks but can be defined by the user
      * Variable block contents should be able to be displayed in the debugging mode
  + Function Blocks
    - These blocks allow the user to define their own reusable code
    - Definition Blocks
      * These are like Start blocks, but can be called at any time and will run their child blocks each time their reference block is run
    - Reference blocks
      * These provide a reference to a function definition block and can be used as a normal process block
* Toolbox panel containing every Block
  + There should be a centralised area from which every building block of the program can be obtained and placed into the workspace. This makes it easier to find the items you want to add into the program
  + Each category should be separated
* Data Types
  + C# is a strongly typed language, and an understanding of Data Types and how they work is imperative for any programmer. As such the program must make it easy to distinguish between Types and their uses
* Menu toolbar at the top of the screen
  + This toolbar would contain commands such as:
    - File
      * Open
      * New
      * Save
      * Options
      * Show in C#
    - Edit
      * Cut
      * Copy
      * Paste
      * Command Palette
    - Run
      * Debugging View
      * Full App
    - Help
      * About Blocks
      * About the Workspace
      * About Debugging
* Command palette panel that drops down on keyboard combination or menu bar selection
  + A panel that opens from the side that you can search for a command in using a textbox. Results could be shown using a ListBox (.NET Framework component). When clicked the command is run and the panel will close

##### Debugging View

* A fully functioning Command-Line Interface program as an output
  + This would be the “\*.exe” file produced by compiled code
* Textboxes to display variable output to the side of the CLI window
  + This allows the user to view the contents of variables in the program as they update
    - * Note: I could mark variable outputs with a `¦` character or similar. Then hide those outputs from the user and instead use them to update variable content information IE `¦ {“var1|data1” ¦ “var2|data2” ¦…} ` etc
* A stop button to immediately end execution and return the User to the design view
  + This allows users to get back to editing their program easily without having to close and reload the program
* If an error occurs, a dialog box should appear informing the user of the error, but the main program should not crash
  + This allows errors that are found to be dealt with quickly without having to reload the program. It also prevents loss of data from a crash

### System Software and Hardware Requirements

|  |  |  |
| --- | --- | --- |
| Requirement | Description | Justification |
| 4GB of RAM or more | The system running the program should possess at least 4GB RAM | This allows the system to run the program with a lower chance of running out of RAM |
| Microsoft Windows 8.1 or later | The OS should be Windows based and released post-Windows 8 | The latest versions of the .NET framework offer the best user experience compared to previous builds |
| MSBuild.exe must be available for use by the user or program | The user or program must have MSBuild.exe installed, and must have the relevant permissions to use it | To build and run programs using the solution specified later, my program must have access to the MSBuild.exe program that compile C# code |
| .NET Framework Runtime Libraries Must Be Preinstalled | The .NET framework is what my application is built upon and it makes many references to these libraries when running | These libraries allow programs that are built with references to them to run. Without them the program would likely crash as it would be unable to gain access to modules and data that are required for .NET programs to run |
| The User must have Read / Write / Run permissions for foreign Executables | The user account must be allowed to run custom executable files at will | On some networks these permissions are reserved for special accounts to protect network security and private data, but my program creates these custom executables and tries to run them. Failure to run them will result in an error |

## Limitations Of The Program

### GUI based app development

Currently, while I would like to implement a graphical interface, the time it would take to develop is significantly greater than the time I have available to me for this coursework. Building a CLI for this program is difficult to start with, so producing a GUI would be a lot harder. While this would provide an interesting challenge, it is likely that it would not be completed within the deadline.

### Dedicated ability to share programs online

If users want to share their work, it’s relatively simple to send files by email. This means that the file is easily shared through external means without a dedicated file transfer service. This means that it is unnecessary to supply such a service with the program as this functionality can be provided elsewhere.

### Support for multiple languages

Because of how the program will work, supporting multiple languages would be very difficult as I would have to design a second type of language to do same thing {like the differences between Visual Basic and C Sharp}, which is likely to make the program more complicated for the user as they would have to choose between two languages. It would also make the teacher’s job harder as they would have to be knowledgeable about either choice.

## Measureable Success Criteria

* Can a teacher use it to help assist students learning?
  + Do the students feel that it is an effective learning aid?
    - If students don’t want to use a program, then they are less likely to engage with it
  + Does the teacher find it easy to teach with?
    - If a program hinders teaching, then it is ineffective as a teaching aid
* Can students use the program to produce a functioning program?
  + Can the resultant program take inputs from the user?
  + Can the resultant program take actions based on those inputs?
  + Can the resultant program output data as the user has instructed?
    - All these requirements are part of the requirements of a true program. A program to help teach students how to write a program must create a proper program to be successful
* If unexpected data is inputted, will the error be handled correctly?
  + Will the resultant program’s execution be halted?
  + Will the main program continue to execute, reporting the error to the user without crashing?
    - If the user’s program halts due to an unexpected error, it shouldn’t affect the main program’s execution
    - If the main program encounters an error, it should inform the user and attempt to save their work before closing itself
* Can students continue working after finishing a session
  + Can they save their work to a file?
  + Can they open their work from a file?
    - These allow students to continue working after periods of time without risking a loss of work
* Do students feel that the program is useful?
  + Can they demonstrate an understanding of programming concepts?
  + When tested, can students produce a program that uses basic concepts of programming correctly?
  + Does this understanding carry over to other languages?
    - If these criteria are not met, then the program has not fulfilled its purpose of explaining these concepts
* Does the program conform to modern standards of data security in schools?
  + Is user data transferred elsewhere without their permission?
    - The school could be fined between £2,000 and £100,000 per breach of personal data
* Can students design and edit their own program?
  + Can they click and drag an item into a workspace?
    - If they cannot place items into the workspace, they’re not going to be able to build up a program in the first place
  + If two compatible items are adjacent, will they “snap” together?
  + Will “snapped” items run their code sequentially?
    - These criteria make it easier for the user to build the program
* Does the program have a debugging view?
  + Can the user view variable contents when required?
    - This makes it easier to debug programs by viewing how and when variables change
  + Can users end debugging at any point?
    - If the program falls into an infinite loop and doesn’t cause a stack overflow error, the user should be able to exit the program anyway
* Is help available if needed?
  + Can users view a description of each block?
    - Being able to know what each block does when needed can assist a user greatly when building a program
  + Can users explain how the software works?
    - If they do not know how to build up a user program, then they are not going to be able to make use of the main program
* When the program is first opened, does a welcome page appear?
  + Does this page display recently opened files?
    - Being able to quickly access previous files makes it easy for users to continue working where they left off
  + Can the user create a new file from here?
    - If the user hasn’t yet made a file or wants to start a new one, then making it easy to do would help them get started

# DESIGN

## The Components Of The Problem

The problem can be split into several sections:

* Assists students in the development of programming techniques
* Allows for saving and restoring of files
* Has a UI to allow users to design their own programs
* Has a clear output
* Allows for basic debugging

The hardest part of this program is likely to be the output. I have two possible approaches to this, outputting a fully functioning .NET program using the C# Compiler, or running an instance of a console window from the program. My decision on this is likely to affect the overall result of the algorithms used and implementations chosen.

### Programming Techniques

This is by far the largest of all the sections, and can be split down again into several subsections

* Inputs and outputs
* Logic and logical operations
* Functions and subroutines
* Data types and structures
* Development techniques

#### Inputs and Outputs

By providing a clear way of inputting data and outputting results, students are given the ability to begin to understand how programming works, as simple command-line programs can be developed and built on easily.

#### Logic and Logical Operations

A fundamental part of computing is the logical operations that allow computers to run. An understanding of how these operations work is essential, and as such the program will contain references to these.

#### Functions and Subroutines

Programmers use functions and subroutines to break a program down into individual elements. These allow them to make more progress and to make code easily re-useable and are essential in modern programming.

#### Data Types and Structures

Types are used to allow data to be represented, manipulated and stored. They are incredibly useful in programming as they allow data to always be used in an exact way. Building on from types are structures, which provide extensibility to standard types as well as added data security, which are important concepts to understand.

#### Development Techniques

There are many development techniques available, and every programmer has a different preference as to how they like to develop. Because of this, the program should not try to force any style upon the user, instead allowing them to develop how they choose to.

### Saving and Opening Files

### Design and Development of The User’s Own Program

### Output of The Program

### Debugging

## Designing a solution

### Language

I will be using Microsoft’s Visual C# programming language, with the extensive .NET API libraries to assist me. This API allows access to many of Windows’ functions and capabilities, including {but not limited to}: predefined data types, built in security functions, simplified access to the file system, and many others. Immediate compiling to an executable file is very useful for debugging purposes, and the finished executable is easily distributed.

C# is a strongly typed, object-oriented programming language. It is loosely based on C, from which it gets its name

A downside to C# is that it relies on already having the relevant libraries installed on the system. This means that any extra libraries either must be manually installed or included in the same folder as the executable file. C# also lacks native high-quality drawing tools, meaning that any custom drawings done are likely to be low quality or produced using bulky code, increasing the file size unnecessarily.

The functionality provided by C#, however, is remarkable. The language allows you to create your own components if the ones already available do not quite suit your needs, and the toolbox included with the IDE (Visual Studio) has a built-in ability to load these components into the GUI editor. This editor allows Windows Forms to be drawn up rapidly and makes it easy to present a visual idea of how the program may work without having to write as much code. This is due in part to the .NET API library, which provides lots of basic components to help with rapid application development.

C# compiles to a “.exe” file, capable of being run by most Windows operating systems. This makes the program easy to place on a network and be run by Windows computers, however other OSs would have to have a version coded specifically for them, perhaps using the Xamarin framework or similar, however as I am not used to these frameworks, I will not be programming using them.

### Something Else // TODO

To begin with, I will develop a Command-Line Interface {CLI} based Input-Output {IO} system that allows users to both take inputs from the CLI and output results to it. If I get the time, I would like to also develop a Graphics-Based interface as well, but given the timeframe I must work with, along with inherent graphical difficulties that come with an approach using C#, it is unlikely that this will be completed.

The logical operators, data types and functions will all be focused on

# DEVELOPMENT

## Overview

I have broken down the task into several smaller sections:

* The UI
  + “Toolbox”
  + Workspace
  + Testing?
* The Blocks
  + BaseBlock
  + Chaining them together
  + Their individual functionality
* Running a program
  + CLI
  + Compile vs Interpret
* Saving and loading your work
  + Saving
  + Loading
  + Save to C#?

## The UI

My UI is going to be somewhat basic in comparison to the UI designs of many other programs. It will consist primarily of two Windows Forms, the first one being the designer, the second being the debugger.

### The Debugger

This form is going to be instantiated by the designer form, so I can pass the name of the compiled program’s executable to the form.

public FrmDebugger()

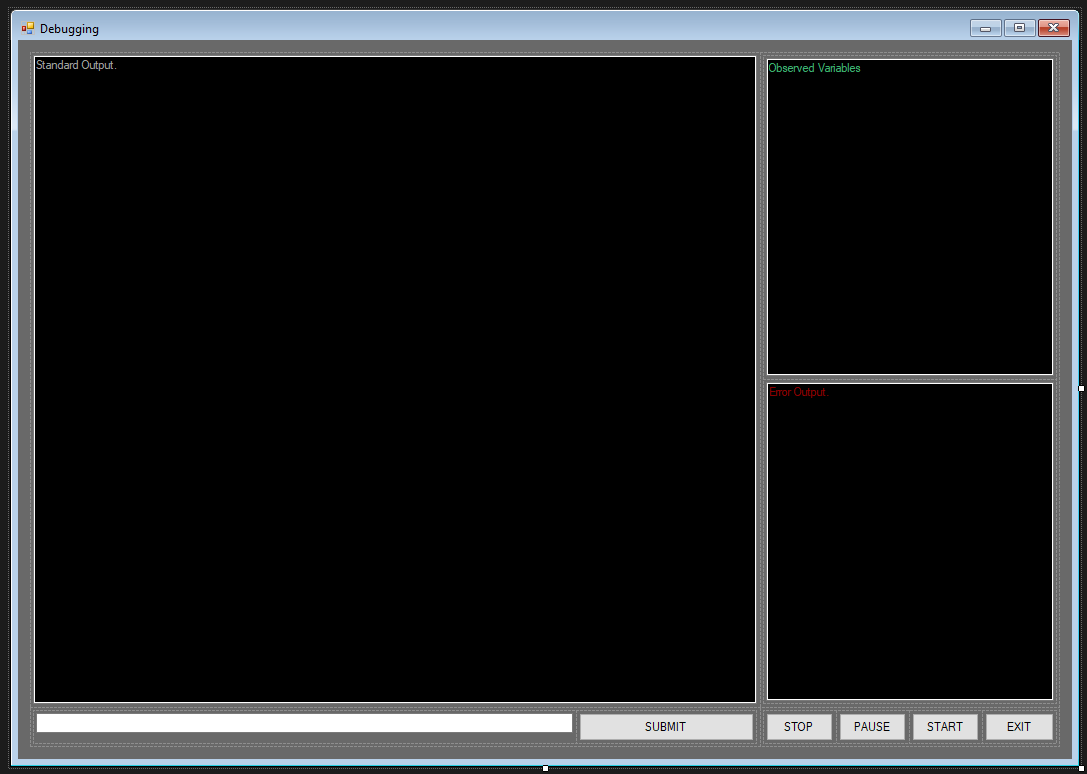
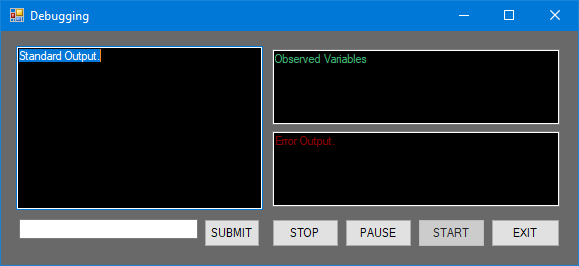
public FrmDebugger(String ToDebug = null, FrmDesigner MyParent = null)

C# has built in method overloading, so there is no need to declare a method as having several overloads; the compiler does it for you. The second constructor allows the instantiating form to be shown or hidden by passing the object reference to the debugger form.

It will feature an Output log, an Error log and a Variable Viewer. There will also be a text input with a submit button to allow the user to send input to the program.

#### The form

I began by adding several “TableLayoutPanel” s to the form to ensure responsiveness and to make it easier to design, reducing the amount of time spent on the UI itself. I then added 4 “TextBox” objects, three for output {On which I set the “Read-Only” property to “true”} and one for input. I also added 5 buttons to Stop, Pause, Start and Exit the debugger as well as Submit an input to the program.

  
The debugger view once all UI elements had been placed  
  
The debugger view resizes automatically. Note that the buttons remain the same height, so text is always displayed. The minimum size of the view has been set to (593, 273) px. I found that this size prevents the outputs from being too cramped, as all the text on the buttons are visible and the standard output is not too small

I then tested that the form would load and be responsive even without a program loaded for debugging {Which is why I have the empty constructor}. To do this I added a button to the start-up form (The designer view) and added an Event Handler to it that shows the debugging view

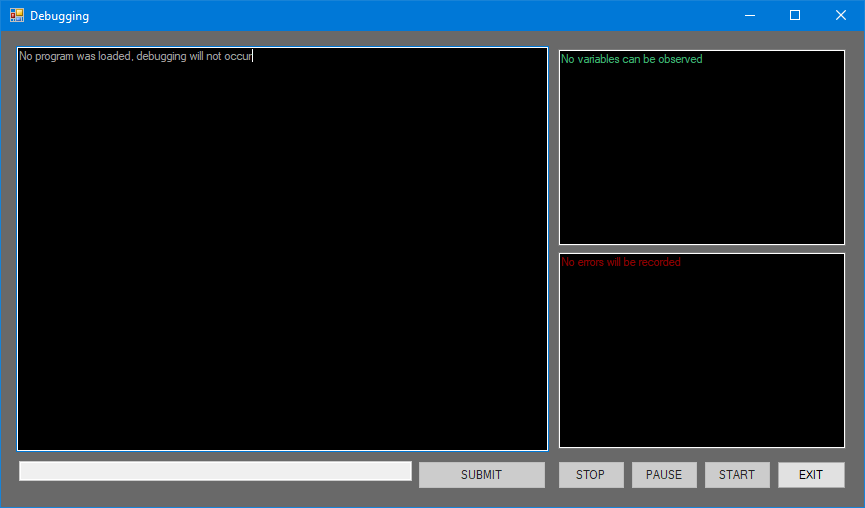
BtnShowDebugger.Click += new EventHandler(BtnShowDebugger\_Click);

private void BtnShowDebugger\_Click(object Sender, EventArgs E) {

var MyDebugger = new FrmDebugger();

MyDebugger.Show();

}

  
When loaded without a program, the buttons are disabled, and the outputs display an error message. A common motif in my programs is “user-proofing”, where even if a user performs an action that should normally be unavailable, the program can handle it. For example: If the user somehow manages to load this form without a program attached, this view will be shown instead of the program crashing

#### The Process

To debug a Process, you first need to have a Process. To do this I decided I would pass the process “\*.exe” file-path through to the FrmDebugger class in the constructor. This would allow the FrmDebugger to manage the process itself. I could have passed an already instantiated program to the debugger, but this would have meant wasting memory as I would end up with two Process objects that refer to the same program.

Thanks to <https://stackoverflow.com/questions/355724/embedding-a-dos-console-in-a-windows-form>, I found out about the RedirectStandardInput, RedirectStandardOutput and RedirectStandardError properties of the Process class, as well as CreateNoWindow and UseShellExecute. These allow you to control the inputs to a program as well as read the outputs. CreateNoWindow also allows you to prevent a console window from being created, making the application appear to run much smoother. My code for instantiating the process is as follows:

private readonly Process \_DebugProcess;

public FrmDebugger(string FileToDebug, FrmDesigner MyParent) {

…

\_DebugProcess = new Process {

StartInfo = new ProcessStartInfo {

FileName = FileToDebug,

CreateNoWindow = true,

// No new window

UseShellExecute = false,

// Don't run it in the system shell

//Arguments = "DEBUG"

RedirectStandardError = true,

// Allows me to read the error output

RedirectStandardOutput = true,

// Allows me to read normal output

RedirectStandardInput = true

// Allows me to send my own inputs to the program

},

EnableRaisingEvents = true

// Allows me to hook on to the "Exited" event

};

…

}

I keep \_DebugProcess scoped to the class to allow functions within the class to operate with ease on the process. Because it can be null, some methods will refer to it by using \_DebugProcess?.SomeMethod(). This builds null checking into the program without taking up large amounts of code with if (NullableType is null) {} statements. For example, when I call KillDebugProcess() in the event of an error or exit, there is a chance that the process has not been assigned to yet, which could cause a NullReferenceError at runtime:

/// <summary>

/// Closes all open streams and then kills the process if it's still alive

/// </summary>

private void KillDebugProcess() {

\_OutputReader?.Close();

\_ErrorReader?.Close();

\_InputWriter?.Close();

\_DebugProcess?.Kill();

\_DebugProcess?.Close();

}

I also added two cosmetic methods that simply enabled / disabled the buttons when debugging began, was paused or stopped:

#region Interface appearance methods

/// <summary>

/// There used to be something here, but removing this

snippet breaks the WYSIWYG editor

and I might need it later

/// </summary>

/// <param name="Sender">Reqd. for events</param>

/// <param name="E">Reqd. for events</param>

private void FrmDebugger\_Load(object Sender, EventArgs E) { }

/// <summary>

/// Disable all the buttons (except EXIT. EXIT is always available)

/// </summary>

private void StopDebuggingInterfaceChanges() {

BtnExitDebugging.Enabled = true;

BtnPauseExecution.Enabled = false;

BtnStartExecution.Enabled = false;

BtnStopExecution.Enabled = false;

BtnSubmitInput.Enabled = false;

\_IsDebugging = false;

TxtInputToProgram.Enabled = false;

}

/// <summary>

/// Enable the buttons except START because we've already started

/// </summary>

private void StartDebuggingInterfaceChanges() {

BtnExitDebugging.Enabled = true;

BtnPauseExecution.Enabled = true;

BtnStopExecution.Enabled = true;

BtnSubmitInput.Enabled = true;

BtnStartExecution.Enabled = false;

\_IsDebugging = true;

TxtInputToProgram.Enabled = true;

}

#endregion

I considered setting up 2 StreamReader objects called \_OutputReader and \_ErrorReader and a StreamWriter object called \_InputWriter. These would be how I interact with the program, but I realised I could simply directly access the StandardInput, StandardOutput and StandardError properties of the Process class, and save myself some unnecessary variables. I then created 3 subroutines that would interact with these properties, with the Readers designed to run when an event is fired, and the Writer to run when the submit button is clicked:

/// <summary>

/// Adds the output text to the Standard Output

/// </summary>

private void ReadOutput() {

var Out = \_DebugProcess.StandardOutput.ReadToEnd();

TxtStandardOutput.Text += Out + Environment.NewLine;

}

/// <summary>

/// Adds the output text to the Error Output

/// </summary>

private void ReadError() {

var Out = \_DebugProcess.StandardError.ReadToEnd();

TxtErrorOutput.Text += Out + Environment.NewLine;

KillDebugProcess();

// Something went wrong so kill the process Just In Case

StopDebuggingInterfaceChanges();

}

// This method is attached to the BtnSubmitInput.Click

event using the WYSIWYG editor

private void BtnSubmitInput\_Click(object Sender, EventArgs E) {

var ToSubmit = Regex.Replace(TxtInputToProgram.Text, @"\p{C}+", string.Empty);

// Remove invisible control characters (Thanks Regex)

if (ToSubmit == string.Empty) return; // Don't send an empty string

\_DebugProcess.StandardInput.WriteLine(ToSubmit);

// Send input to the program

TxtInputToProgram.Text = ""; // Empty the text box

TxtInputToProgram.Focus(); // Give focus to TxtInputToProgram

}

Getting the Input to work was relatively simple. I could have used the async modifier and waited instead of tying up resources, but this way means that no other operation can start until the input has been sent, which is good as I don’t have to worry about asynchronous operations yet.

##### The program to run

I needed a simple program, so I decided to create one that simply outputted its data to the console forever, counting upwards so I could see what was being output when. This simple approach made debugging easy as there was less that could go wrong.

private static void Main(string[] Args) {

var Count = 1;

while (true) {

Console.WriteLine("Outputting to console - " + Count);

Console.ReadLine(); // Wait for ENTER to be pressed

Count += 1;

}

}

##### Asynchronous read operations that don’t cause a program to hang

The two StreamReader listener methods are designed to work on an event being built into the StreamReader class. Unfortunately, such an event does not exist. To circumvent this, I tried simply using StreamReader.ReadToEndAsync(). This did not work. That’s partially due to how programs output data: They only set the “End of stream” marker once they’ve exited, so while the full output of a program will be displayed it will only be shown after the program exits:

private async void StartOutputRead() {

TxtStandardOutput.Text += await

DebugProcess.StandardOutput.ReadToEndAsync();

}

private async void StartErrorRead() {

TxtStandardOutput.Text += await

\_DebugProcess.StandardOutput.ReadToEndAsync();

}

I then tried to approach the problem using C#s Tasks. These function as background tasks of the program but are actually a bit more complicated. When a Task is created it is sent into the “Thread Pool” to run in a different Thread to the UI. This prevents the UI from hanging but means that object references are not preserved. This means that (without the use of Delegate Types) a Task cannot cause the UI thread to update or change information about the objects in the UI thread. Unfortunately, this understanding of Tasks only came to me after spending about 2 hours trying to work out why the following code wasn’t working:

var ReadStreams = Task.Run(() => {

while (!\_DebugProcess.HasExited) {

if (\_DebugProcess.StandardOutput.Peek() == -1) continue;

var Output = ((char)\_DebugProcess.StandardOutput.Read())

.ToString();

TxtStandardOutput.Text += Output;

}

if (\_DebugProcess.StandardError.Peek() != -1) {

var Output = ((char)\_DebugProcess.StandardError.Read())

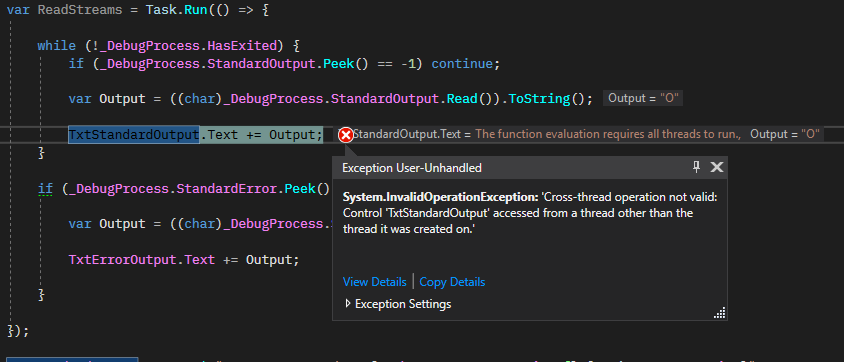
.ToString();

TxtErrorOutput.Text += Output;

}

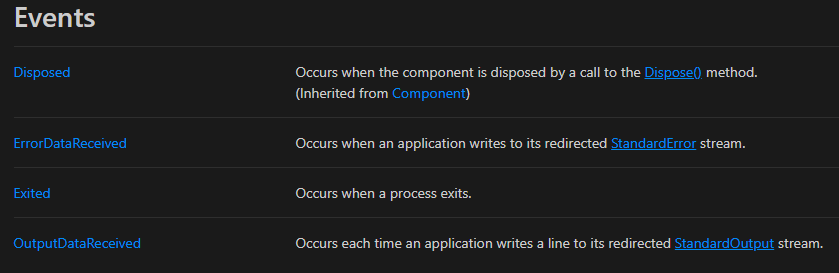
});

The task threw an InvalidOperationException because the Thread it ran in did not contain a reference to the \_DebugProcess.



Well that didn’t go too well. At least Output is working though.

Instead of trying to use the unnecessarily complicated delegate system, I chose to look at the .NET API Reference again. Here I found some Events on the Process class that I didn’t notice before



These would have saved a lot of time and effort if I had already known about them

\_DebugProcess.ErrorDataReceived += new DataReceivedEventHandler((S, E) => {

Console.WriteLine(@"Error: " + E.Data);

});

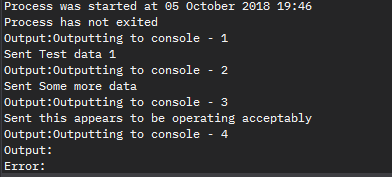
\_DebugProcess.OutputDataReceived += new DataReceivedEventHandler((S,E) => {

Console.WriteLine(@"Output:" + E.Data);

});

\_DebugProcess.BeginOutputReadLine();

\_DebugProcess.BeginErrorReadLine();



From testing I noticed that forcing the program to stop caused an empty error. Fortunately, my error handler will only update if the data isn’t empty so it should be okay

I then added some anti-error fixes to prevent method calls to an errored function. The Try-Catch (InvalidOperationException) statements prevent the code from breaking if the error occurs because the user pressed STOP, which forcibly kills the program and forces it to throw an error.

\_DebugProcess.ErrorDataReceived += new DataReceivedEventHandler((S, E) => {

if (E.Data != "") {

TxtErrorOutput.Text = $@"Error detected: {Environment.NewLine}

{E.Data}";

}

Console.WriteLine(@"Error: " + E.Data);

try {

\_DebugProcess.CancelErrorRead();

} catch (InvalidOperationException) {}

try {

\_DebugProcess.CancelOutputRead();

} catch (InvalidOperationException) {}

\_DebugProcess.EnableRaisingEvents = false;

try {

if (!\_DebugProcess.HasExited) {

\_DebugProcess.Kill();

\_DebugProcess.WaitForExit();

}

} catch (InvalidOperationException) { }

StopDebuggingInterfaceChanges();

});

\_DebugProcess.OutputDataReceived += new DataReceivedEventHandler((S,E) => {

TxtStandardOutput.Text += $@"> {E.Data} {Environment.NewLine}";

Console.WriteLine(@"Output: " + E.Data);

});

Unfortunately, the threading issue returned, and I realised I would have to make use of delegate types.

This meant extracting the lambda expressions (=>) to their own functions and then using a delegate type to invoke them across threads

#region Read operations

delegate void OutputDelegate(object S, DataReceivedEventArgs E);

/// <summary>

/// https://stackoverflow.com/questions/10775367/cross-thread-operation-not-valid-control-textbox1-accessed-from-a-thread-othe

/// </summary>

/// <param name="S">the object that sent the event</param>

/// <param name="E">the data attached to the event</param>

private void ReadOutput(object S, DataReceivedEventArgs E) {

if (this.TxtStandardOutput.InvokeRequired) {

var OutputDelegate = new OutputDelegate(ReadOutput);

this.Invoke(OutputDelegate, new {S, E});

} else {

TxtStandardOutput.Text += $@"> {E.Data}{Environment.NewLine}";

Console.WriteLine(@"Output: " + E.Data);

}

}

/// <summary>

/// https://stackoverflow.com/questions/10775367/cross-thread-operation-not-valid-control-textbox1-accessed-from-a-thread-othe

/// </summary>

/// <param name="S">the object that sent the event</param>

/// <param name="E">the data attached to the event</param>

private void ReadError(object S, DataReceivedEventArgs E) {

if (this.TxtErrorOutput.InvokeRequired) {

// Invoke if needed

var OutputDelegate = new OutputDelegate(ReadError);

this.Invoke(OutputDelegate, new {S, E});

} else {

// We're on the thread so do what I need to do

if (E.Data != "") {

TxtErrorOutput.Text = $@"Error detected:

{Environment.NewLine}{E.Data}";

}

Console.WriteLine(@"Error: " + E.Data);

try {

\_DebugProcess.CancelErrorRead();

} catch (InvalidOperationException) { }

try {

\_DebugProcess.CancelOutputRead();

} catch (InvalidOperationException) { }

\_DebugProcess.EnableRaisingEvents = false;

try {

if (!\_DebugProcess.HasExited) {

\_DebugProcess.Kill();

\_DebugProcess.WaitForExit();

}

} catch (InvalidOperationException) { }

StopDebuggingInterfaceChanges();

}

}

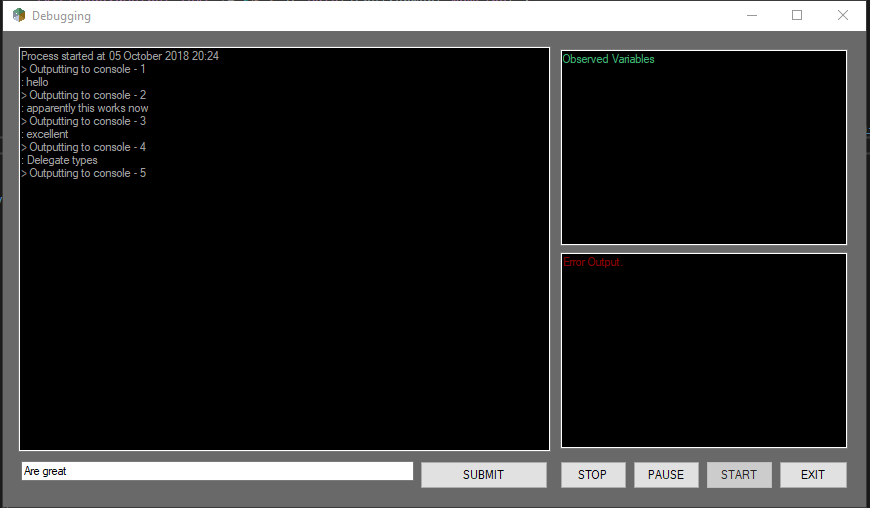
#endregion

I got a brief ParameterCountMismatch error, but it was fixed by changing the following line in both event handlers:

this.Invoke(OutputDelegate, new {S, E});

to

this.Invoke(OutputDelegate, new object[] {S, E});



It finally works! Next: Variable Outputs

#### Variable Observation

With observing variables, the hard part was figuring out how to send them. I considered using Piping and the COM but decided that these were far too time consuming. Instead I opted for a less robust solution that would take less programming overhead to produce: JSON.

[[“Name”,”Value”],[“Name”,”Value”],etc]

Having used JSON in the past for storage, I was familiar with the Newtonsoft.Json library (<https://www.nuget.org/packages/Newtonsoft.Json/>). I updated my test program to be more advanced:

using System;

using System.Collections.Generic;

using Newtonsoft.Json;

namespace TestProgram {

internal class Program {

private static bool Debugging;

private static int Count = 1;

private static void Main(string[] Args) {

try {

if (Args?[0] == "DEBUG")

Debugging = true;

} catch (IndexOutOfRangeException) {

Console.WriteLine("No arguments");

}

while (true) {

Console.WriteLine("Outputting to console - " +

Count);

if (Debugging) {

// Emit JSON

var JsonOut = JsonConvert.

SerializeObject(ObserveVariables());

Console.WriteLine($@"DAT|{JsonOut}");

}

Console.ReadLine();

// Wait for ENTER to be pressed

Count += 1;

}

}

private static List<object[]> ObserveVariables() {

// Get a list of all the variables that are being

// observed and their contents

return new List<object[]> {

new object[] {nameof(Debugging),

Debugging.ToString()},

new object[] {nameof(Count),

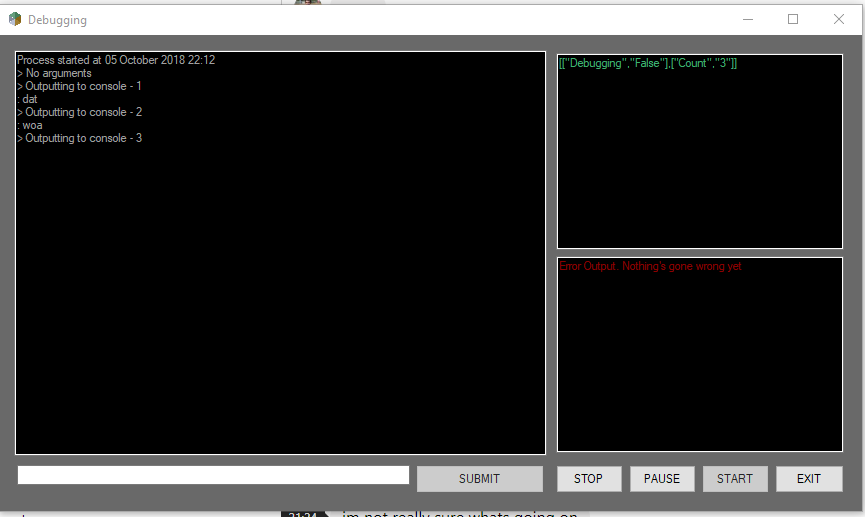
Count.ToString()}

};

}

}

}



Building the executable resulted in a usable template program that was able to output the contents of specified variables. Because the JSON is output through the console, all I needed to do was check if the last output began in “DAT|[” and react accordingly. From there it was a simple matter of cleaning up the string and formatting it to be user friendly:

private void OutputVariables(string Data) {

// Remove `DAT|[`

Data = Data.Remove(0, 5);

// Remove last `]`

var **VariableOut** = Data.Remove(Data.Length - 1);

// Remove `["`

**VariableOut** = **VariableOut**.Replace("[\"", "");

// Remove `"]`

**VariableOut** = **VariableOut**.Replace("\"]", "");

// Replace `","` with ` = `

**VariableOut** = **VariableOut**.Replace("\",\"", " = ");

// Replace `,` with a new line

**VariableOut** = **VariableOut**.Replace(",", Environment.NewLine);

**VariableOut** = "Variables:" + Environment.NewLine + Environment.NewLine + **VariableOut**;

/\* For the input `DAT|[["Name1","Value1"]["Name2","Value2"]]`,

\* you get

\*

\* `Variables:

\*

\* Name1 = Value1

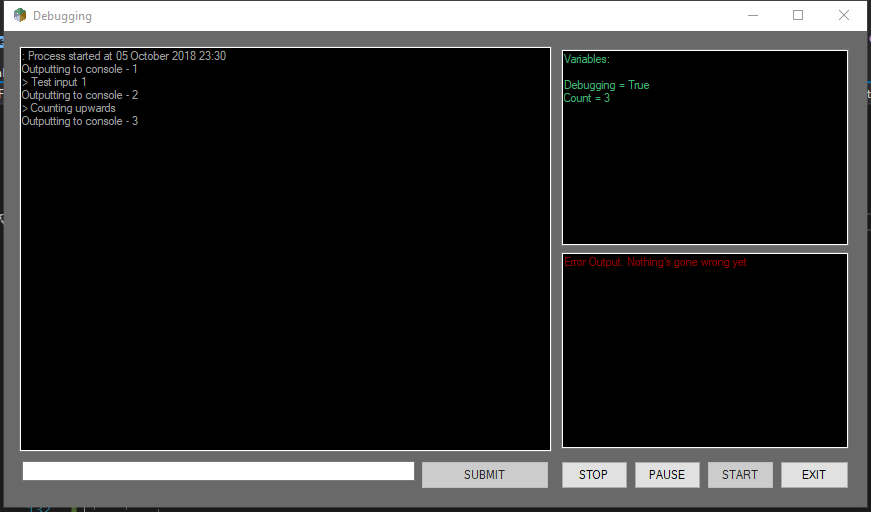
\* Name2 = Value2`

\*

\*/

TxtVariableOutput.Text = **VariableOut**;

}



The variables are displayed neatly in the top right of the screen

##### Testing

|  |  |  |  |
| --- | --- | --- | --- |
| Data (string) | Expected Output (string) | Actual Output (string) | Passed? |
| DAT|[[“Var1”, “true”],[“Var2”, “false”]] | Variables:  Var1 = true  Var2 = true | Variables:  Var1 = true  Var2 = true | Yes |
| DAT|[[“Var1”, “true”],[“Var2”, “false”],[“Var3”,“Hello world”] | Variables:  Var1 = true  Var2 = true  Var3 = Hello world | Variables:  Var1 = true  Var2 = true  Var3 = Hello world | Yes |
| DAT|[[“StringInput”, “DAT|[“[]]”]]”]] | Variables:  StringInput = DAT|[“[]]”]] | Variables:  StringInput = DAT|[]]] | No |

Because of how I coded it, if a variable contained any of the characters shown, it would display incorrectly. I had already imported the JSON library earlier however, so if I simply removed the “DAT|” prefix I could treat the string as JSON and use Newtonsoft.JSON to parse it. It was also at this time that I realised that I wanted to also display the variable types. Instead of using a dedicated list of observed variables I chose to use the System.Reflection methods to iterate through the User Program and find the variables’ names, values and types, then convert them into JSON before outputting them to the program.

##### Rebuilding

private static List<string[]> ObserveVariables() {  
  
 // Get a list of all the variables that are being observed and their contents  
 // using Reflection  
  
 var ReflectionInfo = typeof(Program).GetFields(BindingFlags.**NonPublic** |  
 BindingFlags.**Public** |  
 BindingFlags.**Static**);  
  
 var StringSet = new List<string[]>();  
  
 foreach (var FI in ReflectionInfo) {  
  
 StringSet.Add(new[] {FI.Name, (FI.GetValue(null) ?? "null").ToString(), FI.FieldType.Name});  
  
 }  
  
 return StringSet;  
  
}

This method iterates through the user program, finding the variables, types and values as it goes. It then returns a list of these values.

|  |  |  |  |
| --- | --- | --- | --- |
| Data | Expected Output | Actual Output | Passed? |
| <int> Count = 5 | List<string[]> = {  {“Count”, “5”, “Int16”}  } | List<string[]> = {  {“Count”, “5”, “Int16”}  } | yes |
| <int> Count = 5  <string> Name = null | List<string[]> = {  {“Count”, “5”, “Int16”},  {“Name”, “null”, “string”}  } | List<string[]> = {  {“Count”, “5”, “Int16”},  {“Name”, “null”, “string”}  } | yes |
| <int> Count = 5  <string> Name = null  <bool> Yes = true | List<string[]> = {  {“Count”, “5”, “Int16”},  {“Name”, “null”, “string”},  {“Yes”, “true”, “boolean"}  } | List<string[]> = {  {“Count”, “5”, “Int16”},  {“Name”, “null”, “string”},  {“Yes”, “true”, “boolean"}  } | yes |

private static CancellationTokenSource CTokenSource;  
private static CancellationToken CToken;  
  
/// <summary>  
/// The name of the program being debugged.  
/// Is the namespace of the program.  
/// </summary>  
private static string Name;  
  
private static void Main(string[] Args) {  
  
 CTokenSource = new CancellationTokenSource(); // Black magic  
 CToken = CTokenSource.Token; // Witchcraft and wizardry  
  
 Name = typeof(Main\_Program).Namespace;  
  
 try {  
 if (**Args**?[0] == "DEBUG") {  
  
 Task.Factory.StartNew(async () => {  
 while (true) {  
  
 if (CToken.IsCancellationRequested) return;  
  
 Console.WriteLine(@"DAT|" + JsonConvert.SerializeObject(ObserveVariables()));  
  
 await Task.Delay(250);  
  
 }  
 }, CToken);  
  
 Console.WriteLine($@"Debugging of program ""{Name}"" in progress");  
  
 }  
  
 } catch (IndexOutOfRangeException) {  
 **Args** = new[] {""};  
 } // If no arguments are supplied, this might not have worked...  
  
 // Run their code  
  
 Program.Main(**Args**);  
  
 // Finish up  
  
 CTokenSource.Cancel();  
  
}

This method creates an asynchronous task to observe the variables then output the data every 250 milliseconds when “DEBUG” is passed as an argument. It then runs the user’s program. I rebuilt this to use a second class so the user is not confused by all the asynchronous operations.

#### Code Overview

### The Designer

## The Blocks

### BaseBlock

### Chains

### Functionality

#### Start

#### Stop

#### If

#### Else

#### For

#### While

#### Random

## Running A Program

## Saving Your Work

# REFERENCES

|  |  |  |
| --- | --- | --- |
| Website Name | Web Address | Date First Accessed |
| Python Official Website | <https://www.python.org/> | 14/05/18 |
| Scratch Official Website | <https://scratch.mit.edu/> | 14/05/18 |
| Small Basic Official Website | <http://wwwsmallbasic.com/> | 14/05/18 |
| MicroBit Official Website | <http://www.microbit.org/> | 14/05/18 |
| Microsoft Docs | <https://docs.microsoft.com/en-us/dotnet/> | 14/05/18 |
| Stack Overflow | <https://stackoverflow.com/questions/355724/embedding-a-dos-console-in-a-windows-form> | 05/08/18 |
| Stack Overflow | https://stackoverflow.com/questions/10775367/cross-thread-operation-not-valid-control-textbox1-accessed-from-a-thread-othe | 05/10/18 |
| Microsoft Visual Studio Code | <https://code.visualstudio.microsoft.com/> | 05/08/18 |
| Emmet | https://emmet.io/ | 06/08/18 |
| Web Designer Depot | https://www.webdesignerdepot.com/2009/03/operating-system-interface-design-between-1981-2009/ | 01/10/18 |
| Wikipedia: History of the GUI | https://en.wikipedia.org/wiki/History\_of\_the\_graphical\_user\_interface | 01/10/18 |
| Harding University | https://view.officeapps.live.com/op/view.aspx?src=https://www.harding.edu/fmccown/gui/history-gui.pptx | 01/10/18 |