BASTILLE BURGLAR

Once upon a time...

There was a long lost horde of gold kept in dark dungeon far away from prying eyes. Two powerful mages became away of this stash of treasure and embarked their minions on journey to retrieve this gold, each hoping to gather more than the other.

Through their power of magic, these wizards had learnt how to command their minions through the use of programming...

My First Game

Bastille burglar is fundamentally a two player game, however for your first time, we'll ignore the other player!

To start with we will run the game and set the number of bots per team to 1, and leave everything else as is.

The game will boot up into the main screen, it has several components,.

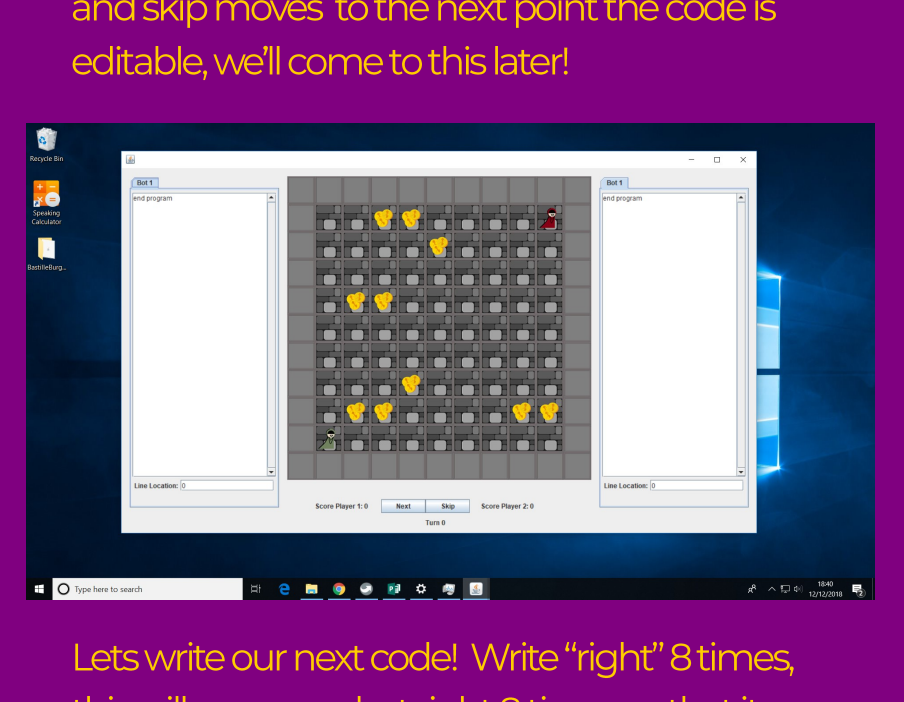
The tabs are where you write the code, the left hand player is number one, and the right hand number 2.

Each bot has a line location, this is where in your code the program is reading from.

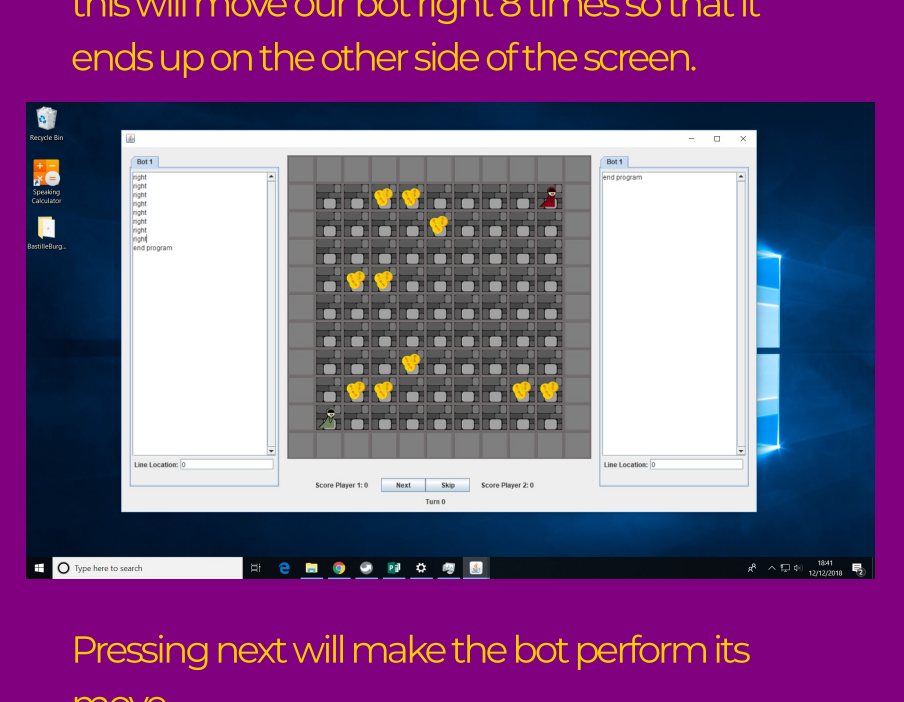
The map shows where the minions or bots are, and hovering your mouse over a bot will show you which player it belongs to.

The score is shown for each player with the current turn.

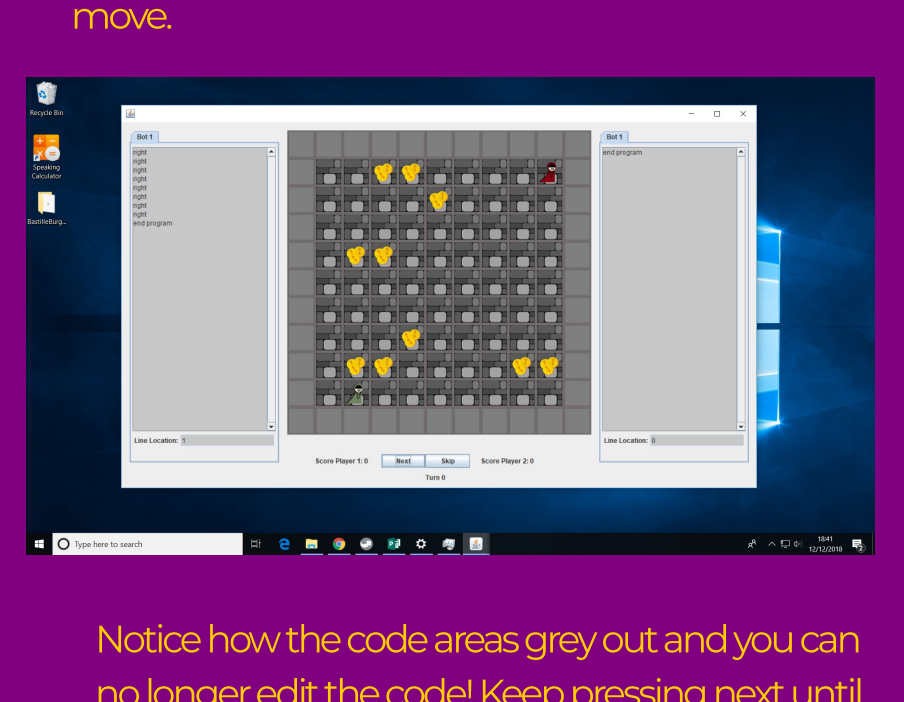
Finally the button next makes the next move and skip moves to the next point the code is editable, we'll come to this later!



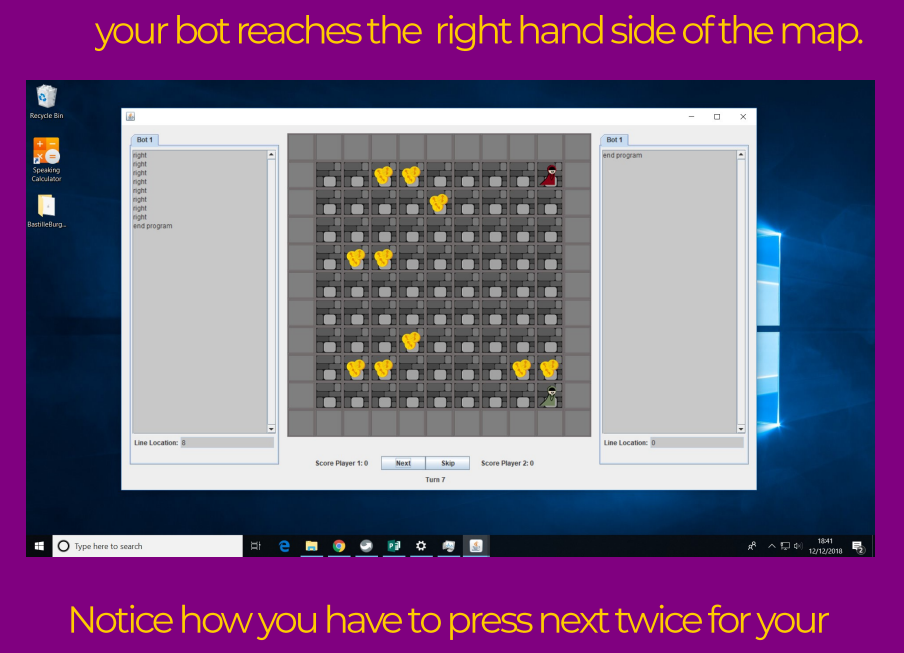
Lets write our next code! Write "right" 8 times, this will move our bot right 8 times so that it ends up on the other side of the screen.



Pressing next will make the bot perform its move.



Notice how the code areas grey out and you can no longer edit the code! Keep pressing next until your bot reaches the right hand side of the map.

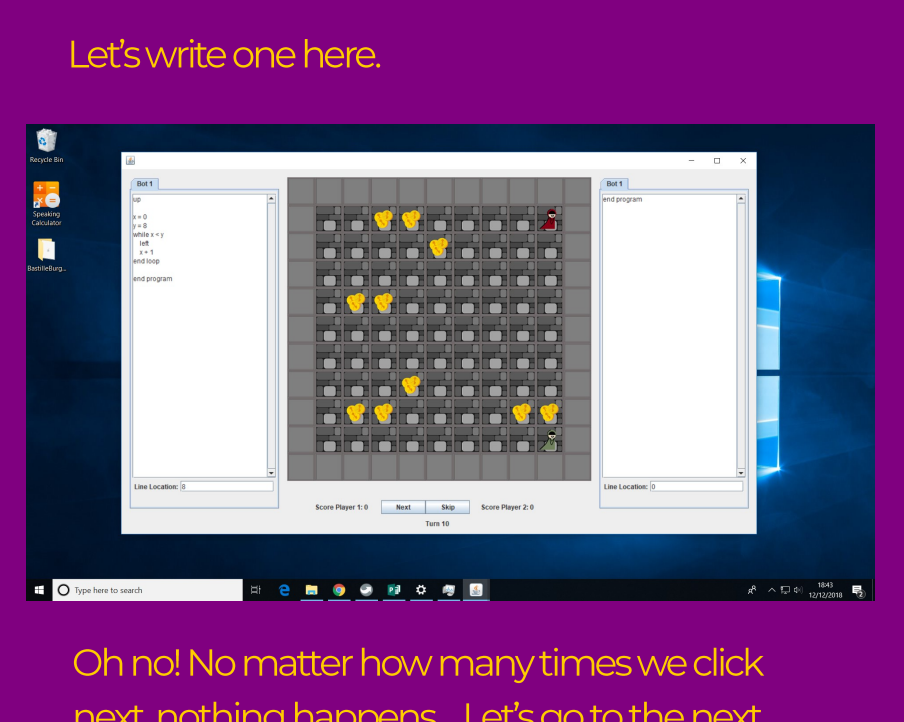


Notice how you have to press next twice for your bot to move, this is because on every other move it is the other players go, w just haven't programmed the other bot. Keep clicking next until the input area is white again.

Because the input area is no longer greyed out we can edit our code again, let's do this.

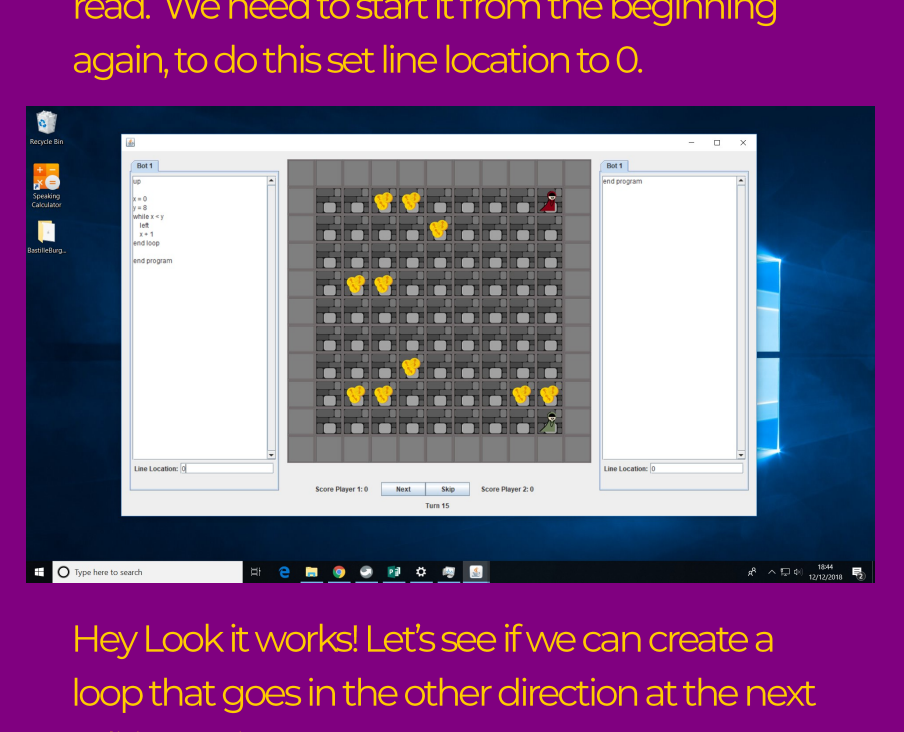
Wouldn't it be nice to not have to write "right" 8 times? It would, and that's why we have a loop. Loops provide a way of repeating things for a certain number of times.

Let's write one here.

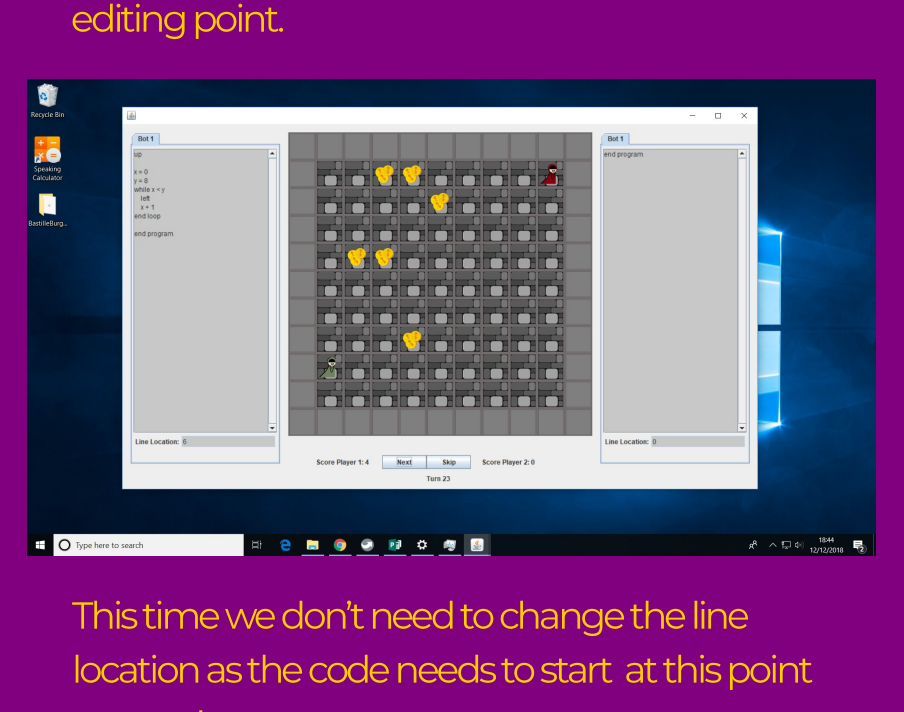


Oh no! No matter how many times we click next, nothing happens... Let's go to the next editing point and work out what's wrong.

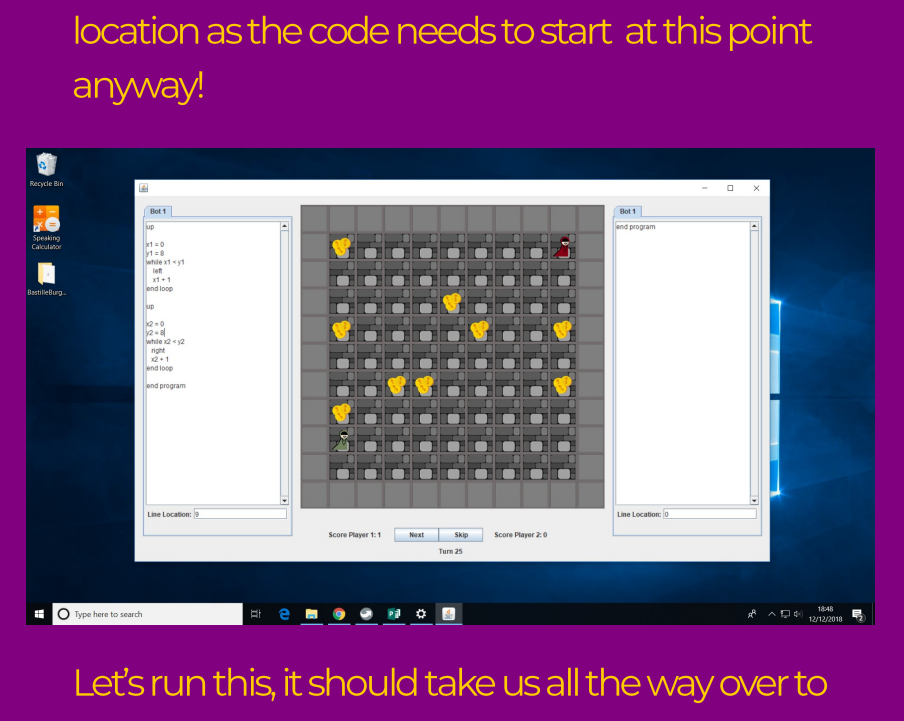
Notice that the line location is currently set at 8, this means that the line "end program" is being read. We need to start it from the beginning again, to do this set line location to 0.



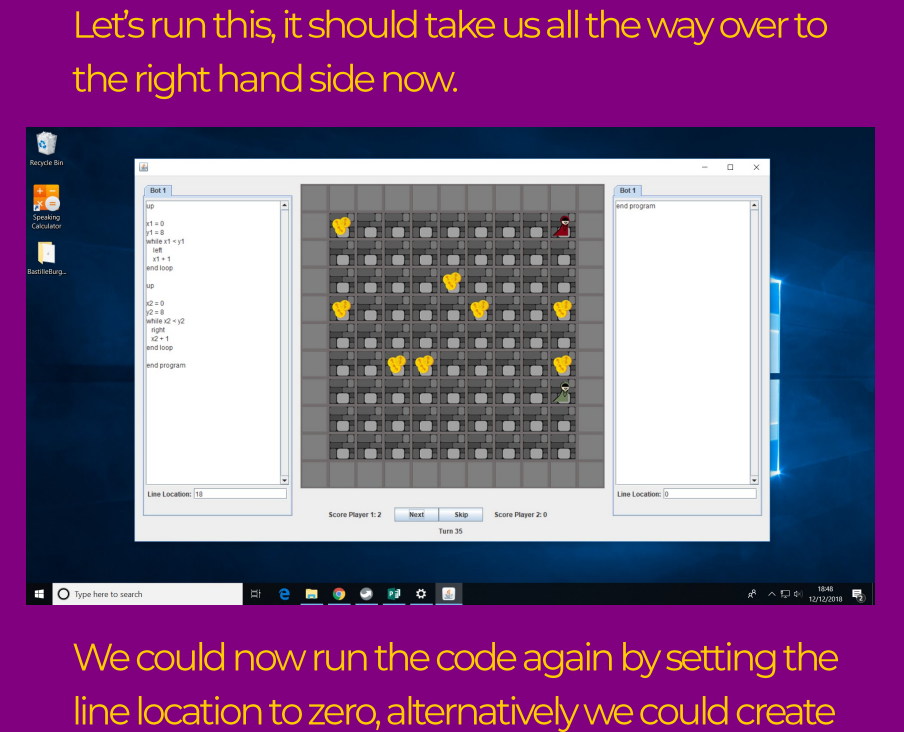
Hey Look it works! Let's see if we can create a loop that goes in the other direction at the next editing point.



This time we don't need to change the line location as the code needs to start at this point anyway!

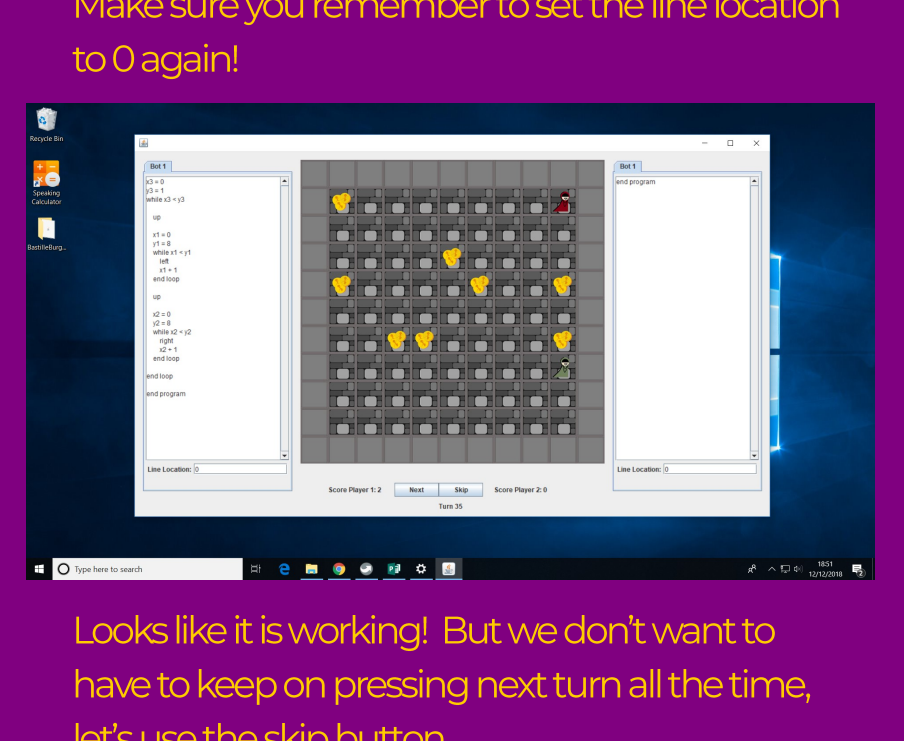


Let's run this, it should take us all the way over to the right hand side now.



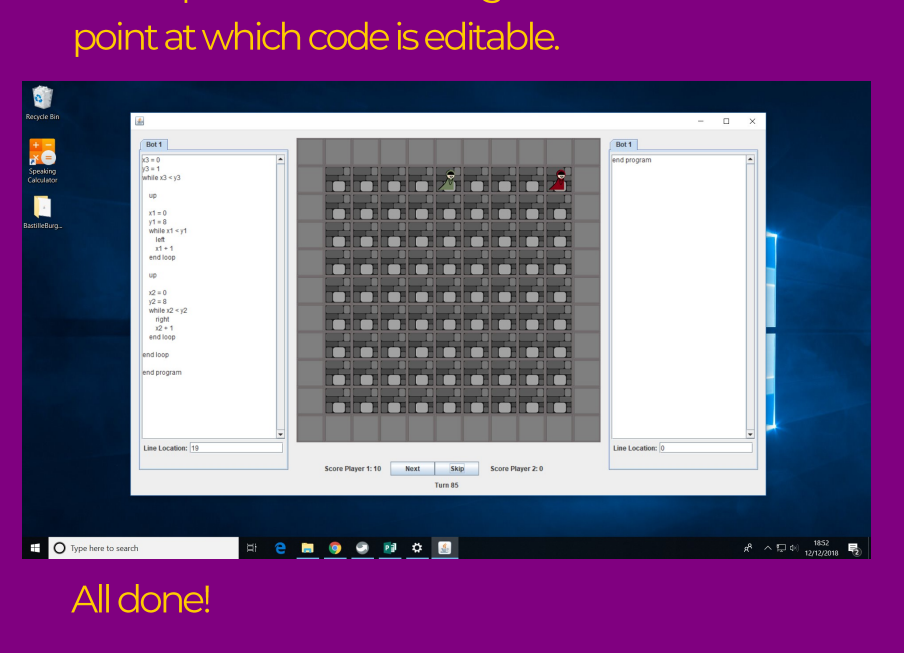
We could now run the code again by setting the line location to zero, alternatively we could create a nested loop to run the entire program continuously.

Make sure you remember to set the line location to 0 again!



Looks like it is working! But we don't want to have to keep on pressing next turn all the time, let's use the skip button.

The skip button takes the game to the next point at which code is editable.



All done!

Program Features

How does the programming work?

- In essence the game is an interpreter, this means it runs each line of your program until it finds a movement command, at which point the bot moves or idles.
- The line location is the next line the game will read.

What features are possible in the code?

- Nested loops and Nested if statements are all allowed, you can even have if statements in loops and vice versa.
- Variables can have any name, so long as the name does not have spaces.
- Numbers must be integers, but can be negative or positive.
- Any amount of white space can be inserted between words or before lines, this allows indentation if you want!
- Empty lines are also allowed, remember whitespace is useful.

Features of the game.

- The game allow you to change the number of turns between being able to modify code, a higher number makes the game harder, this is done in the title screen.
- The number of bots per player can be changed, but only to a maximum of 9!
- Skipping turns to the next modification point can speed things up.
- The number of coins can be changed too.

Bugs...

- Unfortunately as the game is in it's alpha stage there are three key bugs which can cause the game to crash or freeze, these are quite easy to have happen to you.
- Firstly syntactically incorrect programs will cause the game to crash.
- Secondly if you create an infinite loop the game will freeze.
- Thirdly if you try and call up a non-existent variable, the game will also crash.

Syntax Guide

The programming language for Bastille Burglar has lots of different functions you can perform, let's go over them. Please remember to use the syntax as presented, spaces between words are critical.

```
up
```
This is the up command, it moves the bot up.

```
down
```
This is the up command, it moves the bot down.

```
left
```
This is the up command, it moves the bot left.

```
right
```
This is the up command, it moves the bot right.

```
idle
```
This command the bot to not make a move.

```
# Any Comments
```
Single line comment, all words after a hash are ignored.

```
end program
```
This terminates the program, it is required at the end of all programs, or else the game will crash.

```
var = number
```
This sets a variable of any name to an integer also creating the variable if not already created.

```
var + number
```
This adds a number to a named variable.

```
var - number
```
This subtacts a number to a named variable.

```
while var1 < var2
    # Do stuff
end loop
```
This is the conditional less than while loop.

```
while var1 > var2
    # Do stuff
end loop
```
This is the conditional greater than while loop.

```
while var1 = var2
    # Do stuff
end loop
```
This is the conditional equal to while loop.

```
if bot in TL
    # Do stuff
end if
```
This is an if statement which checks for bots in a named tile. TL is the top left tile but other tiles are listed below.

TL	TC	TR
CL	Bot	CR
BL	BC	BR

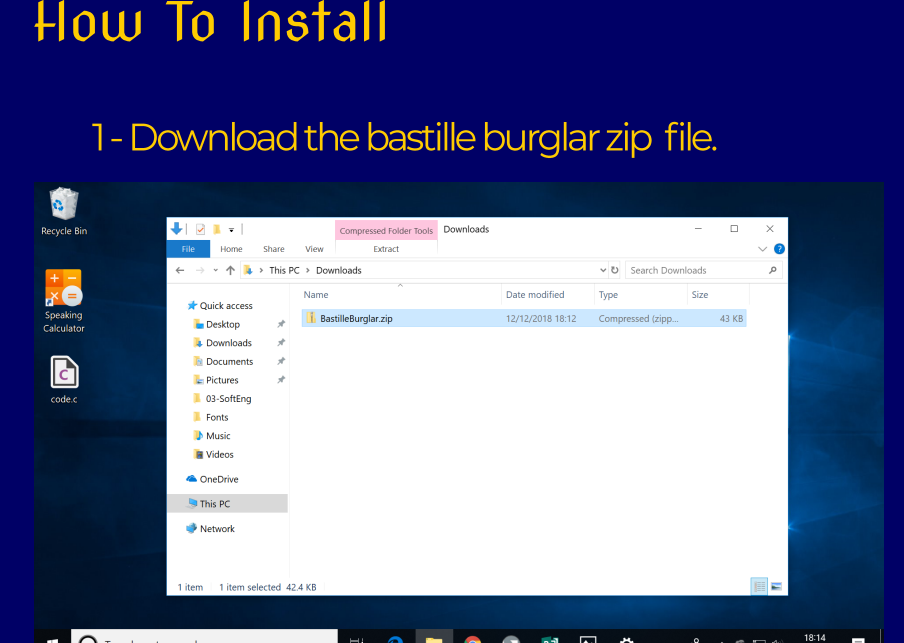
```
if var1 < var2
    # Do stuff
end if
```
This is the less than if statement.

```
if var1 > var2
    # Do stuff
end if
```
This is the greater than if statement.

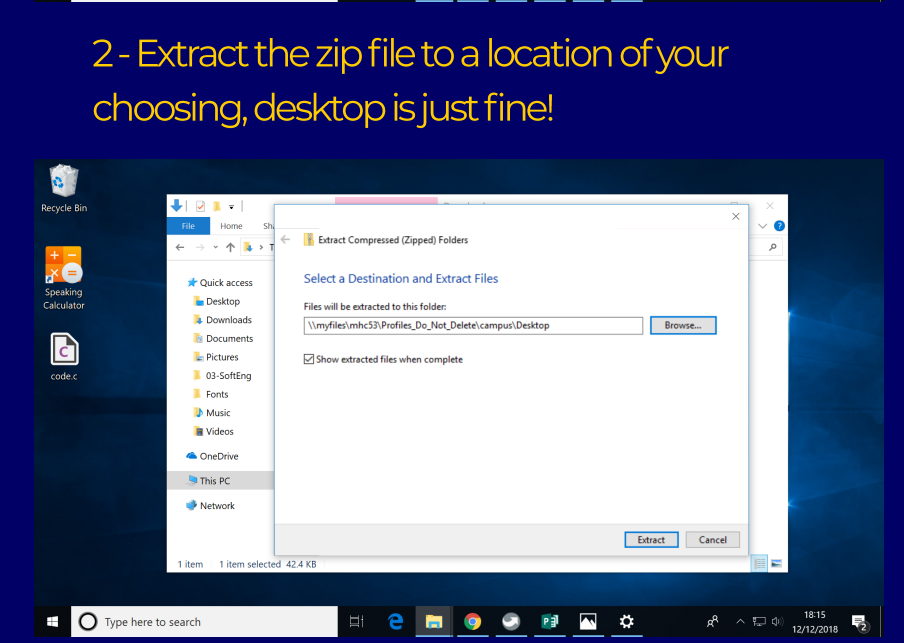
```
if var1 = var2
    # Do stuff
end if
```

How To Install

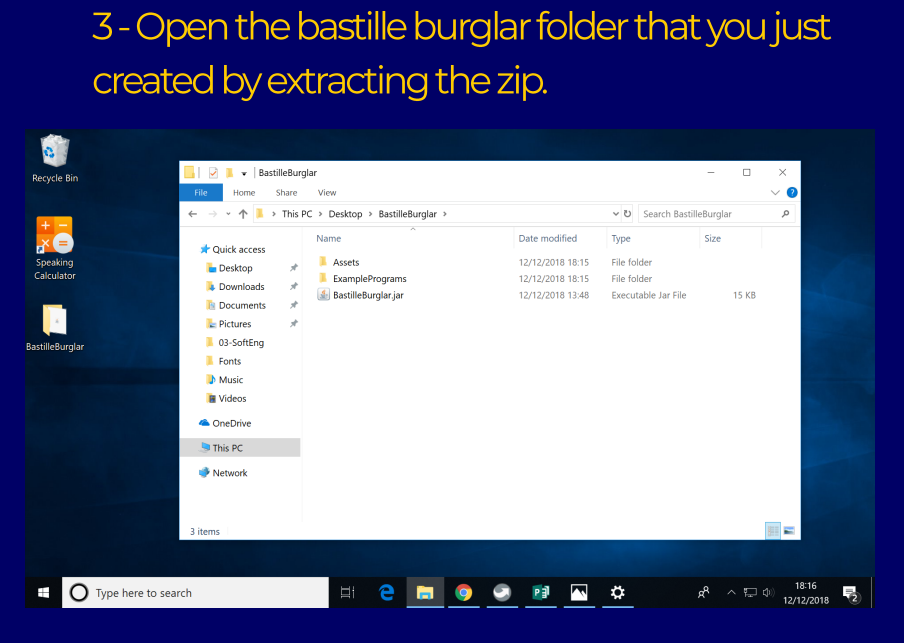
1 - Download the bastille burglar zip file.



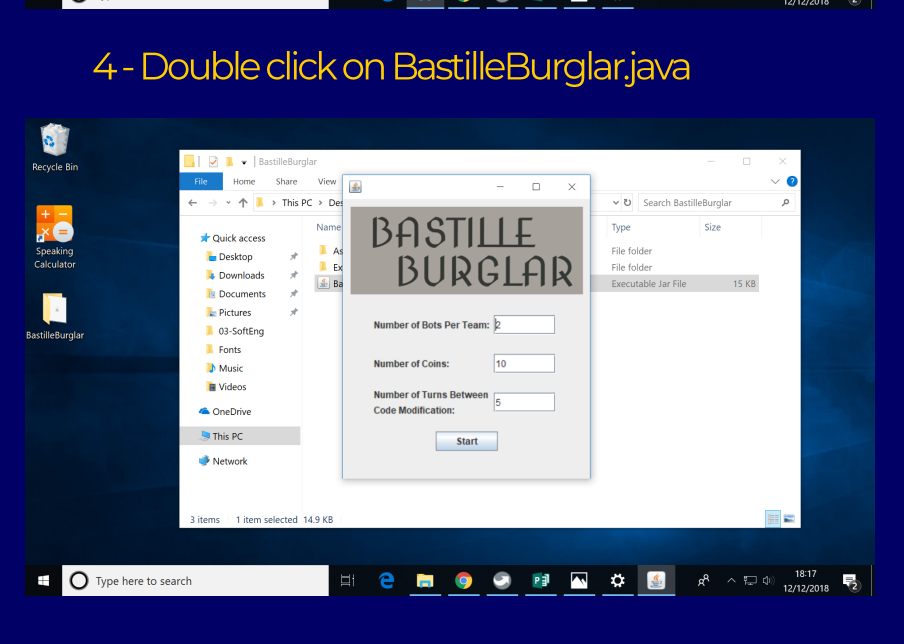
2 - Extract the zip file to a location of your choosing, desktop is just fine!



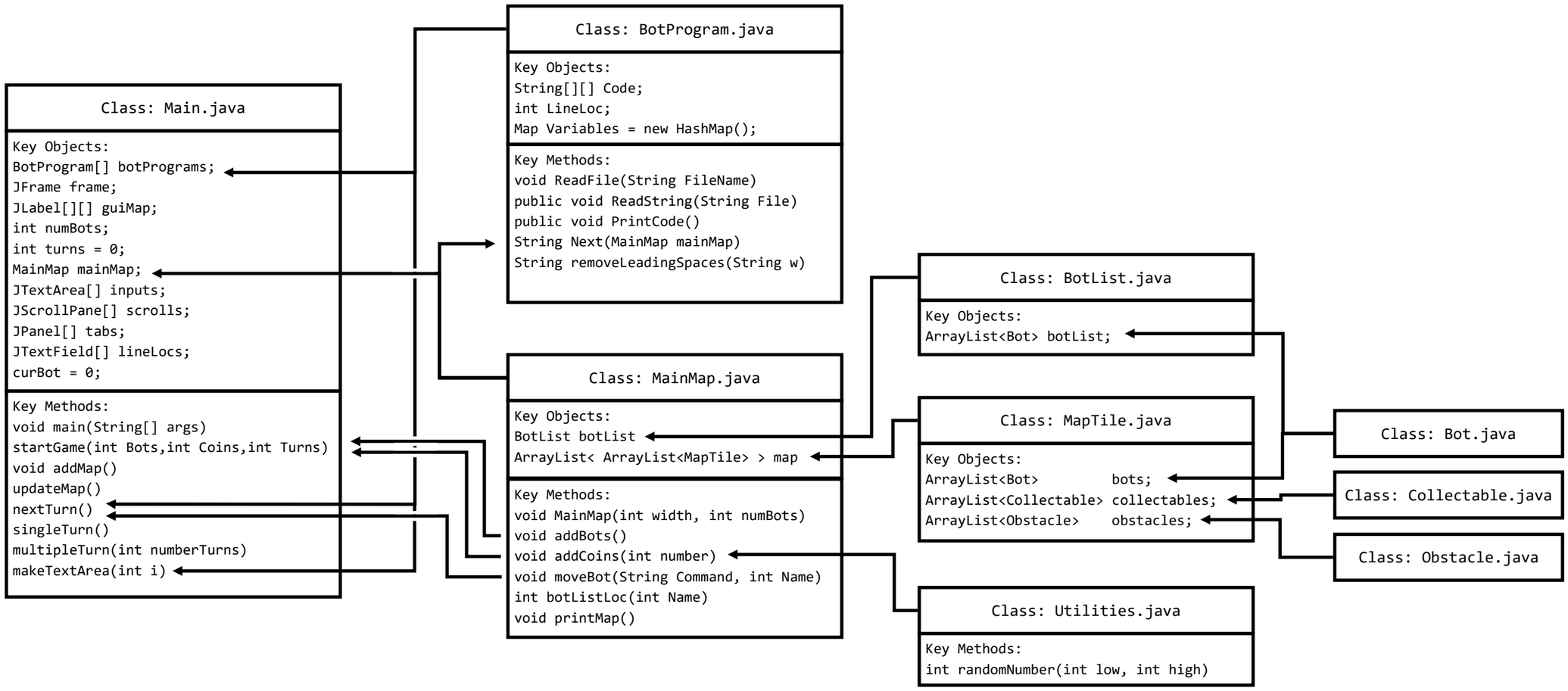
3 - Open the bastille burglar folder that you just created by extracting the zip.



4 - Double click on BastilleBurglar.java



Bastille Burglar : Program Architecture Diagram



Bastille Burglar

Maintenance guide

Contents

Introduction

This document, the maintenance guide, is written to provide a detailed description of all classes with all functions and key variables. This has been done so that you, the owner of the software have at least some chance of being able to either add extra functionality in the future or remove current bugs. This is especially pertinent as the current game is only at an alpha stage and has several critical bugs that can cause the game to either freeze or crash.

The document has been organised into the classes, with each class containing a list of all key objects and all functions. Some objects have been excluded from this guide as they are simple in their function and thoroughly commented.

This document is to be used in conjunction with comments on the code. The code comments often make short practical statements about how something functions or what it is, whereas this document provides the detailed rationale behind how things work and why they are hoe they are. This should help lead to a deeper understanding of the rationale of the code, which will help you implement either new features or fix bugs.

Furthermore a program architecture diagram has been created too; this is to assist your understanding in a more general way as to how each of the modules fit together, to create the overall game.

Hopefully with all three documents: the code comments, the architecture diagram and this, the maintenance guide, you should gain an in depth understanding of how the game works and why certain things were done.

Thank you very much on behalf of the whole team.

Class: `Collectable.Java`

The class is an object of collectables, the point of this is that the class does not necessarily have to be a coin. Currently only coin objects are initialised, but you could feasibly create for example a chest object too.

Key Objects:

`int score`	This is the score that the object contains, that is given to a bot that goes over it. Currently this is set to a score of one for coins, but could be expanded to any number, such that you could create a chest object with a score of 3.
`int type`	This is the type of collectable it is, currently this is 0 for a coin, but for example a chest might have a type of 1.

Class: `Obstacle.Java`

The class is an object of an obstacle. Currently only walls are allowed, but it would not be difficult to expand the calls to include trap objects, which kill bots.

Key Objects:

`int type`	This is the type of obstacle this is. Currently this is set to one to represent a wall, however this could be changed to add more obstacle types. For example you could have a trap set to type 1, so that if a bot tries to move over it, the bot dies.

Class: `Bot.Java`

The class represents a bot object, this contains information about every bot loaded.

IMPORTANT NOTE: Currently there are only two owners: 1 and 2. Owner one has all bots with an odd name (1,3,5…) and owner 2 has all bots with an even name (2,4,6…).

Key Objects:

`int mame`	The name is a unique identifier of the bot, this is used to paint and identify the correct bot within the map.
`int score`	This is the current score of the bot in the game. This is modified by the `MainMap` function
`int owner`	This is the player which owns the bot, the purpose of this is to identify the player who is in charge of the bot.
`int x`	The current x position of the bot, allows for fast location of the bot which is held in the map that was located by its copy in the `BotList`.
`itn y`	The current y position of the bot.

Key Methods:

`Bot(int name, int owner, int x, int y)`	This is the class constructor, as the name, owner x and y must be instantiated when the bot is created.

Class: `Utilities.java`

Simply contains the random function, though could be expanded to contain more functions in the future. The idea is to create a static container of useful functions for the game.

Key Methods:

`int randomNumber(`
`int low,`
`int high)`
	This simply provides a random integer between the integers of low and high inclusive.

Class: `BotList.java`

This class works in tandem with the `MainMap` class. One of the problems faced with simply having bots in the map object is that of how do we find a specific bot? To do this would involve having to search through every tile in the map, a very inefficient process.

One solution would be to not have a map at all, but then to move a bot would involve searching through every single other object to find their current locations.

Instead a compromise was found; first is to have a map with `MapTiles`, which contain all three objects of Bot, Collectable and Obstacle, second is to have list of Bots which contain copies of the bots in the map.

This means that when moving a bot, `MainMap` does the following sequence, first it looks for the bot with the required name in the `BotList`, and then it retrieves the current x and y position of that bot. The program uses the x and y indexes to directly call up the bot in the map object. Once it finds the bot in the map object it moves the bot in the map and simultaneously updates the x and y positions in the `BotList`.

Key Objects:

`ArrayList<Bot> botList`	An array of Bot objects. Using an array list allows the list to be of variable size.

Class: `MapTile.java`

This is an object that represents a singular tile of the game map. It contains three variable sized arrays containing the three different map objects.

VERY IMPORTANT NOTE: Although in theory the game could have more than one object per tile, this in practice has not been implemented. Using array lists allows this, however the MainMap, BotProgram and Main classes are built around the assumption that only one of any object is every in a map tile. This has led much of the code not asking what type of object is in a tile but simply is there an object in a tile. Consequently if there is to be more than one object in a tile of a type, significant alterations will need to be done.

Furthermore an empty tile is simply represented by having nothing in any of the lists.

Key Objects:

`ArrayList<Bot>`	ArrayList of the bots on the tile.
`ArrayList<Collectable>`	ArrayList of the collectables.
`ArrayList<Obstacle>`	ArrayList of the objects.

Class: `MainMap.java`

This class creates the map object and handles almost all of the backend for the game. It is responsible for manipulating the map object, moving about bots for a given turn and working out scores.

Key Objects:

`ArrayList< ArrayList<MapTile> > map`	A two dimensional ArrayList of `MapTiles` representing the game map.
`BotList botList`	The `BotList` for the map. This holds a copy of all bots in the map object, for more details of why this is done see `BotList`.
`int size`	This is simply the size of the map, to ensure it has a value to satisfy the compiler, this is set to 0.
`int numBots`	This is the total number of bots in the map.

Key Methods:

`MainMap(int width, int numBots)`	This is the constructor, it does two processes to prepare the map. Firstly it creates an empty map which has a square size two larger than the input width, this is to allow for a border of walls. Secondly it then sets the outside edge of tiles to have a wall in them by creating an `Obstacle` object. Furthermore it sets the number of bots to be loaded. There is no reason why obstacles could not be added in different positions if needed, however not having an wall border can cause `BotProgram`'s `if statement` to try and access an out of bounds array value, causing the game to crash.
`void addBots()`	This function actually adds the bots to the map. What it does is it iterates from 0 through to `numBots`, then on even numbers it adds bots to player1 giving it an odd numbered name, then on even numbers it does a similar thing to player2. For more details see the `Bot` class. Player1's bots are added from the bottom left tile stacking up, and player2's bots from the top right, stacking down. Finally bots are added simultaneously to both the map and the bot list, for more details see the `BotList` class. This is the function to modify if the placing of the bots needs to be changed at all. For example you might want to spread them evenly over the left and right sides.
`void addCoins(int number)`	This adds the specified number of coins to the map in random locations. It pulls the `randomNumber` function from `Utilities` to do this task. It is structured so that it does not add a coin on to any currently loaded object including other coins. This is the function to change if you want to change how the coins are positioned within the map.
`void moveBot(String Command, int Name)`	This function is to move a bot in the map, its flow is done like this. • It creates two vectors, the bots current position, `curX` and `curY`, and the bots movement vector, `moveX`, `moveY`.

	• It assigns values to the move vector based on the input command. The input command is still a string of a word, this is legacy from previous versions and should be changed at some point. • It then gets the named bot's location and sets the current vector to this. • The program then checks that the move to be attempted is actually allowed, as moving the bot into another bot or wall is not allowed. • If the move is allowed, the function then proceeds to move the bot into the new tile in the map and update the x and y values of both the bot in the `map` and the bot on the `BotList`. • If the bot has moved onto a tile with a `Collectable`, the function deletes it and adds the score to the score of the bot, both in the map and bot list. • Finally if the bot's move is invalid the program prints invalid move to the command line, and does not move the bot. The printing is part of an uncompleted feature which would have given each player a command line output to identify bugs, see BotProgram for more details.
`int botListLoc(int Name)`	This function finds the location of a bot in the BotList given its name, see the class for the bot list for more details as to why this is done.
`printMap()`	This function prints out a command line version of the map, the primary purpose of this is as a diagnostic for maintenance to ensure that the correct backend for the map is working minus all of the GUI wrappers. Furthermore the cod is simple and easy to modify, which would be useful tool if you want to modify the code. The code works by iterating through each `MapTile`, and printing a # if there is more than one `Obstacle` in the tile's list, a $ if there is more than one `Collectable`, a space if the tile is empty and the first `Bot` in the tile's list's name. The function is essentially a stripped down version of `updateMap` in `Main`.

Class: `BotProgram.java`

This is probably the least transparent of all of the classes, consequently I have provided an example of the program flow on an example program after describing each of the methods and objects, see Example Program Run.

The class is quite simply an interpreter for the code written by the players, its purpose is to read a player's code then out put the next command for the bot in a given turn.

IMPORTANT NOTE: the language in which the player's code is NOT a standard language, but invented by ourselves to be easy to read and use, additionally its functionality is intentionally very specific and restricted.

Key Objects:

`String[][] Code`	This is a 2D array that represents the user's code. Each word is held in a "line" which is the sub array, then each line is held in the "code" which is the main array.
`int LineLoc`	This is the line location, which is simply the current line in the code which is being read.
`Map Variables`	The game's programming language is capable of storing integer variables, hence somewhere is required to store the variables. The hash map does this by allowing the class to call up variables by name. It functions much the same as a dictionary would in other languages such as python.
`int botName`	This is so that a particular program can be identified to a particular bot. See `Bot` for more details.

Key Methods:

`void ReadFile(String FileName)`	This function reads a user's program from file and parses it into the `Code` array. It starts by working out how many lines there are in the file and uses this to work out the size of the array required. It then proceeds to split each line in the code into words which it put's in each lines array. Not only does the function ignore arbitrary length whitespace, it also removes leading whitespace, including tabs using the `removeLeadingSpaces` function. This function is now deprecated, but kept for either diagnostics or future use.
`void ReadString(String File)`	This function does exactly the same thing as `ReadFile`, however it reads a string into the array as opposed to a file. This is the current version of the function in use as the current game reads straight from the GUI.
`void PrintCode()`	This is a purely diagnostic function, the purpose of this is to print out the code that is in the array to validate that the code has been loaded correctly. This is useful for debugging.
`String Next(MainMap mainMap)`	This function is very complex and has lots of constituent parts, consequently it is described in its' own section, see `Next()`.
`String removeLeading Spaces(String w)`	This function is taken from Lokesh Gupta 2018, Java remove leading whitespaces from String: https://howtodoinjava.com/java/string/remove-leading-whitespaces/ Its purpose is to remove leading whitespaces from a string and then to return that without the whitespace. This was done as simply using split on a regEx fails to remove leading whitespace.

Function: `Next()`

The function `String Next(MainMap mainMap)` has been given entirely its own section in the document as it is very much a black box with lots of components that need explaining.

Furthermore the function has a strange and potentially difficult to understand manner of program flow.

The fundamental purpose of the function is this: read the input program from the user, loaded as an array by previous functions in `BotProgram`, and output the next command for the bot.

The function however has three core bugs are as follows, firstly if a syntactically incorrect code is provided, the game can quickly crash, secondly if the user writes code with an infinite loop the program will simply freeze up. Consequently the game firstly requires either a syntax checker to prevent incorrect code being written, or requires a modification to this function to make it more robust. Secondly this needs a cap on the number of time it can run for instance 1000, this would mitigate the consequences of an infinite loop. Finally trying to call up a non-existent variable will also cause the program to crash, this needs a catch to prevent this happening.

Function Flow:

The function exists in a continual infinite loop.

- The function first starts by reading the current `lineLoc` or line location in the user's program.
- It then reads the line at which `lineLoc` points to.
- Then it checks the line to see if it matches a series of different cases. If the line does indeed match one of these cases or "if statements" as they are implemented, the program then performs what is inside the statement.
- Inside a statement the function can do several things:
    - If a command is seen for example `"down"`, the function will return the string `"down"`, halting the loop by terminating the function. Prior to this `lineLoc` is increased by one, so that the next time the function is run it pucks up from the next line beyond the current line.
    - If a non-command is seen the program performs the content of the statement before moving `lineLoc` onwards by one, the program then proceeds to start the loop all over again reading the next line.
    - If and while loops are coded largely the same way, if these are seen the program will check to see if the specified value is true. If it is not the program moves `lineLoc` one beyond the end statement for this loop, if not then it proceeds to only move the `lineLoc` onwards by one, so that the next loop of the function executes what is inside the loop
    - While loops do something extra, once they see their end loop, the while loops `lineLoc` back to the line containing the while.
    - If the command `"end program"` is seen the function does not progress `lineLoc` by one and returns the command `end`. This means that the loop terminates yet the next time the function is run it still reads `"end program"`, hence never doing anything else.
- Finally if the function fails to read anything then `"Syntax not recognised at lineLoc"` is then printed. The printing is for an undeveloped feature whereby players would have their own command lines. The feature is buggy as often instead of reaching this point, the code will try and access an array value out of index causing it to fail. To fix this you could feasibly pad out each line in the `Code` array with a null character, or by creating a syntax checker,
- The program then proceeds to the next loop of the program if a command has not been returned, this is the reason why the code can fail if a player writes code with an infinite loop, as the code never terminates. As stated earlier a loop cap should be implemented.

If Statements:

`"print"`	This statement is intended for use in an undeveloped feature whereby each player would have their own command line, currently prints out a word to the command line.
`"printvar"`	Similar to the previous, this prints out the named variable.
`"up"`	Returns `"up"` whilst pushing `lineLoc` on by one.
`"down"`	Similar to above.
`"left"`	Similar to above.
`"right"`	Similar to above.
`"idle"`	Similar to above. Returns `"idle"` causes the `moveBot` function in `MainMap` to do nothing, as this function does not change `moveX` or `moveY` if neither of the other commands is input
`""`	An empty string just causes the program to move `lineLoc` on by one. This is to allow white space.
`"#"`	Allows for comments as the line is ignored by moving `lineLoc` on by one.
`"end" "program"`	Ends the program by keeping `lineLoc` in the same location and returning `end`.
`string "=" int`	Assigns an integer to a string in the hash map `Variables`.
`string "+" int`	Adds the integer to the specified variable.
`string "-" int`	Subtracts the integer from the specified variable.

`"while" string "<"|">"|"=" string`	This function performs a while loop.
	• Checks to see if the specified condition is true by comparing named variables against each other.
	• If specified condition is true
	  • `lineLoc` increases by one.
	• Else
	  • `lineLoc` is increased until the number of `"while"`s seen is the same as the number of `"end" "loop"`s seen, this allows for nested loops.
	  • It then goes one line further, so that the function reads the line after the end of the loop.

`"end" "loop"`	This moves `lineLoc` back up to the top of the while loop by doing the same thing as the while loop but in reverse by waiting for parity of `"while"`s and `"end" "loop"`s seen.
`"if" "bot" "=" "TL"|"TC"|"TR"| "CL"| "CR"| "BL"|"BC"|"BR"`	This function checks to see if a bot is in the specified tile with TL representing the top left tile etc. It firstly reads the `MainMap` passed to the function and uses the `botName` to work out if another bot is actually in the tile. It then proceeds in exactly the same way as the `while` loop, simply looking for `"if"`s and `"end" "if"`s to skip if need be.
`"if" string "<"|">"|"=" string`	It functions in exactly the same way as the `while` loop, simply looking for `"if"`s and `"end" "if"`s to skip if need be.
`"end" "if"`	Unlike `"end" "loop"` this simply moves `lineLoc` on by one as an if statement is not a loop and does not need to return to the top once it is done.

Example Program Run

Here I provide a short example program to demonstrate the flow of the function `Next()`.

Input String:

```
x = 0
y = 2
while x < y
    down
    x + 1
end while
right
end program
```

Function `ReadString()` then parses this program into the array, `Code`.

```
0 ['x', '=', '0']
1 ['y', '=', '1']
2 ['while', 'x', '<', 'y']
3 ['down']
4 ['x', '+', '1']
5 ['end', 'while']
6 ['right']
7 ['end', 'program']
```

`lineLoc` is set to `0`, function `Next()` is run:

Line 0 is read.
Variable `x` is put into map `Variables` with value `0`.
`lineLoc` is increased.
`Next()`'s loop begins again, line 1 is read.
Variable `y` is put into map `Variables` with value `2`.
`lineLoc` is increased.
`Next()`'s loop begins again, line 2 is read.
`while` is read, `x` is `0`, `y` is `1`, hence condition is true.
`lineLoc` is increased.
Line 3 is read.
`down` is read causing `Next()` to return `"down"`.
`lineLoc` is increased.

`lineLoc` is now `4`, function `Next()` is run:

Line 4 is read.
Variable `x` has `1` added to it, `x` is now `1`.
`Next()`'s loop begins again, Line 5 is read.
`end while` is read.
This causes `lineLoc` to return to 2.
`Next()`'s loop begins again, Line 2 is read.
`while` is read, `x` is `1`, `y` is `1`, hence condition is false.
This causes `lineLoc` to proceed to 6.
`Next()`'s loop begins again, Line 6 is read.

`right` is read causing `Next()` to return `"right"`.
`lineLoc` is increased.

`lineLoc` is now 7, function `Next()` is run:

Line 7 is read.
`end program` is read causing `Next()` to return `"end"`.

`lineLoc` is now 7, function `Next()` is run:

Line 7 is read.
`end program` is read causing `Next()` to return `"end"`.

`lineLoc` is now 7, function `Next()` is run… this ends up happening no matter how many times `Next()` is run, which is the point.

Class: `Main.java`

This is the main function, which runs the entirety of the program. Almost all of the code here is about running the game and creating the GUI.

The notion of what a turn is, is very important to the functioning of the game, on the GUI a turn is described as when all of the individual bots have made their move, however internally and henceforth a turn is referred to as the step taken for a single bot to move, hence the turns in the GUI really take turns*number of bots.

Key Objects:

`BotProgram[] botPrograms`	This is a list which contains all of the bot programs for the bots, this is indexed in the same way as the `BotList`.
`JFrame frame`	This is the main frame for the GUI in which everything is placed for the game, note this is separate from the title frame.
`JLabel[][] guiMap`	This is a 2D array of labels representing the map, this is how the map is displayed in the GUI.
`int numBots`	This is the number of bots in the game per player.
`int turns`	This is the current turn of the game, this is instantiated as zero as the number of turns must be zero before the game starts.
`modTurns`	This is the number of turns between which players can modify their code.
`MainMap mainMap`	This is the game's map from the `MainMap` object.
`JTextArea[] inputs`	This is an array of all of the text areas, this is because each bot requires a single text area to put in the code. The text areas are held in an array so that they are always accessible.
`JScrollPane[] scrolls`	This is an array of scroll panes to hold each text area, this is so that the text areas scroll.
`JPanel[] tabs`	This is an array of panels which become the tabs, this is done so that the line location text area can be added.
`JTextField[] lineLocs;`	This is an array of text fields for the line location inputs, this is attached with the scroll panes in the tabs.
`curBot`	This is the current bot, this keeps track of what bot's turn it is.

Key Methods:

`void main(String[] args)`	This is the main function, this is where the game runs from. • Creates a title screen where players can enter three specified values: the number of bots per team, the number of coins in the map and the number of turns between which they can modify their code. • Creates a button and listener which runs the `startGame()` function which boots up the game.
`startGame(int Bots, int Coins, int Turns)`	This games boots up the GUI for a game and runs the game. • Instantiates all required objects. • Creates buttons and listeners for a single and multiple turn. The single turn listener runs `singleTurn()`, multiple turn runs, you guessed it, `multipleTurn()`. • Creates the tabbed panes for the input fields and adds the panels to them by running `makeTextArea()`. • It then runs `addMap()` which adds the map. • It then runs `updateMap()` which performs the first update, adding the required icons to the tile of the map.

`addMap()`	The method is simply to instantiate all map the required map tiles as labels with the correct positions and dimensions.
`updateMap()`	This essentially paints all of the map tiles onto their respective labels. It iterates through each map tile, assigning an icon and potentially a tool tip text. • If it sees that the `MainMap`'s `MapTile`'s arrays are empty it adds an empty tile icon. • If the tile's wall array is not empty it adds a wall icon. • If the tile's coin array is not empty it adds a coin. • If the tile's bot array is not empty, it calls up the first bot and if the owner is player1 it adds a green burglar, if it is player 2 it adds a red burglar. Furthermore it adds a tool tip to the tile saying what bot it is and who owns it.
`nextTurn()`	This runs the backend part of the next turn of the game by moving the current bot according to its program then increasing the turn number by one and the current bot number by one. Once all of the bots have had their turn, it sets the current bot number back to zero.
`singleTurn()`	This wraps up all of the code required to run a turn of the game. • Goes through each bots code that has been input by the user and the line locations input by the user, reads them and sets them to the actual values for the `BotProgram` by running `ReadString` and setting `lineLoc`. • Furthermore unless the turn is a turn in which the players can modify their code, the inputs are set to not editable and greyed out. • Runs the `nextTurn()` function, updating the map internally. • Runs the `updateMap()` function, repainting the map according to its new state. • Reads the current turn and puts this on the applicable label, note this prints the number of turns divided by the number of bots, see the top of the class description for why this is. • Reads the scores from each player's bots and totals them, then sets these to their respective labels.
`multipleTurn()`	This simply auto runs `singleTurn()` skipping until it gets the next point at which players can modify their code.
`makeTextArea()`	This is responsible for assembling a single panel of the code input areas for a given index, `i`. • Creates a new panel. • Creates a new text area; this is the code input area. • Sets a default text to the text area of `"end program"` so that the next turn can be run without causing the game to crash straight of the bat. • Creates a new scroll pane. • Creates a new text field; this is the line location input area. • Adds all of this to the tab. • Finally it instantiates the bot program, as this is a convenient place to do so.