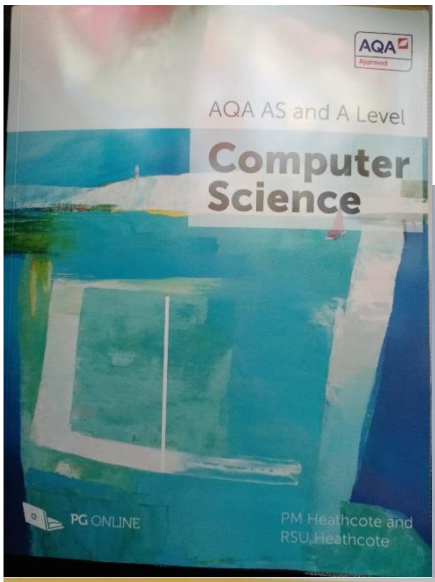
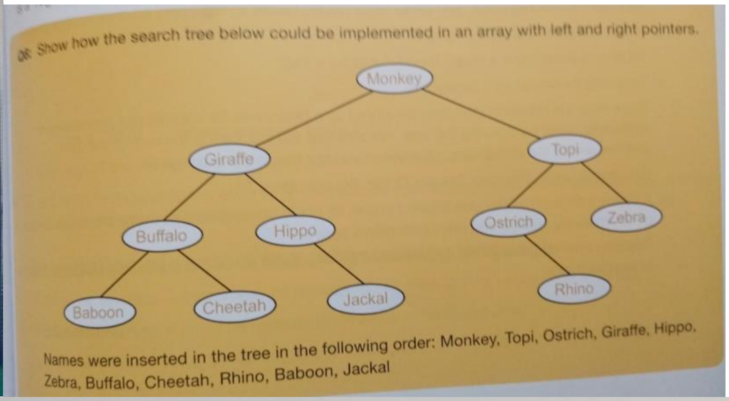
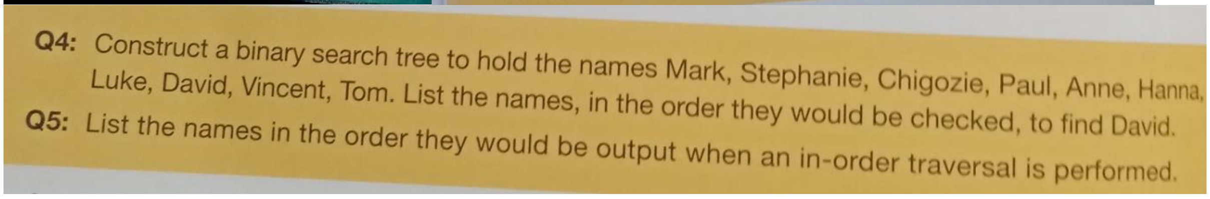
<u>Analysis</u>

I intend to develop a desktop app that can be used to perform mathematical calculations as well as allow a user to quiz themselves on topics by having the program generate random questions relating to an A-level maths or computer science topic, such as vectors, polynomials, differentiation, integration and binary search trees, and some dual-question topics for a difficult mode, such as polynomial questions involving kinematics. The demographic who will benefit from my app is typically going to be A-Level maths and computer science students, as well as people who want to strengthen knowledge in programming data structures and mathematical practices, like how binary search trees work and how to differentiate. Other potential end-users could be people who work in IT or data analysis careers or want to go into those and thus want to sharpen skills like observation (with binary search tree questions) and precision/error checking (following mathematical practices to find solutions and check desired outcomes). Below is a picture from an A-Level computer science textbook (cover picture included for reference) with some examples of potential questions that could be used (the ones shown below relate to binary trees):




Q6: Show how the search tree below could be implemented in an array with left and right pointers.

Names were inserted in the tree in the following order: Monkey, Topi, Ostrich, Giraffe, Hippo, Zebra, Buffalo, Cheetah, Rhino, Baboon, Jackal

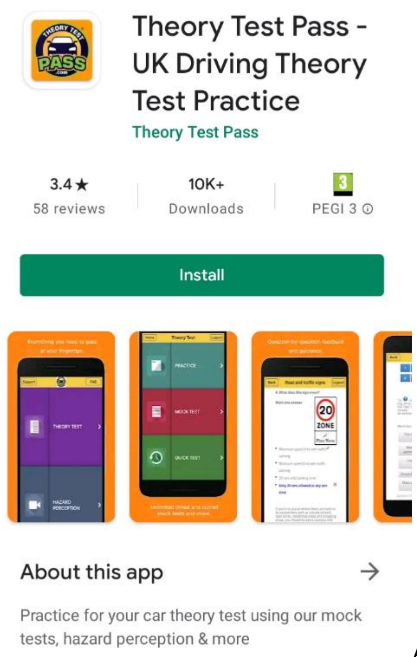

Q4: Construct a binary search tree to hold the names Mark, Stephanie, Chigozie, Paul, Anne, Hanna, Luke, David, Vincent, Tom. List the names, in the order they would be checked, to find David.

Q5: List the names in the order they would be output when an in-order traversal is performed.

The computer science A-Level textbook is useful for practice at questions, however it doesn't have many in it and if you want to practice a topic, you will need to have paper to write on typically, as well as take the rather thick book with you if you are travelling. There are online versions of textbooks which mitigate this problem of transporting a large book, however they cost money unless your school has access to them whereas a physical book can be borrowed from your school library, and the online textbook may only be accessible online and not always viewable offline. An app which can randomly generate a seemingly infinite number of questions on various topics offline as well as provide a simple interface for answering questions would be an apt solution to the disadvantages posed by revising from textbooks.

The other use of textbooks is to learn how to do new things, like formulas for solving equations, or an explanation of how to solve a process, like performing an in-order traversal. Practising skills with questions is the best way to learn to do something but sometimes you need help in checking if you are doing things correctly, like needing to check the answer section of your maths textbook to see if you are getting the right answers or using a calculator to solve a quadratic equation to be sure that you are using the formula correctly. Not all textbooks will have answers to their questions, like the computer science textbook which lacks answers for some questions; no textbook can instantly solve a calculation either: you need a calculator for that, and even then, more sophisticated questions can't be solved simply using a calculator. A solution to these issues is to have a section of the app which shows you how the equations/processes work and then serves as a calculator as such, allowing you to enter inputs and apply formulae to find an output, e.g. to check if you are applying the Newton-Raphson method correctly, you could enter the initial estimate for the polynomial and click a button to generate the closest root found using the method.

The maths textbook is a good resource to rate and criticise however it isn't a functioning system on a piece of technology, so I decided to do some research into quiz apps and revision tools, in particular, for phones. I found a driving theory test app which allows users to take tests and quizzes based on different factors, like doing a full 50 question test, or revising specific sections/topics only. I used the app personally for when I was learning driving theory and found it effective how you can just do questions over and over, which is arguably the best way to revise/learn something.


Theory Test Pass - UK Driving Theory Test Practice

Theory Test Pass

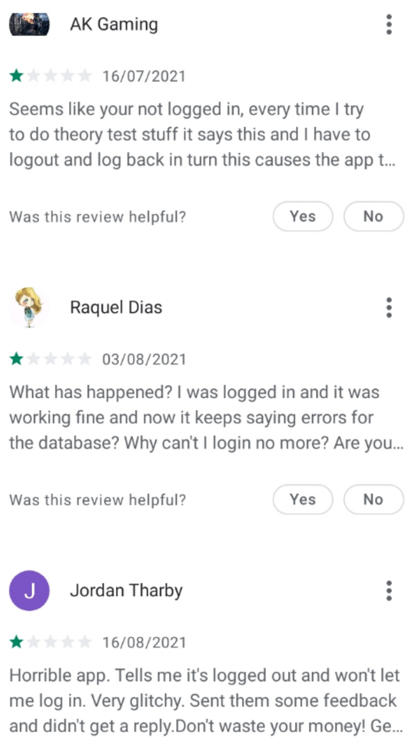
3.4★ 58 reviews | 10K+ Downloads | PEGI 3

Install

About this app

Practice for your car theory test using our mock tests, hazard perception & more

As you can see in this screenshot, the app has average reviews. Although I personally found it very straightforward and simple, other people clearly had issues with it that I was lucky to not experience or be badly affected by. I found these out when I clicked on the reviews.


AK Gaming

★ 16/07/2021

Seems like your not logged in, every time I try to do theory test stuff it says this and I have to logout and log back in turn this causes the app t...

Was this review helpful? Yes No

Raquel Dias

★ 03/08/2021

What has happened? I was logged in and it was working fine and now it keeps saying errors for the database? Why can't I login no more? Are you...

Was this review helpful? Yes No

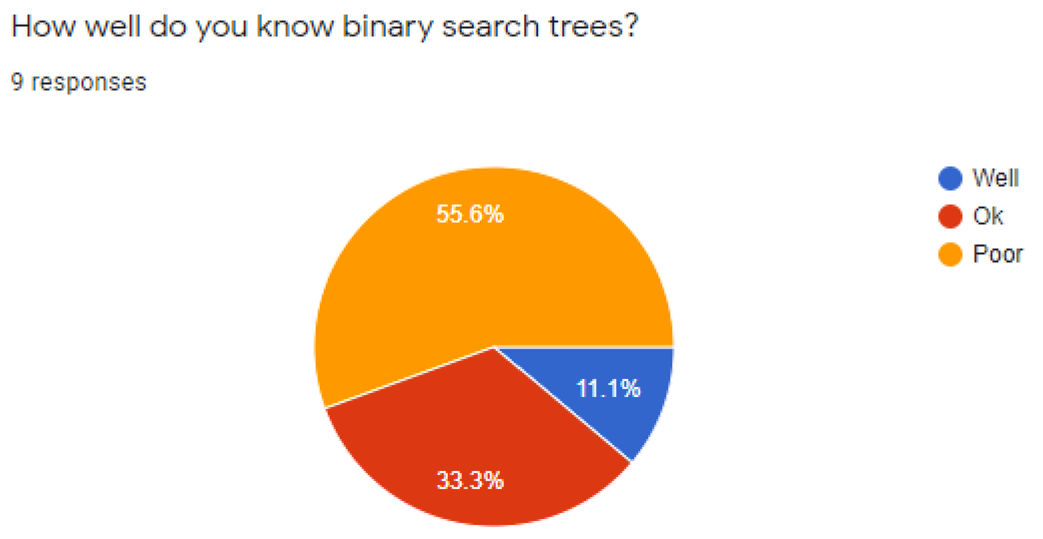
Jordan Tharby

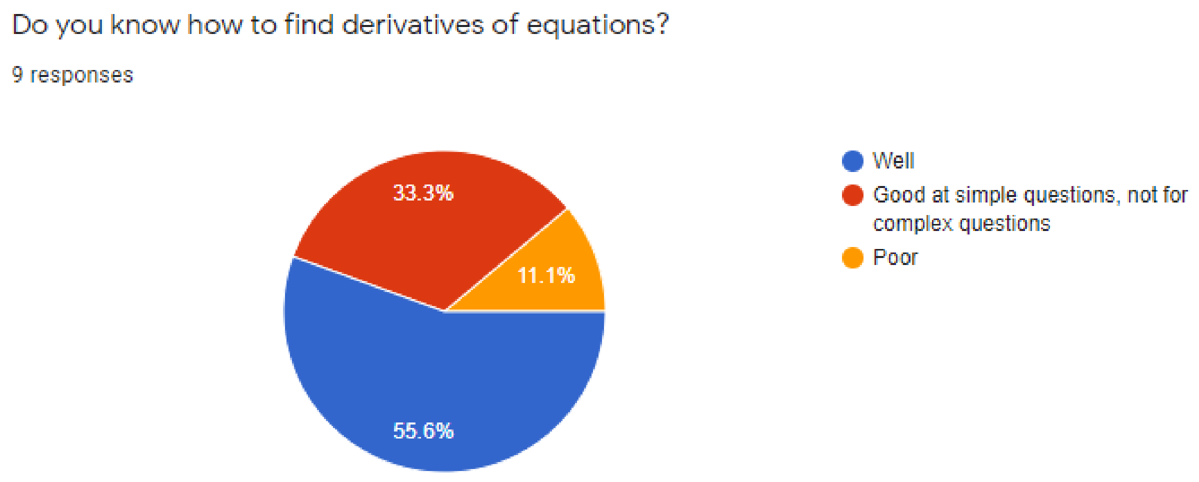
★ 16/08/2021

Horrible app. Tells me it's logged out and won't let me log in. Very glitchy. Sent them some feedback and didn't get a reply.Don't waste your money! Ge...

In summary, it seems many users (with me being an exception) suffered login issues, which probably comes from inconsistencies with database records and management of it; how the app works is you get a mobile app for phones which uses a web API to access the program and database with logins, with other devices being able to log in, such as computers, through the internet. I speculate that sessions probably weren't logged out of properly and when people accessed accounts through different devices, this caused issues with data inconsistencies and updating records with new quiz scores. Essentially, I believe poor database management and a lack of support for concurrent access rights of the database resulted in these issues, so when building my app, I need to plan out how my database system will work. Another thing to note is that the theory test app is good for revising facts and learning knowledge but in maths and computing, learning techniques and being able to problem-solve is more important, so building in a section which can show you how to perform certain skills as well as solve an equation based on inputs, like differentiating a polynomial, or putting names into a binary tree that the user inputs – like a demo/tutorial section.

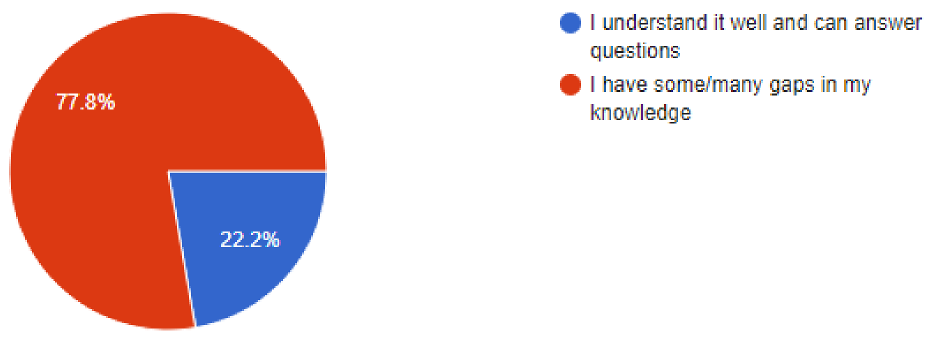
In summary, what I want to make is a desktop app that allows users to practice maths and computing with ease and on-the-go, as well as perform calculations to help them check answers with any work or questions that they might have, like a worksheet. Effectively, it is like having textbook questions (but an endless amount of them due to randomising algorithms being used to generate questions), a specialist calculator and some tutorials in one package as an app, which also monitors user progress by storing quiz scores for the user to view and compare over time. To determine whether my app would be of use, I set up a Google Forms survey for potential end-users (people who take maths, computing or do/study something similar with cross-over topics, such as engineering or IT, as well as employees of companies who use IT on a regular basis). Below are the responses I received:





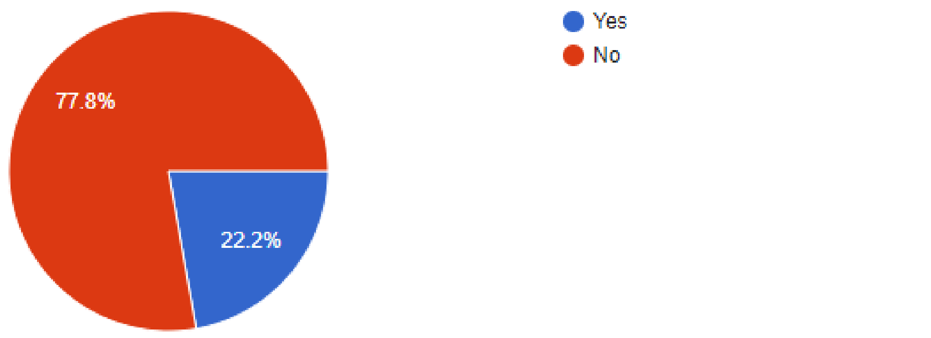
Do you understand the binomial theorem and its use of factorial? Can you answer questions on it?

9 responses



Do you have suitable resources for easy, on-the-go revision of mathematical and computing topics?

9 responses



Based on the responses, one can draw observations like how, of the people surveyed, many lack knowledge of binary search trees and likely other complex data structures like graphs, which have more real-world applications than people realise (for example, tree structures can be used to display family trees, structuring an HTML document, a book's contents, performing fast searches on numerical data and creating trees in 3D video games, while graph theory and graphs are used very extensively for developing maps and services using location, such as Google maps which uses Dijkstra's shortest path algorithm to find routes from one location to another). Another observation to be drawn is that of the people surveyed, over half have very strong knowledge of derivatives, while a smaller percentage have good knowledge of basic differentiation and a smaller percentage again cannot do it; even though the end-user feedback would imply that differentiation isn't something too hard to worry about, I still think developing an app which can test users regularly to help boost confidence, allow them to check answers with calculations they have performed with a derivative solving section and teach those who haven't grasped the concept, as well as push those with harder questions relating to differentiating logarithms for example would be a very useful tool. The next question of the survey regards the binomial theorem and Pascal's triangle, which have incredibly significant applications in the real word in statistics and making predictions on economic development; according to the survey, only a small segment of people has strong knowledge and understanding of how it works (just under ¼ of the sample). The final question of the survey that I made asked about whether people have suitable resources to learn and study from, which had an outcome where over ¾ of the sample said they lacked valuable resources, thus developing my app would provide them and likely many others with a great tool to use for assistance in maths and computing.

Objectives list:

-the app should have a log-in page for users to sign up, and for new users to create an account

-the app should store login information on the device in a SQL database for offline use

-the app should be able to solve a variety of mathematical and computational equations and problems, with a calculator for quadratics and higher order polynomials like cubics, differentiation/integration, performing expansion of bracketed polynomials by using the binomial theorem (and Pascal's triangle/factorial by extension), creating a Fibonacci sequence and finding vector dot products and angles of two vectors

-the app should be able to allow the user to input numbers into a binary search tree for demonstrative purposes

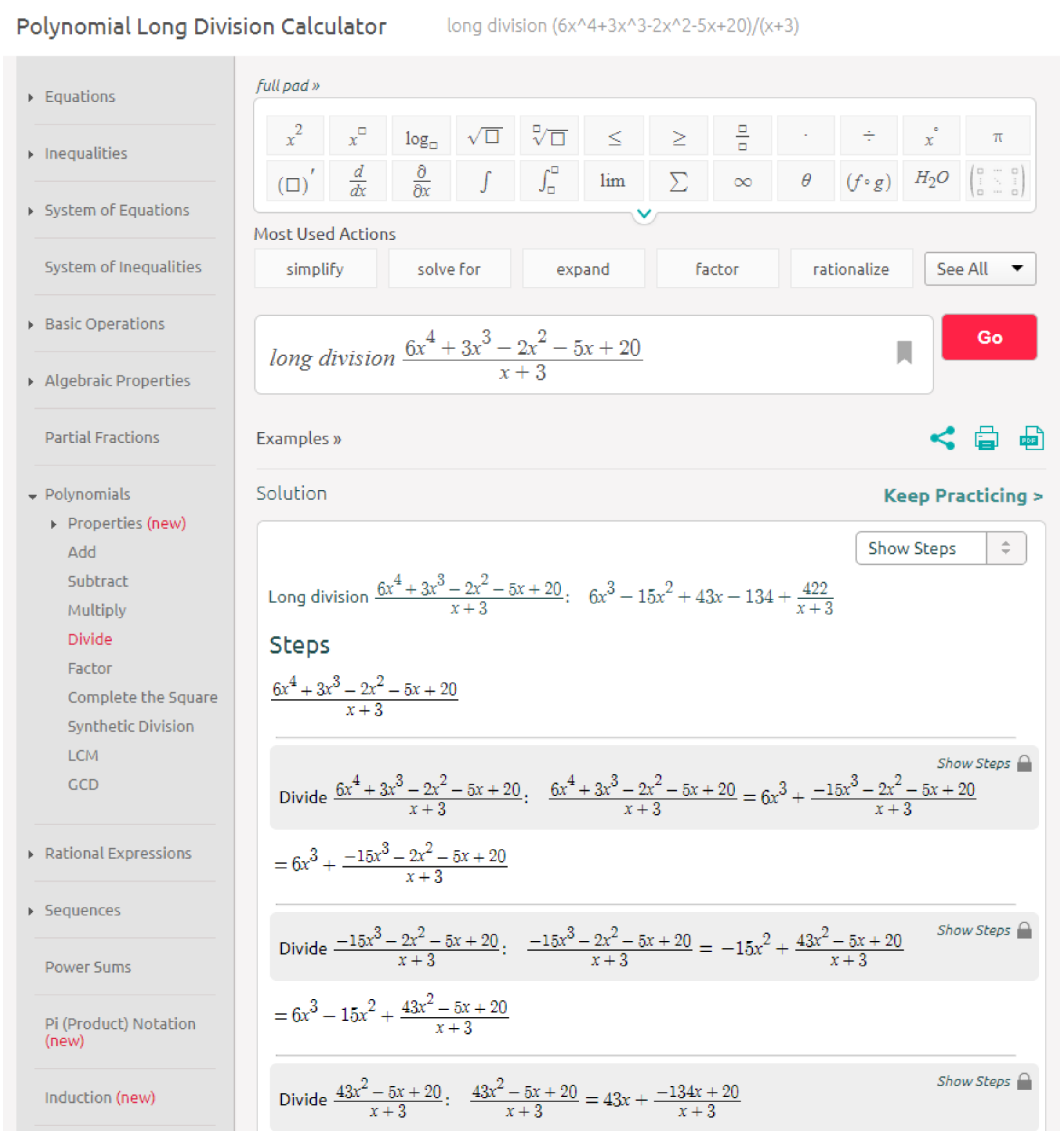
-the app should be able to generate quizzes of 5-10 largely random questions based on one of two difficulty levels (easy and hard) on the following topics: polynomials (solving quadratics, factorising cubics, and a few on the binomial theorem), differentiation/integration (involving differentiating or integrating equations involving high-power polynomials for difficult mode) and binary search trees/vectors (complex data structures)

> -the algorithm for generating the binary search tree questions will involve scraping random names from a website and generating a random birthday for each name based on a date class that I will write before inserting each person object with a name and date into the tree based on either alphabetical order or age order, with the questions being based on different traversals of the tree
>
> -the app should be able to load people's quiz scores from the database so that they can monitor their progress over time on different topics and difficulty levels in a personal leader board

Modelling, Initial Design Ideas, Prototypes and Potential Algorithms

The site 'Symbolab' has lots of elaborate calculators in it for using. One of these is for performing long division of polynomials. Below are screenshots of how the site looks after a polynomial to long divide is input.

https://www.symbolab.com/solver/polynomial-long-division-calculator/long%20division%20%5Cfrac%7B6x%5E%7B4%7D%2B3x%5E%7B3%7D-2x%5E%7B2%7D-5x%2B20%7D%7Bx%2B3%7D?or=input



$long\ division\ \frac{6x^4+3x^3-2x^2-5x+20}{x+3}$

Long division $\frac{6x^4+3x^3-2x^2-5x+20}{x+3}$: $6x^3-15x^2+43x-134+\frac{422}{x+3}$

Steps

$\frac{6x^4+3x^3-2x^2-5x+20}{x+3}$

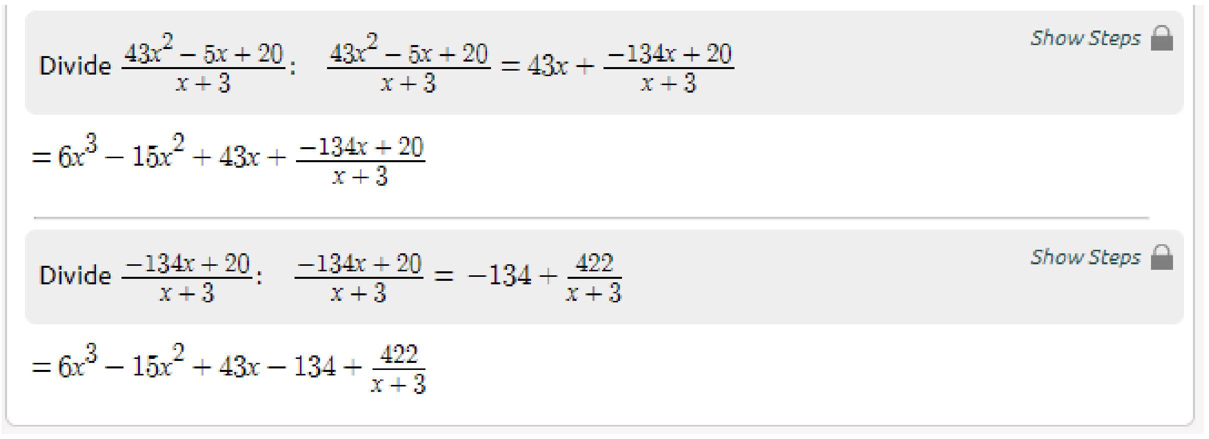
Divide $\frac{6x^4+3x^3-2x^2-5x+20}{x+3}$: $\frac{6x^4+3x^3-2x^2-5x+20}{x+3}=6x^3+\frac{-15x^3-2x^2-5x+20}{x+3}$ *Show Steps*

$=6x^3+\frac{-15x^3-2x^2-5x+20}{x+3}$

Divide $\frac{-15x^3-2x^2-5x+20}{x+3}$: $\frac{-15x^3-2x^2-5x+20}{x+3}=-15x^2+\frac{43x^2-5x+20}{x+3}$ *Show Steps*

$=6x^3-15x^2+\frac{43x^2-5x+20}{x+3}$

Divide $\frac{43x^2-5x+20}{x+3}$: $\frac{43x^2-5x+20}{x+3}=43x+\frac{-134x+20}{x+3}$ *Show Steps*



Divide $\frac{43x^2-5x+20}{x+3}$: $\frac{43x^2-5x+20}{x+3}=43x+\frac{-134x+20}{x+3}$ *Show Steps*

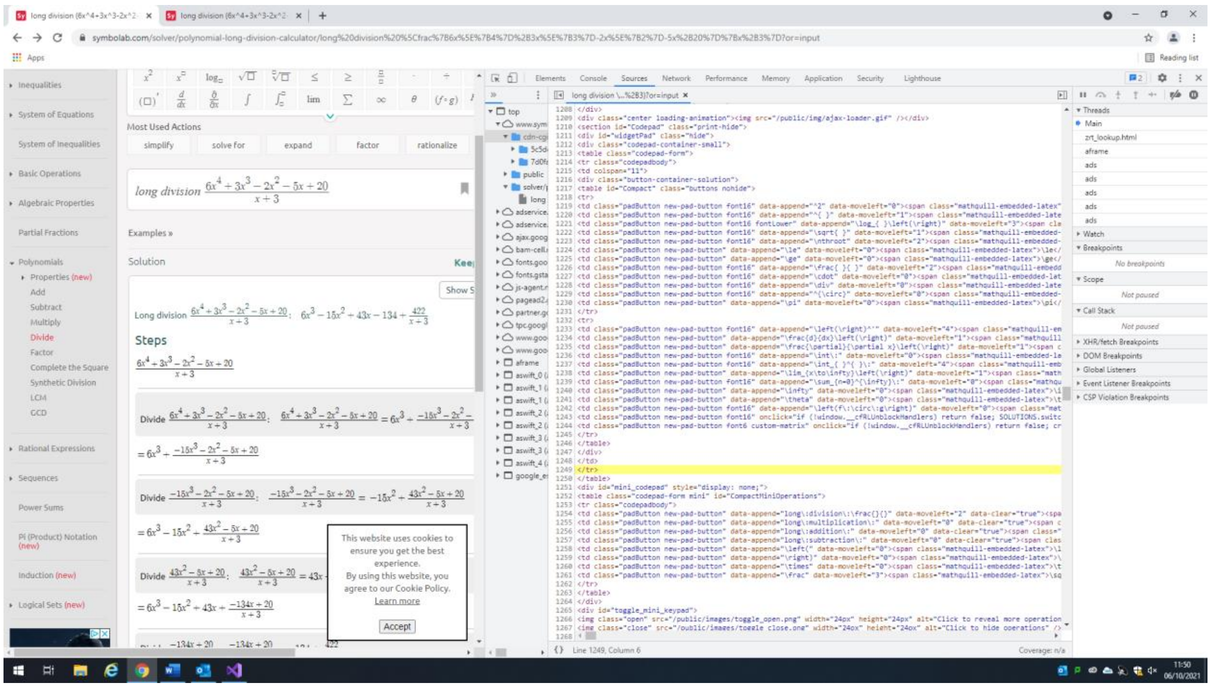
$=6x^3-15x^2+43x+\frac{-134x+20}{x+3}$

Divide $\frac{-134x+20}{x+3}$: $\frac{-134x+20}{x+3}=-134+\frac{422}{x+3}$ *Show Steps*

$=6x^3-15x^2+43x-134+\frac{422}{x+3}$

When looking into the html, the way it shows each step is by creating a table with each row representing each step as far as I can tell, and then data items within each row appear to contain numbers involved in the process of dividing the polynomial. I struggled to make much of the html however I can deduce that the best way to approach this problem is to create some form of recursive algorithm which keeps dividing the polynomial while the parts of the polynomial are added to an object that belongs to the polynomial class.

```html
<div class="button-container-solution">
<table id="Compact" class="buttons nohide">
<tr>
<td class="padButton new-pad-button font16" data-append="^2" data-moveleft="0"><span class="mathquill-embedded-late
<td class="padButton new-pad-button font16" data-append="^{ }" data-moveleft="1"><span class="mathquill-embedded-la
<td class="padButton new-pad-button font16 fontLower" data-append="\log_{ }\left(\right)" data-moveleft="3"><span c
<td class="padButton new-pad-button font16" data-append="\sqrt{ }" data-moveleft="1"><span class="mathquill-embedde
<td class="padButton new-pad-button font16" data-append="\nthroot" data-moveleft="2"><span class="mathquill-embedde
<td class="padButton new-pad-button" data-append="\le" data-moveleft="0"><span class="mathquill-embedded-latex">\le
<td class="padButton new-pad-button" data-append="\ge" data-moveleft="0"><span class="mathquill-embedded-latex">\ge
<td class="padButton new-pad-button font16" data-append="\frac{ }{ }" data-moveleft="2"><span class="mathquill-embe
<td class="padButton new-pad-button font16" data-append="\cdot" data-moveleft="0"><span class="mathquill-embedded-l
<td class="padButton new-pad-button font16" data-append="\div" data-moveleft="0"><span class="mathquill-embedded-la
<td class="padButton new-pad-button font16" data-append="^{\circ}" data-moveleft="0"><span class="mathquill-embedde
<td class="padButton new-pad-button" data-append="\pi" data-moveleft="0"><span class="mathquill-embedded-latex">\pi
</tr>
<tr>
<td class="padButton new-pad-button font16" data-append="\left(\right)^'" data-moveleft="4"><span class="mathquill-
<td class="padButton new-pad-button" data-append="\frac{d}{dx}\left(\right)" data-moveleft="1"><span class="mathqui
<td class="padButton new-pad-button" data-append="\frac{\partial}{\partial x}\left(\right)" data-moveleft="1"><span
<td class="padButton new-pad-button font16" data-append="\int\:" data-moveleft="0"><span class="mathquill-embedded-
<td class="padButton new-pad-button font16" data-append="\int_{ }^{ }\:" data-moveleft="4"><span class="mathquill-e
<td class="padButton new-pad-button" data-append="\lim_{x\to\infty}\left(\right)" data-moveleft="1"><span class="ma
<td class="padButton new-pad-button font16" data-append="\sum_{n=0}^{\infty}\:" data-moveleft="0"><span class="math
<td class="padButton new-pad-button" data-append="\infty" data-moveleft="0"><span class="mathquill-embedded-latex">
<td class="padButton new-pad-button" data-append="\theta" data-moveleft="0"><span class="mathquill-embedded-latex">
<td class="padButton new-pad-button font16" data-append="\left(f\:\circ\:g\right)" data-moveleft="0"><span class="m
<td class="padButton new-pad-button font16" onclick="if (!window.__cfRLUnblockHandlers) return false; SOLUTIONS.swi
<td class="padButton new-pad-button font6 custom-matrix" onclick="if (!window.__cfRLUnblockHandlers) return false;
</tr>
</table>
</div>
```



symbolab.com/solver/polynomial-long-division-calculator/long%20division%20%5Cfrac%7B6x%5E%7B4%7D%2B3x%5E%7B3%7D-2x%5E%7B2%7D-5x%2B20%7D%7Bx%2B3%7D?or=input

long division $\frac{6x^4+3x^3-2x^2-5x+20}{x+3}$

Solution

Long division $\frac{6x^4+3x^3-2x^2-5x+20}{x+3}$: $6x^3-15x^2+43x-134+\frac{422}{x+3}$

Steps

$\frac{6x^4+3x^3-2x^2-5x+20}{x+3}$

Divide $\frac{6x^4+3x^3-2x^2-5x+20}{x+3}$: $\frac{6x^4+3x^3-2x^2-5x+20}{x+3}=6x^3+\frac{-15x^3-2x^2-}{x+3}$

$=6x^3+\frac{-15x^3-2x^2-5x+20}{x+3}$

Divide $\frac{-15x^3-2x^2-5x+20}{x+3}$: $\frac{-15x^3-2x^2-5x+20}{x+3}=-15x^2+\frac{43x^2-5x+20}{x+3}$

$=6x^3-15x^2+\frac{43x^2-5x+20}{x+3}$

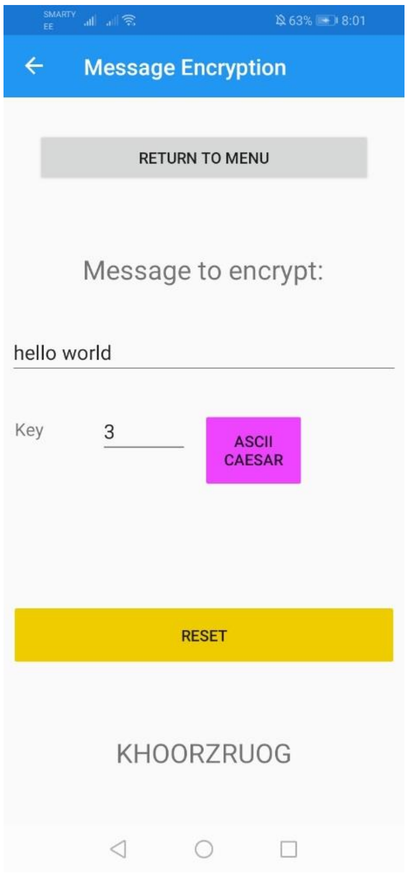
Divide $\frac{43x^2-5x+20}{x+3}$: $\frac{43x^2-5x+20}{x+3}=43x$

$=6x^3-15x^2+43x+\frac{-134x+20}{x+3}$

This website uses cookies to ensure you get the best experience. By using this website, you agree to our Cookie Policy. Learn more

Accept

I have developed a multitude of prototype pages for the app, including a Caesar cipher, a differentiation and integration calculator (shown below is differentiation), a quadratic equation solver which can also solve quadratics with imaginary roots and will serve as a basis for developing a polynomial solver, e.g. cubics and quartics, a memory section which loads previous calculations performed on the device (which in the final version will be linked to the user who is logged in at the time) into a list view using a SQLite database, and a binomial theorem program whereby you enter values for a, b and n in the form (ax +b) ^ n and the polynomial is displayed in a list view with each power of x and its coefficient taking up one cell each in the listview (this uses factorial a lot and thus uses a recursive function) - see screenshots below.

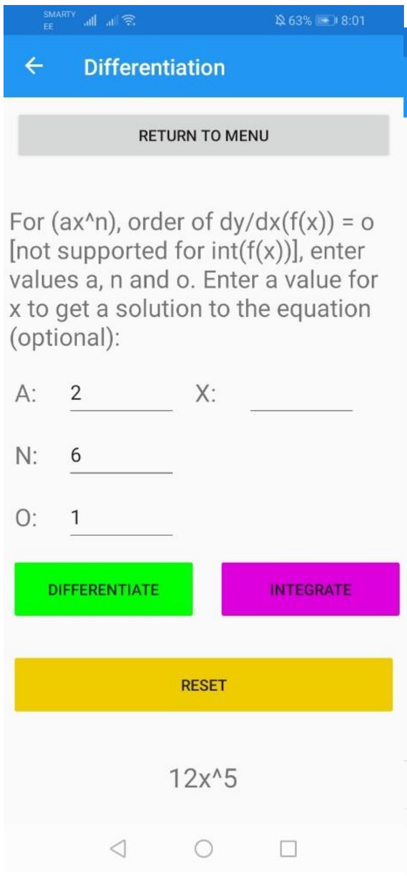


Message to encrypt:

hello world

Key 3

KHOORZRUOG



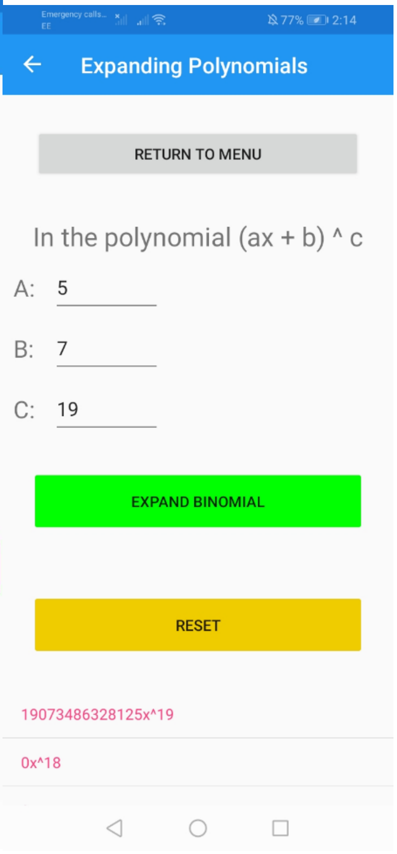
For (ax^n), order of dy/dx(f(x)) = o [not supported for int(f(x))], enter values a, n and o. Enter a value for x to get a solution to the equation (optional):

A: 2 X:

N: 6

O: 1

12x^5



In the polynomial (ax + b) ^ c

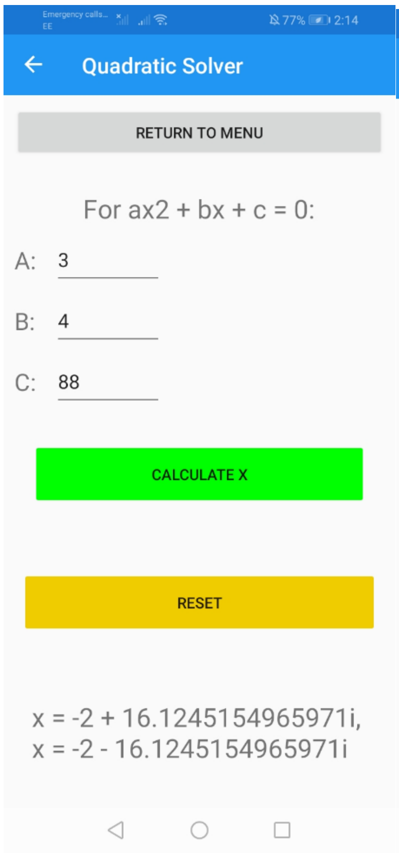
A: 5

B: 7

C: 19

19073486328125x^19

0x^18



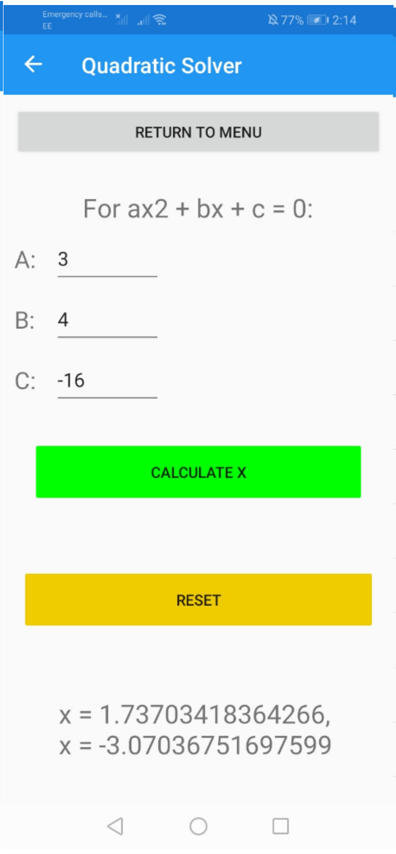
For ax2 + bx + c = 0:

A: 3

B: 4

C: 88

x = -2 + 16.1245154965971i,
x = -2 - 16.1245154965971i



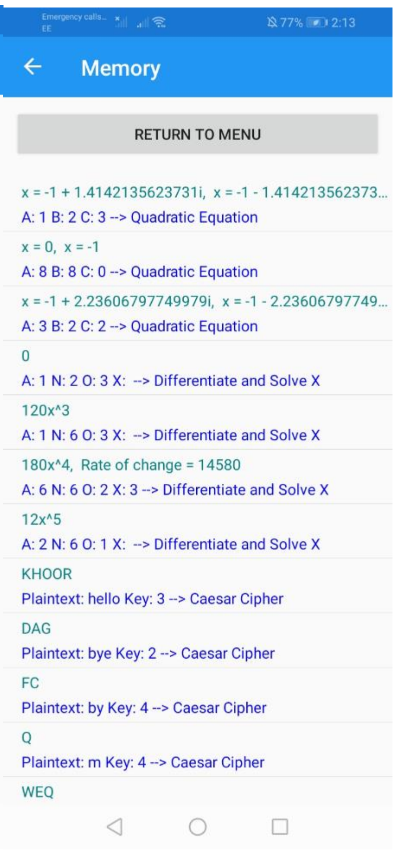
For ax2 + bx + c = 0:

A: 3

B: 4

C: -16

x = 1.73703418364266,
x = -3.07036751697599



x = -1 + 1.4142135623731i, x = -1 - 1.414213562373...
A: 1 B: 2 C: 3 --> Quadratic Equation

x = 0, x = -1
A: 8 B: 8 C: 0 --> Quadratic Equation

x = -1 + 2.23606797749979i, x = -1 - 2.23606797749...
A: 3 B: 2 C: 2 --> Quadratic Equation

0
A: 1 N: 2 O: 3 X: --> Differentiate and Solve X

120x^3
A: 1 N: 6 O: 3 X: --> Differentiate and Solve X

180x^4, Rate of change = 14580
A: 6 N: 6 O: 2 X: 3 --> Differentiate and Solve X

12x^5
A: 2 N: 6 O: 1 X: --> Differentiate and Solve X

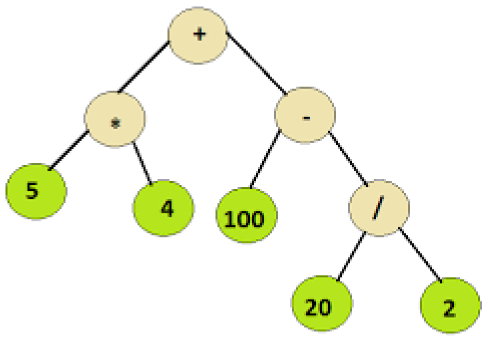
KHOOR
Plaintext: hello Key: 3 --> Caesar Cipher

DAG
Plaintext: bye Key: 2 --> Caesar Cipher

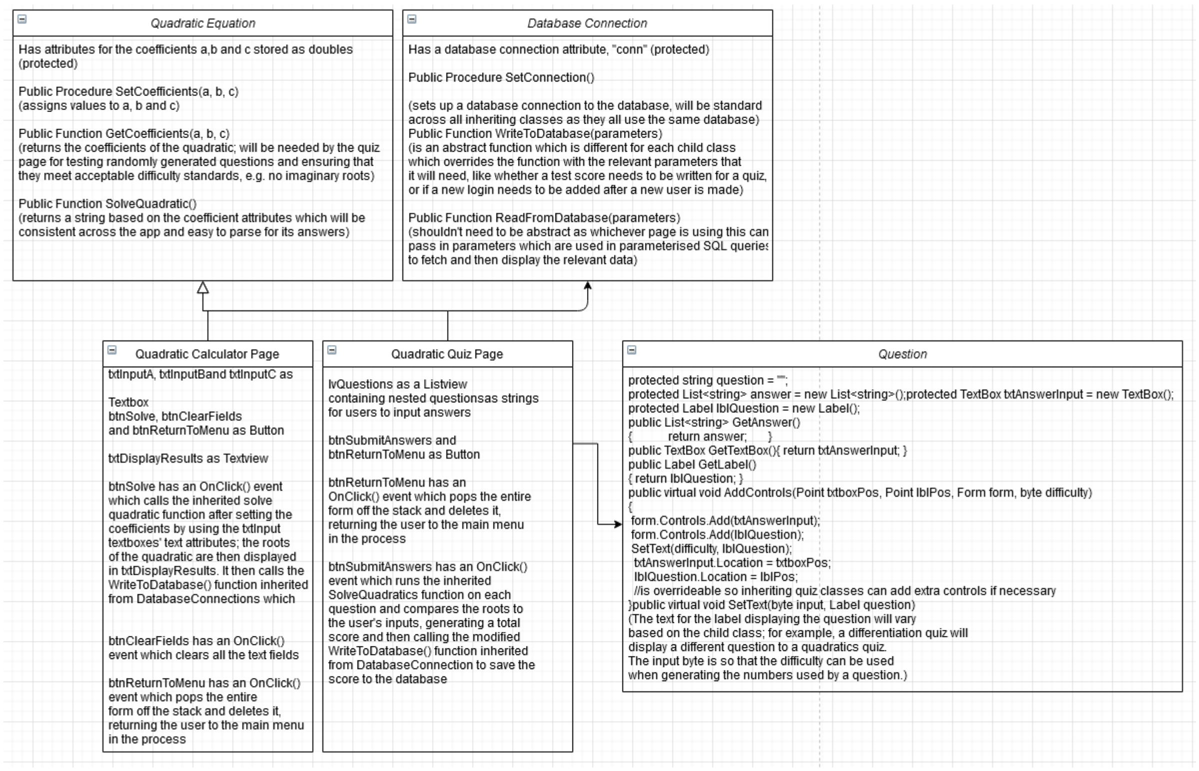
FC
Plaintext: by Key: 4 --> Caesar Cipher

Q
Plaintext: m Key: 4 --> Caesar Cipher

WEQ



I have also developed a prototype for the random quiz page generations using Winforms which is mainly for practising how my algorithms will run and how the classes will be constructed - 'fleshing out' ideas. Once I have the algorithms in place and class structures sorted, it isn't hard to simply copy the code into the main project and alter the UI elements that it interacts with so that it is compatible for a Xamarin Forms project. Below is a screenshot of a basic class diagram I came up with for the quadratics quiz page which shows which pages inherit from what and some of their attributes. Using Winforms, I fleshed out some of these as more ideas of what should be included came to mind.



Quadratic Equation

Has attributes for the coefficients a,b and c stored as doubles (protected)

Public Procedure SetCoefficients(a, b, c)
(assigns values to a, b and c)

Public Function GetCoefficients(a, b, c)
(returns the coefficients of the quadratic; will be needed by the quiz page for testing randomly generated questions and ensuring that they meet acceptable difficulty standards, e.g. no imaginary roots)

Public Function SolveQuadratic()
(returns a string based on the coefficient attributes which will be consistent across the app and easy to parse for its answers)

Database Connection

Has a database connection attribute, "conn" (protected)

Public Procedure SetConnection()

(sets up a database connection to the database, will be standard across all inheriting classes as they all use the same database)
Public Function WriteToDatabase(parameters)
(is an abstract function which is different for each child class which overrides the function with the relevant parameters that it will need, like whether a test score needs to be written for a quiz, or if a new login needs to be added after a new user is made)

Public Function ReadFromDatabase(parameters)
(shouldn't need to be abstract as whichever page is using this can pass in parameters which are used in parameterised SQL queries to fetch and then display the relevant data)

Quadratic Calculator Page

txtInputA, txtInputBand txtInputC as

Textbox
btnSolve, btnClearFields
and btnReturnToMenu as Button

txtDisplayResults as Textview

btnSolve has an OnClick() event which calls the inherited solve quadratic function after setting the coefficients by using the txtInput textboxes' text attributes; the roots of the quadratic are then displayed in txtDisplayResults. It then calls the WriteToDatabase() function inherited from DatabaseConnections which

btnClearFields has an OnClick() event which clears all the text fields

btnReturnToMenu has an OnClick() event which pops the entire form off the stack and deletes it, returning the user to the main menu in the process

Quadratic Quiz Page

lvQuestions as a Listview
containing nested questionsas strings
for users to input answers

btnSubmitAnswers and
btnReturnToMenu as Button

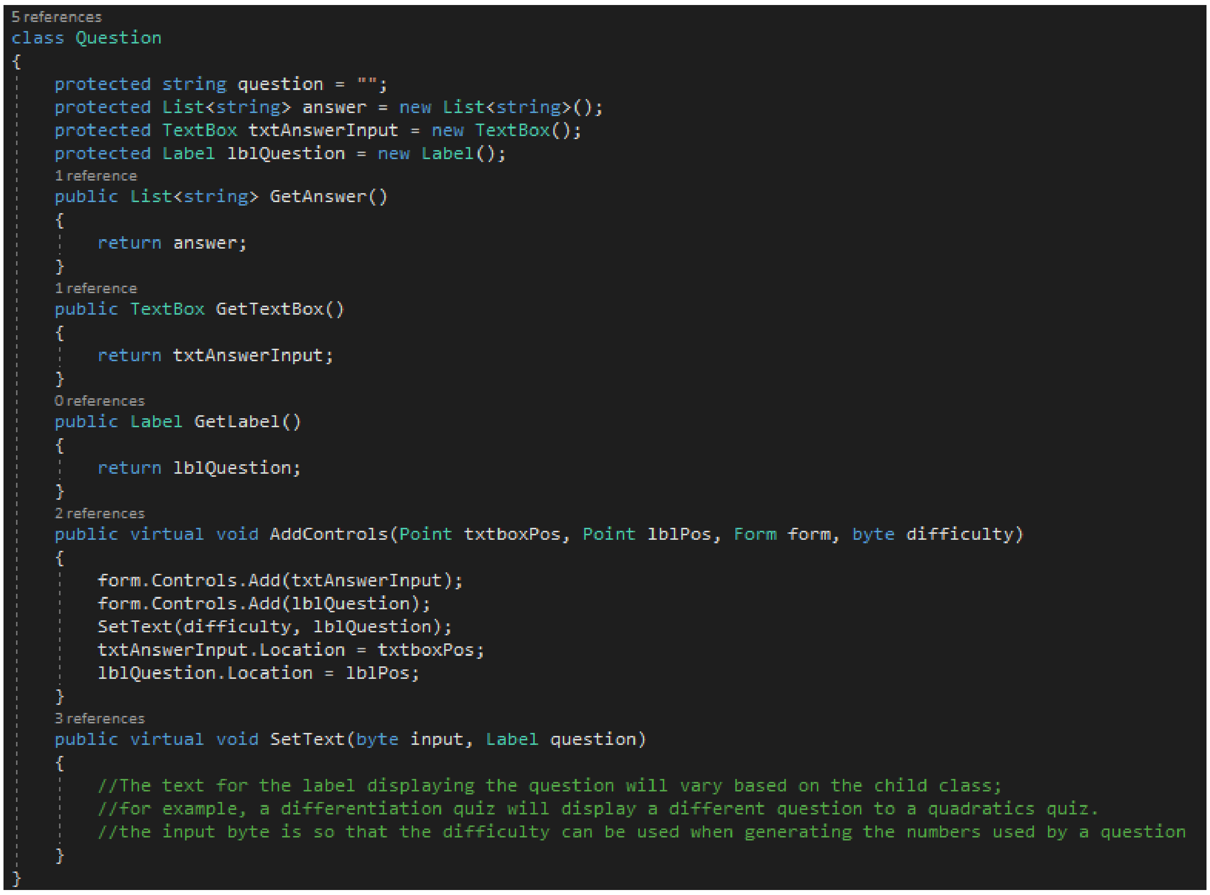
btnReturnToMenu has an OnClick() event which pops the entire form off the stack and deletes it, returning the user to the main menu in the process

btnSubmitAnswers has an OnClick() event which runs the inherited SolveQuadratics function on each question and compares the roots to the user's inputs, generating a total score and then calling the modified WriteToDatabase() function inherited from DatabaseConnection to save the score to the database

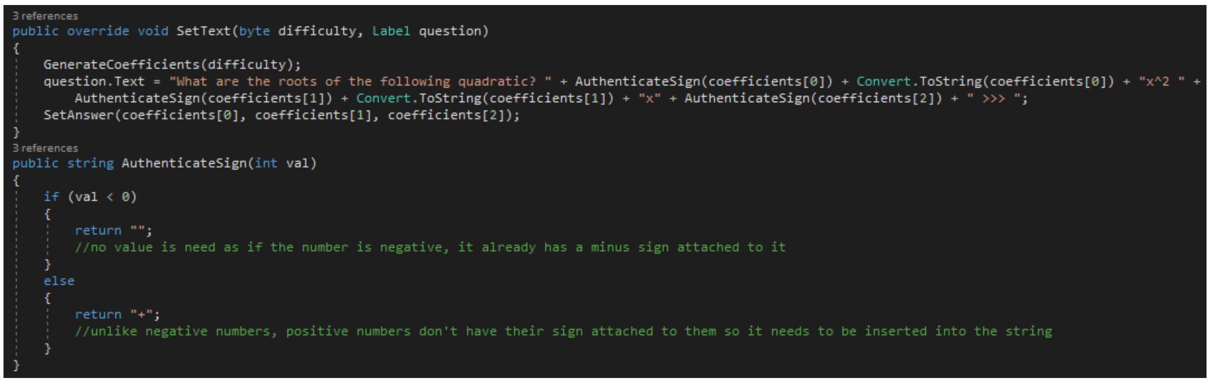
Question

```
protected string question = "";
protected List<string> answer = new List<string>();protected TextBox txtAnswerInput = new TextBox();
protected Label lblQuestion = new Label();
public List<string> GetAnswer()
{           return answer;       }
public TextBox GetTextBox(){ return txtAnswerInput; }
public Label GetLabel()
{ return lblQuestion; }
public virtual void AddControls(Point txtboxPos, Point lblPos, Form form, byte difficulty)
{
 form.Controls.Add(txtAnswerInput);
 form.Controls.Add(lblQuestion);
 SetText(difficulty, lblQuestion);
  txtAnswerInput.Location = txtboxPos;
  lblQuestion.Location = lblPos;
  //is overrideable so inheriting quiz classes can add extra controls if necessary
}public virtual void SetText(byte input, Label question)
(The text for the label displaying the question will vary
based on the child class; for example, a differentiation quiz will
display a different question to a quadratics quiz.
The input byte is so that the difficulty can be used
when generating the numbers used by a question.)
```

Note: the above screenshot talks about classes in reference to a Xamarin Forms unlike the screenshot below this note which is of code in a Winforms project and thus may contain Winforms-specific UI interactions which will need to be taken into account for my final build.



```
5 references
class Question
{
    protected string question = "";
    protected List<string> answer = new List<string>();
    protected TextBox txtAnswerInput = new TextBox();
    protected Label lblQuestion = new Label();
    1 reference
    public List<string> GetAnswer()
    {
        return answer;
    }
    1 reference
    public TextBox GetTextBox()
    {
        return txtAnswerInput;
    }
    0 references
    public Label GetLabel()
    {
        return lblQuestion;
    }
    2 references
    public virtual void AddControls(Point txtboxPos, Point lblPos, Form form, byte difficulty)
    {
        form.Controls.Add(txtAnswerInput);
        form.Controls.Add(lblQuestion);
        SetText(difficulty, lblQuestion);
        txtAnswerInput.Location = txtboxPos;
        lblQuestion.Location = lblPos;
    }
    3 references
    public virtual void SetText(byte input, Label question)
    {
        //The text for the label displaying the question will vary based on the child class;
        //for example, a differentiation quiz will display a different question to a quadratics quiz.
        //the input byte is so that the difficulty can be used when generating the numbers used by a question
    }
}
```

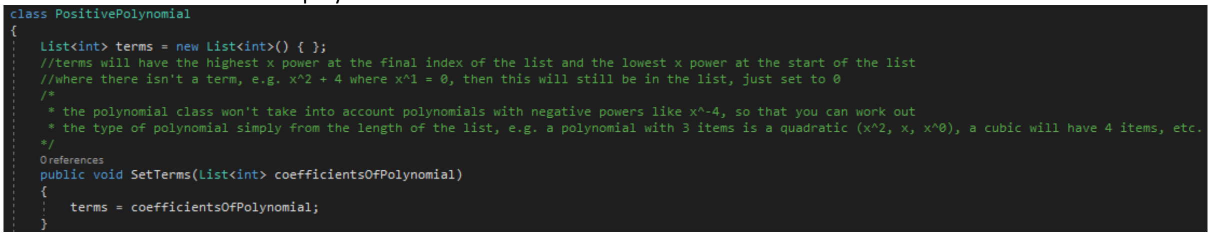


```
3 references
public override void SetText(byte difficulty, Label question)
{
    GenerateCoefficients(difficulty);
    question.Text = "What are the roots of the following quadratic? " + AuthenticateSign(coefficients[0]) + Convert.ToString(coefficients[0]) + "x^2 " +
        AuthenticateSign(coefficients[1]) + Convert.ToString(coefficients[1]) + "x" + AuthenticateSign(coefficients[2]) + " >>> ";
    SetAnswer(coefficients[0], coefficients[1], coefficients[2]);
}
3 references
public string AuthenticateSign(int val)
{
    if (val < 0)
    {
        return "";
        //no value is need as if the number is negative, it already has a minus sign attached to it
    }
    else
    {
        return "+";
        //unlike negative numbers, positive numbers don't have their sign attached to them so it needs to be inserted into the string
    }
}
```

The above screenshot is code from the quadratic question class which shows an example of how quiz pages which inherit from a basic question class will be able to override the SetText() method for setting the question's label based on the question type and what numbers will be shown.

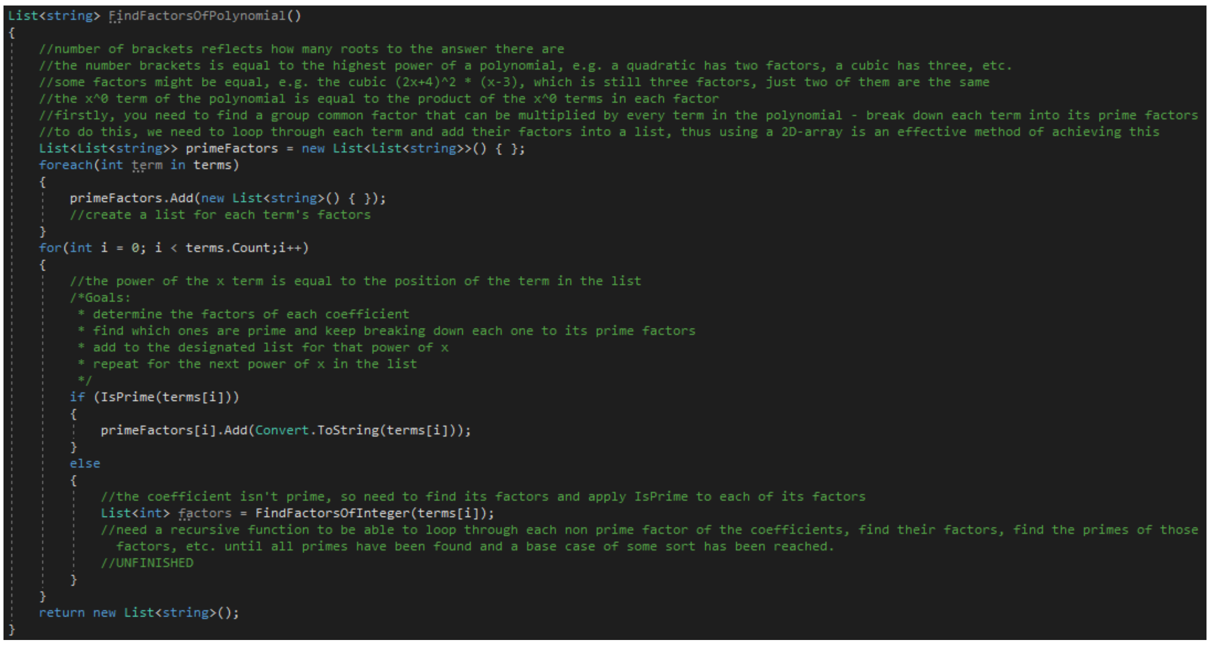
Whilst developing my prototype of quiz generation in Winforms, I decided that it would be useful to create a class which I can use to standardise how my app will handle polynomials like quadratics, cubics, quartics, etc. One way I thought of doing this was to create a list attribute for the polynomial class whereby each item in the list is a coefficient of a term in the polynomial and the index of the item in the list is the power of x that the coefficient is for, e.g the cubic equation 5x^3 - 6x^2 + 4x + 7 would be written as { 7, 4, -6, 5 }. The last of index of this list is 3 which corresponds to the power of x that the number at this index (5) is the coefficient of. This system makes it easy to recreate a polynomial equation as a string to insert into a question as well as deal with specific parts of it and factorise it. Below is the start of a prototype for a polynomial class called PositivePolynomial, which means the powers of it are positive, e.g. x^-2 + 3x + 5 wouldn't be a possible polynomial to make - I may add functionality for polynomials of this type however most questions at A-Level for maths typically deal with positive polynomials as such so it wouldn't be the most useful thing to add.

Below: code to set terms for a polynomial

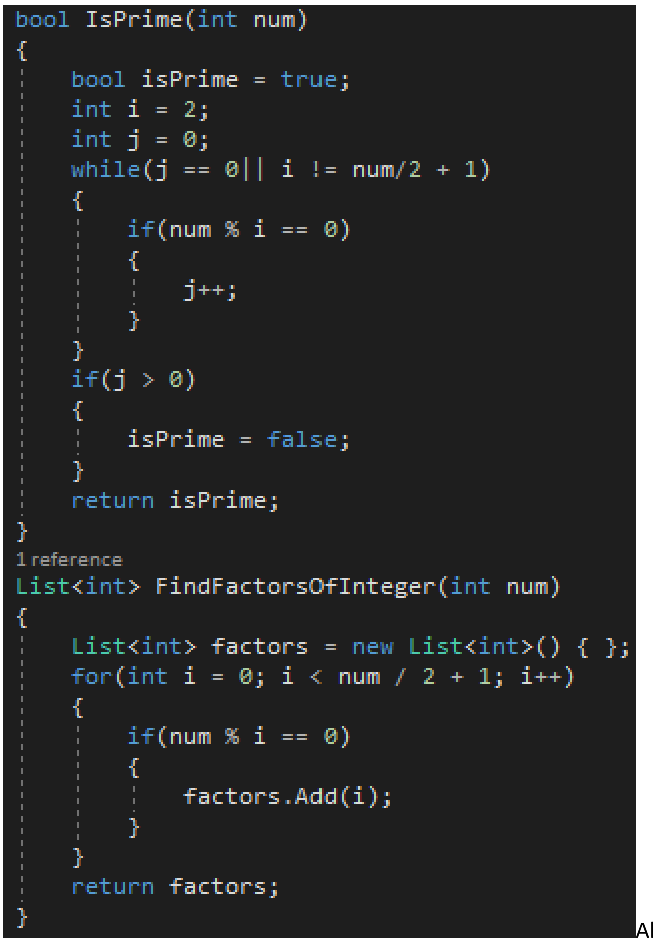


```
class PositivePolynomial
{
    List<int> terms = new List<int>() { };
    //terms will have the highest x power at the final index of the list and the lowest x power at the start of the list
    //where there isn't a term, e.g. x^2 + 4 where x^1 = 0, then this will still be in the list, just set to 0
    /*
     * the polynomial class won't take into account polynomials with negative powers like x^-4, so that you can work out
     * the type of polynomial simply from the length of the list, e.g. a polynomial with 3 items is a quadratic (x^2, x, x^0), a cubic will have 4 items, etc.
    */
    0 references
    public void SetTerms(List<int> coefficientsOfPolynomial)
    {
        terms = coefficientsOfPolynomial;
    }
```

Below: A start to an algorithm which will find the factors of a polynomial - it is unfinished for now, however its intended functionality is explained in the code's comments whereby it should use a recursive function to iterate through its coefficients to ensure they are broken down into prime factors.



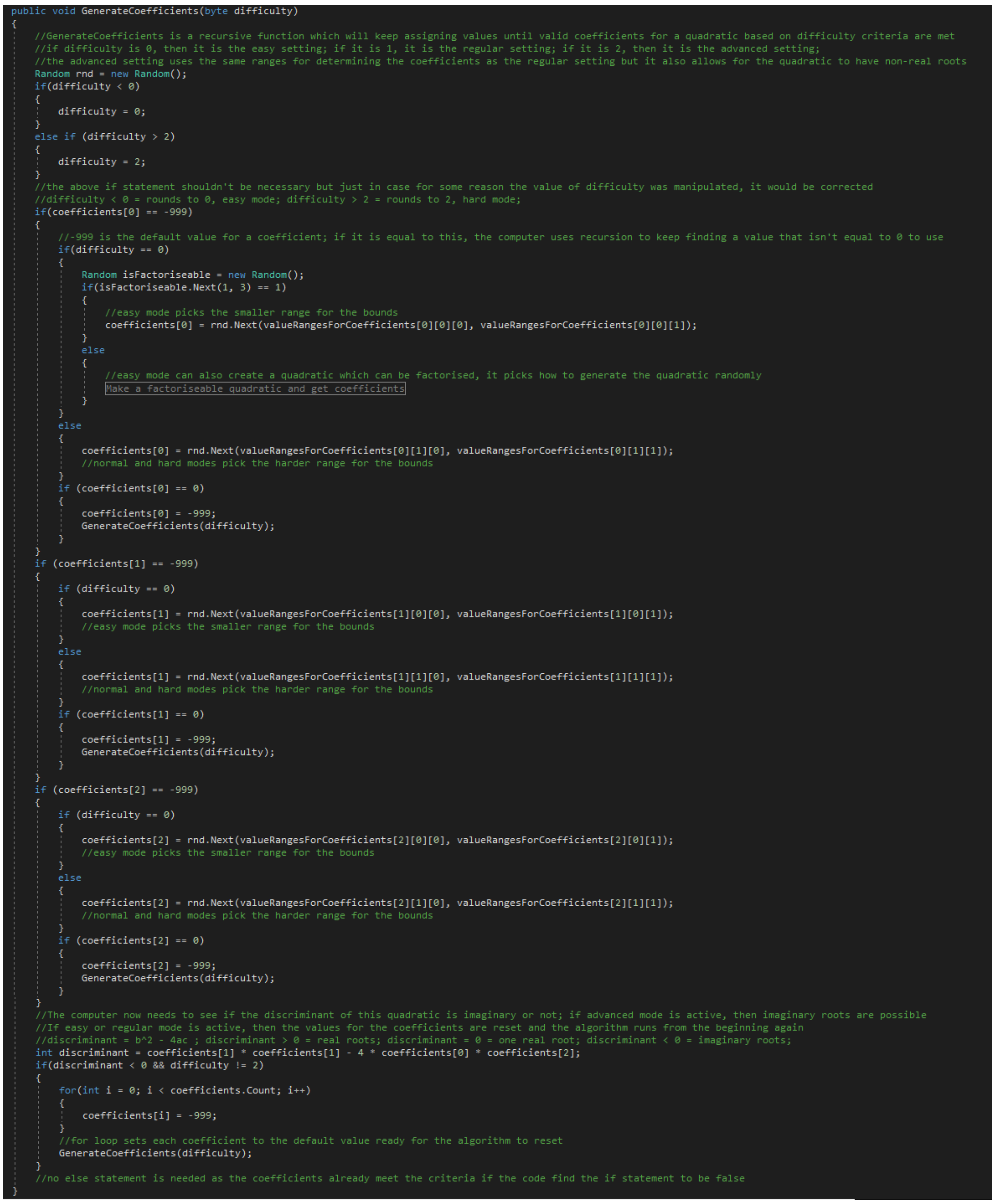
```
List<string> FindFactorsOfPolynomial()
{
    //number of brackets reflects how many roots to the answer there are
    //the number brackets is equal to the highest power of a polynomial, e.g. a quadratic has two factors, a cubic has three, etc.
    //some factors might be equal, e.g. the cubic (2x+4)^2 * (x-3), which is still three factors, just two of them are the same
    //the x^0 term of the polynomial is equal to the product of the x^0 terms in each factor
    //firstly, you need to find a group common factor that can be multiplied by every term in the polynomial - break down each term into its prime factors
    //to do this, we need to loop through each term and add their factors into a list, thus using a 2D-array is an effective method of achieving this
    List<List<string>> primeFactors = new List<List<string>>() { };
    foreach(int term in terms)
    {
        primeFactors.Add(new List<string>() { });
        //create a list for each term's factors
    }
    for(int i = 0; i < terms.Count;i++)
    {
        //the power of the x term is equal to the position of the term in the list
        /*Goals:
         * determine the factors of each coefficient
         * find which ones are prime and keep breaking down each one to its prime factors
         * add to the designated list for that power of x
         * repeat for the next power of x in the list
         */
        if (IsPrime(terms[i]))
        {
            primeFactors[i].Add(Convert.ToString(terms[i]));
        }
        else
        {
            //the coefficient isn't prime, so need to find its factors and apply IsPrime to each of its factors
            List<int> factors = FindFactorsOfInteger(terms[i]);
            //need a recursive function to be able to loop through each non prime factor of the coefficients, find their factors, find the primes of those
              factors, etc. until all primes have been found and a base case of some sort has been reached.
            //UNFINISHED
        }
    }
    return new List<string>();
}
```



```
bool IsPrime(int num)
{
    bool isPrime = true;
    int i = 2;
    int j = 0;
    while(j == 0|| i != num/2 + 1)
    {
        if(num % i == 0)
        {
            j++;
        }
    }
    if(j > 0)
    {
        isPrime = false;
    }
    return isPrime;
}
1 reference
List<int> FindFactorsOfInteger(int num)
{
    List<int> factors = new List<int>() { };
    for(int i = 0; i < num / 2 + 1; i++)
    {
        if(num % i == 0)
        {
            factors.Add(i);
        }
    }
    return factors;
}
```

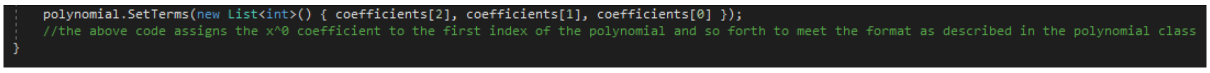
Algorithm for finding out if a number is prime and its factors, intended to be used for finishing the algorithm that breaks down a polynomial into its factors by breaking down its coefficients into prime factors.

Below is a screenshot of an algorithm which generates coefficients for a quadratic based on an input byte which determines the difficulty of the quiz; this model could be incorporated into my project or used alongside another algorithm or two for creating quadratics which are suitably challenging.

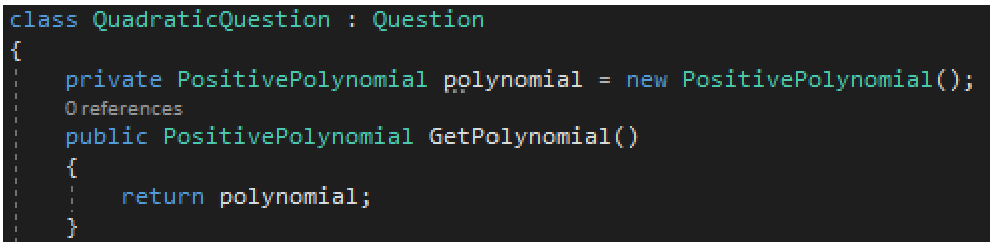


```
public void GenerateCoefficients(byte difficulty)
{
    //GenerateCoefficients is a recursive function which will keep assigning values until valid coefficients for a quadratic based on difficulty criteria are met
    //if difficulty is 0, then it is the easy setting; if it is 1, it is the regular setting; if it is 2, then it is the advanced setting;
    //the advanced setting uses the same ranges for determining the coefficients as the regular setting but it also allows for the quadratic to have non-real roots
    Random rnd = new Random();
    if(difficulty < 0)
    {
        difficulty = 0;
    }
    else if (difficulty > 2)
    {
        difficulty = 2;
    }
    //the above if statement shouldn't be necessary but just in case for some reason the value of difficulty was manipulated, it would be corrected
    //difficulty < 0 = rounds to 0, easy mode; difficulty > 2 = rounds to 2, hard mode;
    if(coefficients[0] == -999)
    {
        //-999 is the default value for a coefficient; if it is equal to this, the computer uses recursion to keep finding a value that isn't equal to 0 to use
        if(difficulty == 0)
        {
            Random isFactoriseable = new Random();
            if(isFactoriseable.Next(1, 3) == 1)
            {
                //easy mode picks the smaller range for the bounds
                coefficients[0] = rnd.Next(valueRangesForCoefficients[0][0][0], valueRangesForCoefficients[0][0][1]);
            }
            else
            {
                //easy mode can also create a quadratic which can be factorised, it picks how to generate the quadratic randomly
                Make a factoriseable quadratic and get coefficients
            }
        }
        else
        {
            coefficients[0] = rnd.Next(valueRangesForCoefficients[0][1][0], valueRangesForCoefficients[0][1][1]);
            //normal and hard modes pick the harder range for the bounds
        }
        if (coefficients[0] == 0)
        {
            coefficients[0] = -999;
            GenerateCoefficients(difficulty);
        }
    }
    if (coefficients[1] == -999)
    {
        if (difficulty == 0)
        {
            coefficients[1] = rnd.Next(valueRangesForCoefficients[1][0][0], valueRangesForCoefficients[1][0][1]);
            //easy mode picks the smaller range for the bounds
        }
        else
        {
            coefficients[1] = rnd.Next(valueRangesForCoefficients[1][1][0], valueRangesForCoefficients[1][1][1]);
            //normal and hard modes pick the harder range for the bounds
        }
        if (coefficients[1] == 0)
        {
            coefficients[1] = -999;
            GenerateCoefficients(difficulty);
        }
    }
    if (coefficients[2] == -999)
    {
        if (difficulty == 0)
        {
            coefficients[2] = rnd.Next(valueRangesForCoefficients[2][0][0], valueRangesForCoefficients[2][0][1]);
            //easy mode picks the smaller range for the bounds
        }
        else
        {
            coefficients[2] = rnd.Next(valueRangesForCoefficients[2][1][0], valueRangesForCoefficients[2][1][1]);
            //normal and hard modes pick the harder range for the bounds
        }
        if (coefficients[2] == 0)
        {
            coefficients[2] = -999;
            GenerateCoefficients(difficulty);
        }
    }
    //The computer now needs to see if the discriminant of this quadratic is imaginary or not; if advanced mode is active, then imaginary roots are possible
    //If easy or regular mode is active, then the values for the coefficients are reset and the algorithm runs from the beginning again
    //discriminant = b^2 - 4ac ; discriminant > 0 = real roots; discriminant = 0 = one real root; discriminant < 0 = imaginary roots;
    int discriminant = coefficients[1] * coefficients[1] - 4 * coefficients[0] * coefficients[2];
    if(discriminant < 0 && difficulty != 2)
    {
        for(int i = 0; i < coefficients.Count; i++)
        {
            coefficients[i] = -999;
        }
        //for loop sets each coefficient to the default value ready for the algorithm to reset
        GenerateCoefficients(difficulty);
    }
    //no else statement is needed as the coefficients already meet the criteria if the code find the if statement to be false
}
```

After inserting the above screenshot, I updated the QuadraticQuestion class so that each object of this class in this Winforms prototype project has a PositivePolynomial object to use (class described further up in document). Thus here are some screenshots of the code in the QuadraticQuestion making use of the added object (first screenshot is an extension to the method, "GenerateCoefficients(byte difficulty)":

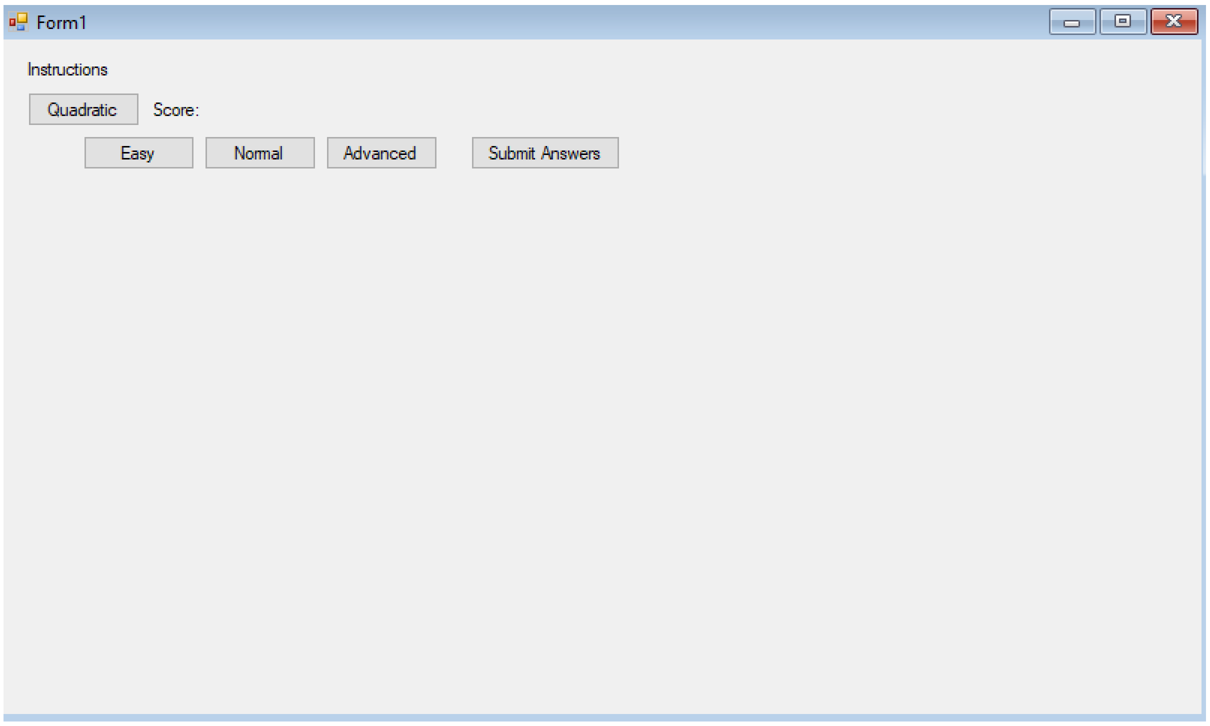


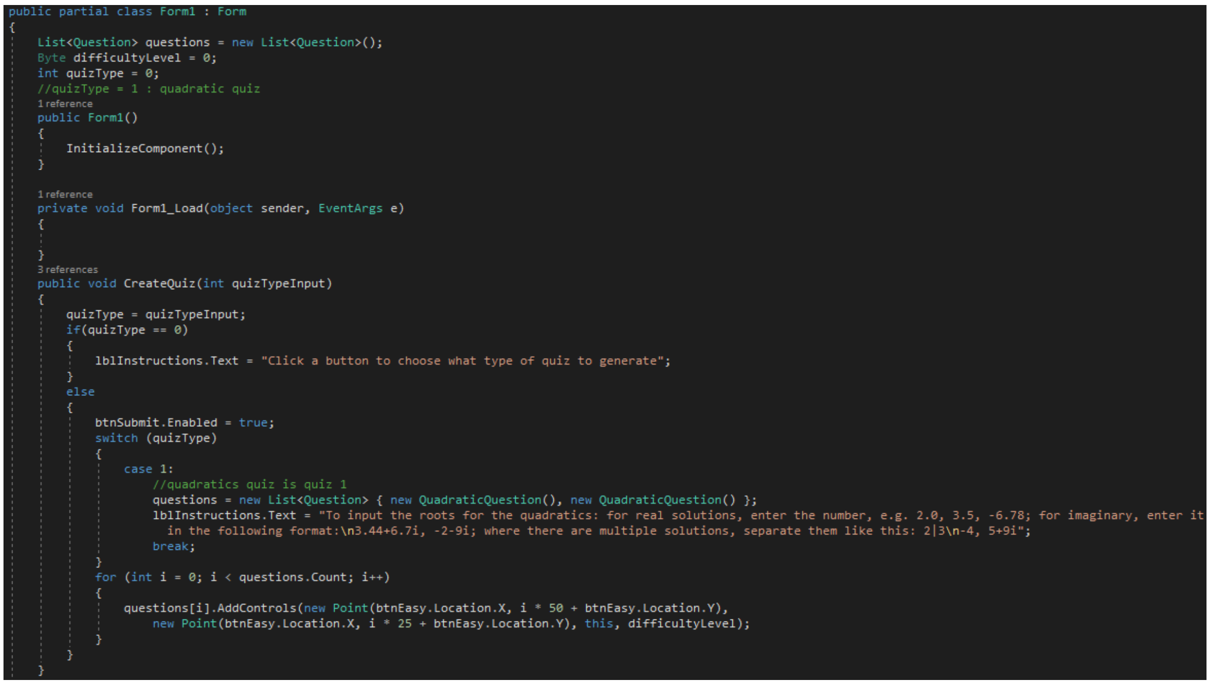
```
    polynomial.SetTerms(new List<int>() { coefficients[2], coefficients[1], coefficients[0] });
    //the above code assigns the x^0 coefficient to the first index of the polynomial and so forth to meet the format as described in the polynomial class
}
```



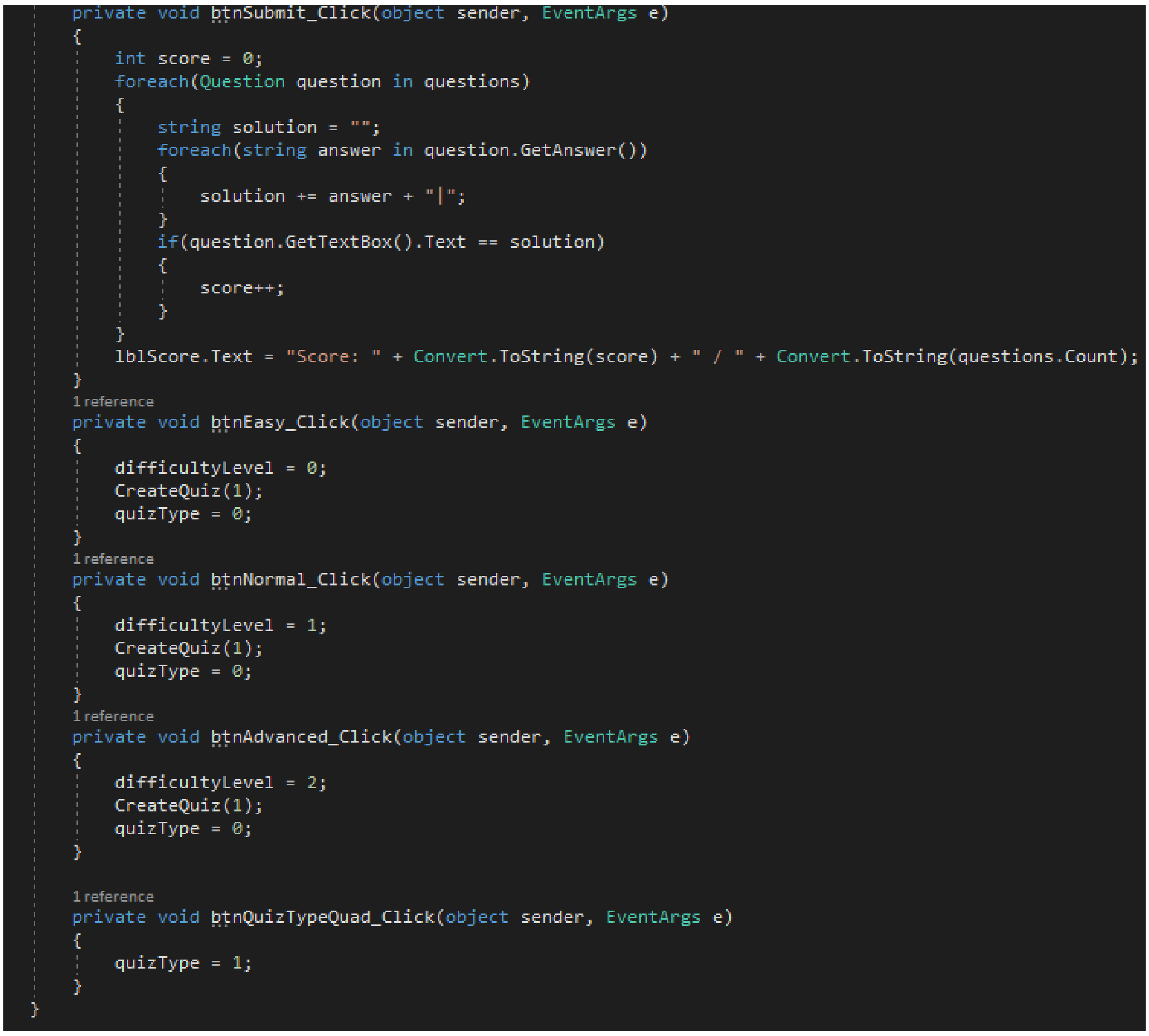
```
class QuadraticQuestion : Question
{
    private PositivePolynomial polynomial = new PositivePolynomial();
    0 references
    public PositivePolynomial GetPolynomial()
    {
        return polynomial;
    }
```

The following screenshot is a basic UI design model for the Winforms prototype: you click on the Quadratic button to set the type of quiz to be generated to be a quadratic quiz (in this screenshot that is the only mode currently implemented) and then click on easy, normal or advanced to generate a quiz of the relevant difficulty. Using these parameters, the code then adds controls to the current form to be used to inputting answers and reading what the questions are; once all answers are in the input text boxes, you then click 'Submit Answers' to get the quiz graded.



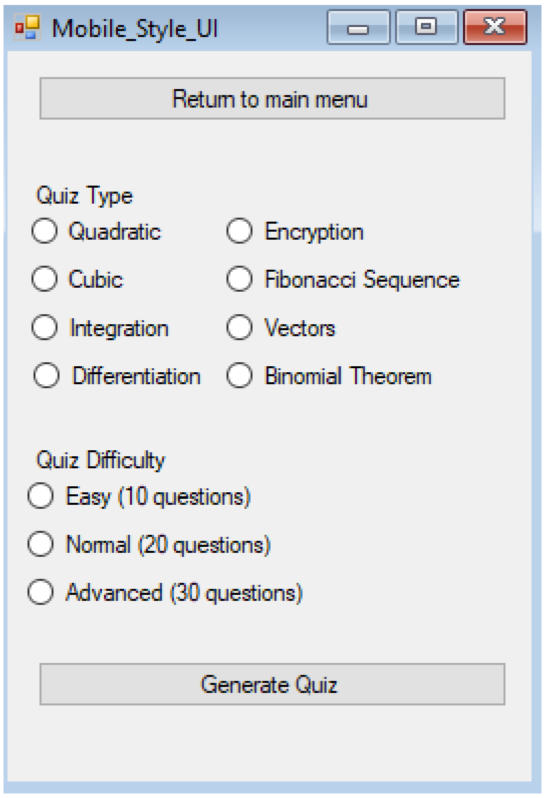


```
public partial class Form1 : Form
{
    List<Question> questions = new List<Question>();
    Byte difficultyLevel = 0;
    int quizType = 0;
    //quizType = 1 : quadratic quiz
    1 reference
    public Form1()
    {
        InitializeComponent();
    }

    1 reference
    private void Form1_Load(object sender, EventArgs e)
    {

    }
    3 references
    public void CreateQuiz(int quizTypeInput)
    {
        quizType = quizTypeInput;
        if(quizType == 0)
        {
            lblInstructions.Text = "Click a button to choose what type of quiz to generate";
        }
        else
        {
            btnSubmit.Enabled = true;
            switch (quizType)
            {
                case 1:
                    //quadratics quiz is quiz 1
                    questions = new List<Question> { new QuadraticQuestion(), new QuadraticQuestion() };
                    lblInstructions.Text = "To input the roots for the quadratics: for real solutions, enter the number, e.g. 2.0, 3.5, -6.78; for imaginary, enter it
                      in the following format:\n3.44+6.7i, -2-9i; where there are multiple solutions, separate them like this: 2|3\n-4, 5+9i";
                    break;
            }
            for (int i = 0; i < questions.Count; i++)
            {
                questions[i].AddControls(new Point(btnEasy.Location.X, i * 50 + btnEasy.Location.Y),
                    new Point(btnEasy.Location.X, i * 25 + btnEasy.Location.Y), this, difficultyLevel);
            }
        }
    }
```

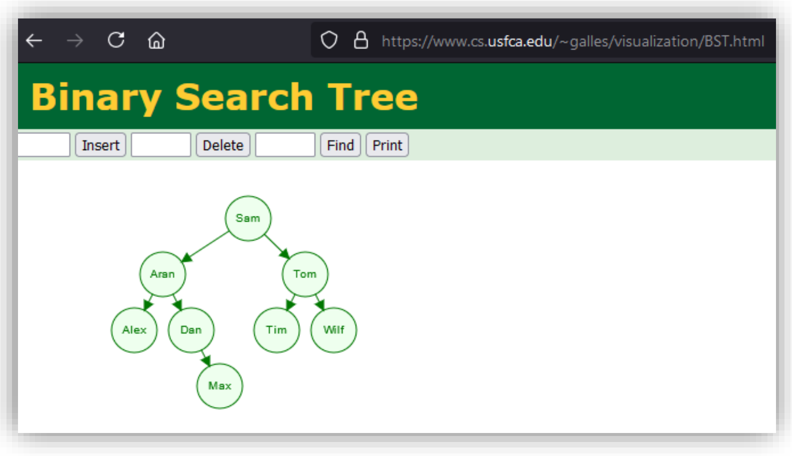


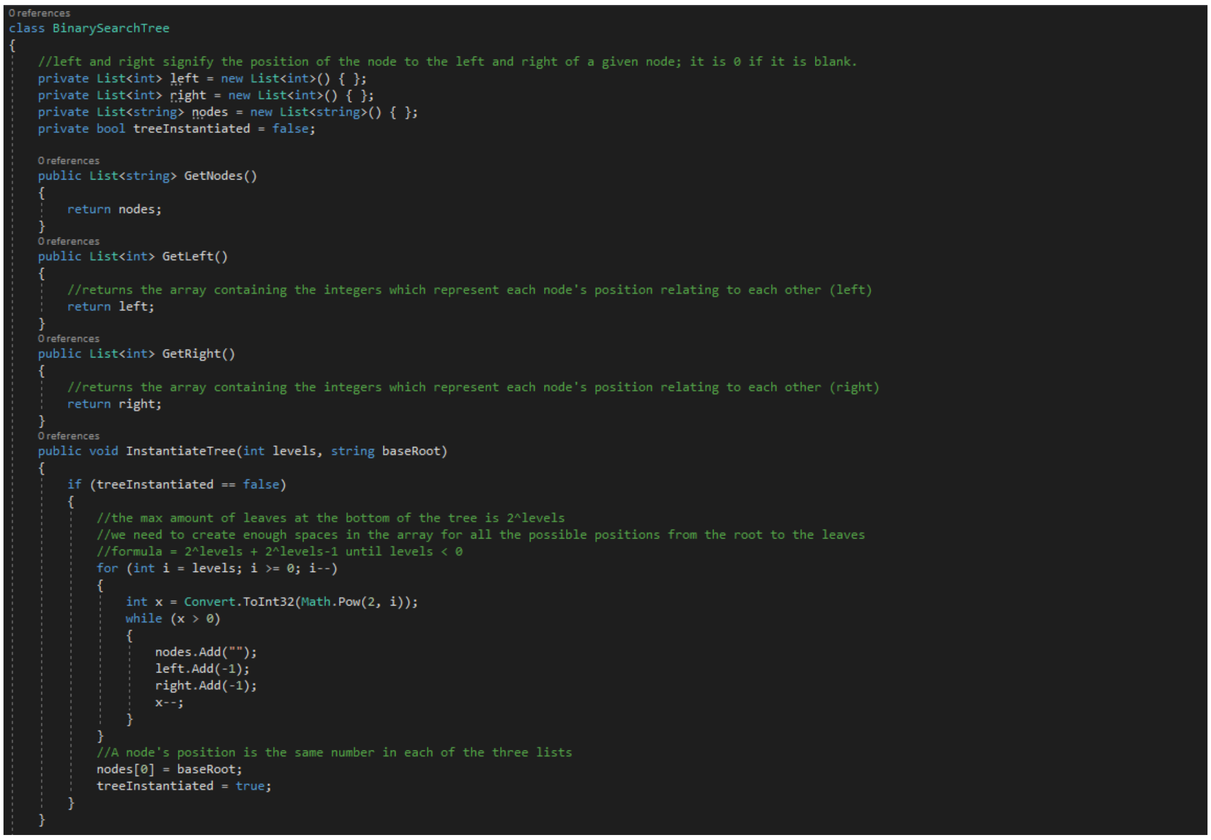
```
    private void btnSubmit_Click(object sender, EventArgs e)
    {
        int score = 0;
        foreach(Question question in questions)
        {
            string solution = "";
            foreach(string answer in question.GetAnswer())
            {
                solution += answer + "|";
            }
            if(question.GetTextBox().Text == solution)
            {
                score++;
            }
        }
        lblScore.Text = "Score: " + Convert.ToString(score) + " / " + Convert.ToString(questions.Count);
    }
    1 reference
    private void btnEasy_Click(object sender, EventArgs e)
    {
        difficultyLevel = 0;
        CreateQuiz(1);
        quizType = 0;
    }
    1 reference
    private void btnNormal_Click(object sender, EventArgs e)
    {
        difficultyLevel = 1;
        CreateQuiz(1);
        quizType = 0;
    }
    1 reference
    private void btnAdvanced_Click(object sender, EventArgs e)
    {
        difficultyLevel = 2;
        CreateQuiz(1);
        quizType = 0;
    }

    1 reference
    private void btnQuizTypeQuad_Click(object sender, EventArgs e)
    {
        quizType = 1;
    }
}
```

Below is a possible template for what the quiz menu may look like:

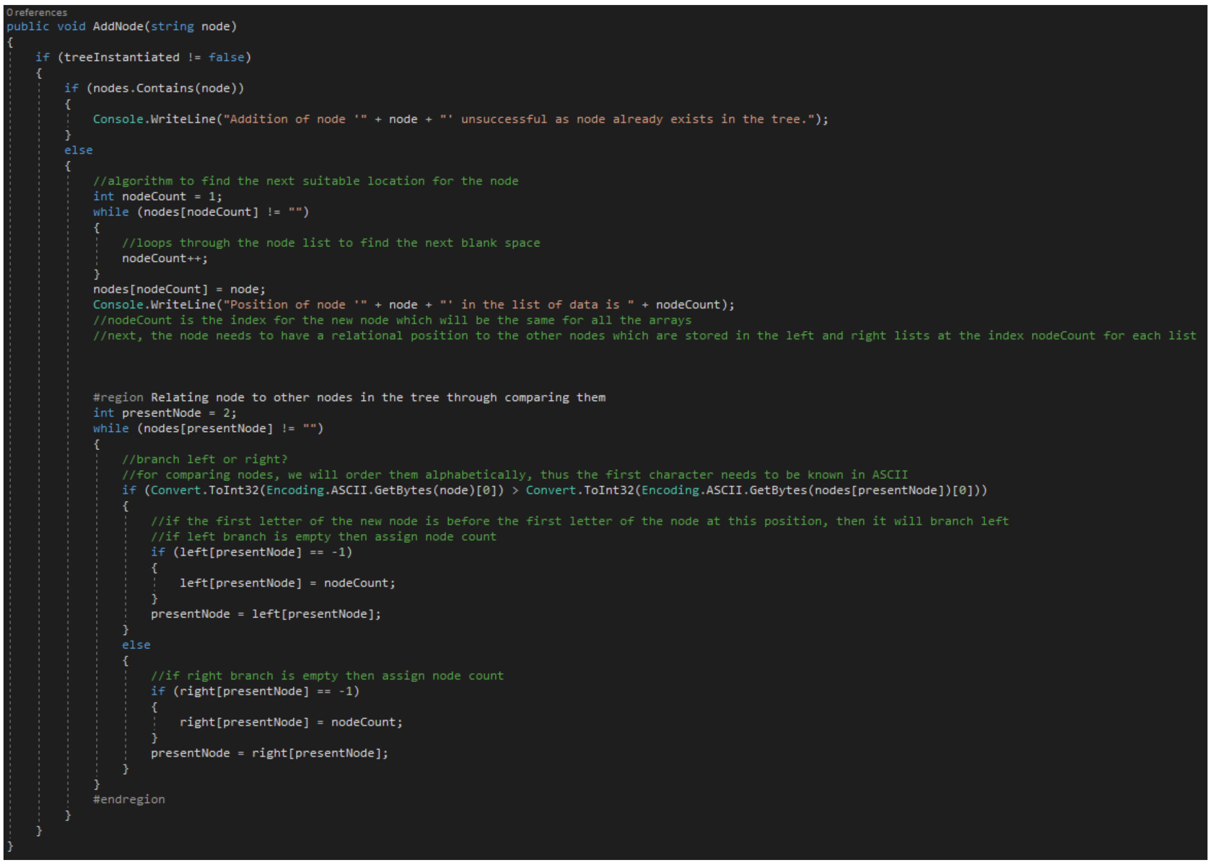


Below is a screenshot of a site that has a binary search tree simulator which allows the user to input names which get assigned to branches/nodes on the tree based on their alphabetic relation to pre-existing nodes - site = https://www.cs.usfca.edu/~galles/visualization/BST.html. I intend to basically develop a copy of this but for a mobile app that runs locally on the phone and not in a browser. I have drafted a prototype for a binary search tree class which isn't fully functional yet (screenshots are below the website screenshot).

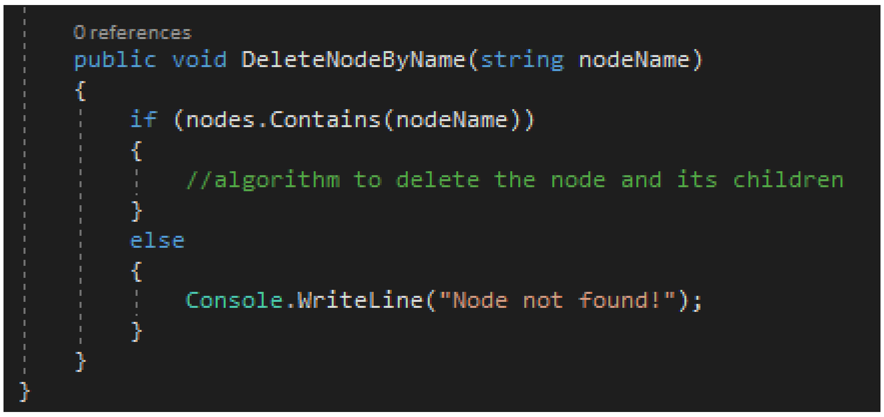




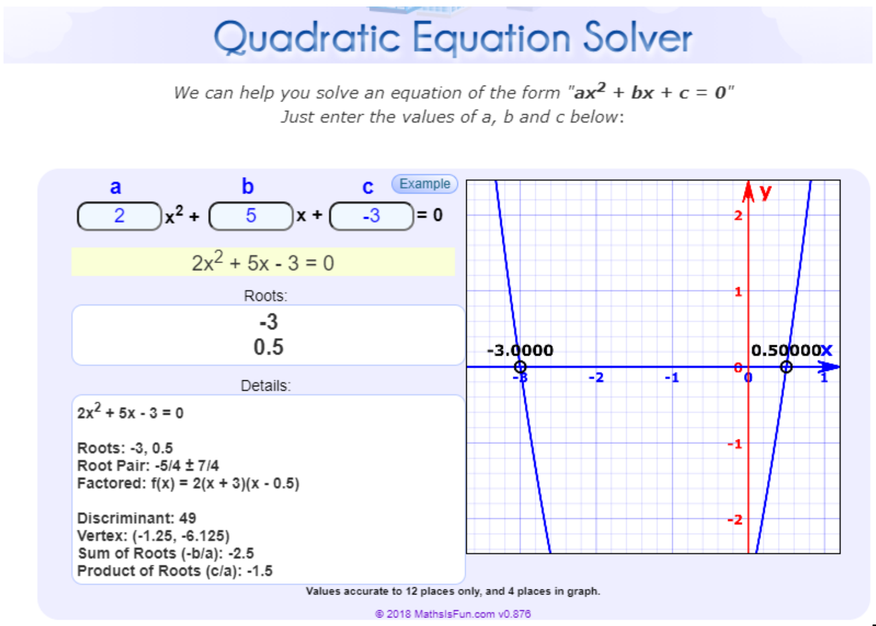
```csharp
0 references
class BinarySearchTree
{
    //left and right signify the position of the node to the left and right of a given node; it is 0 if it is blank.
    private List<int> left = new List<int>() { };
    private List<int> right = new List<int>() { };
    private List<string> nodes = new List<string>() { };
    private bool treeInstantiated = false;

    0 references
    public List<string> GetNodes()
    {
        return nodes;
    }
    0 references
    public List<int> GetLeft()
    {
        //returns the array containing the integers which represent each node's position relating to each other (left)
        return left;
    }
    0 references
    public List<int> GetRight()
    {
        //returns the array containing the integers which represent each node's position relating to each other (right)
        return right;
    }
    0 references
    public void InstantiateTree(int levels, string baseRoot)
    {
        if (treeInstantiated == false)
        {
            //the max amount of leaves at the bottom of the tree is 2^levels
            //we need to create enough spaces in the array for all the possible positions from the root to the leaves
            //formula = 2^levels + 2^levels-1 until levels < 0
            for (int i = levels; i >= 0; i--)
            {
                int x = Convert.ToInt32(Math.Pow(2, i));
                while (x > 0)
                {
                    nodes.Add("");
                    left.Add(-1);
                    right.Add(-1);
                    x--;
                }
            }
            //A node's position is the same number in each of the three lists
            nodes[0] = baseRoot;
            treeInstantiated = true;
        }
    }
```



```csharp
0 references
public void AddNode(string node)
{
    if (treeInstantiated != false)
    {
        if (nodes.Contains(node))
        {
            Console.WriteLine("Addition of node '" + node + "' unsuccessful as node already exists in the tree.");
        }
        else
        {
            //algorithm to find the next suitable location for the node
            int nodeCount = 1;
            while (nodes[nodeCount] != "")
            {
                //loops through the node list to find the next blank space
                nodeCount++;
            }
            nodes[nodeCount] = node;
            Console.WriteLine("Position of node '" + node + "' in the list of data is " + nodeCount);
            //nodeCount is the index for the new node which will be the same for all the arrays
            //next, the node needs to have a relational position to the other nodes which are stored in the left and right lists at the index nodeCount for each list


            #region Relating node to other nodes in the tree through comparing them
            int presentNode = 2;
            while (nodes[presentNode] != "")
            {
                //branch left or right?
                //for comparing nodes, we will order them alphabetically, thus the first character needs to be known in ASCII
                if (Convert.ToInt32(Encoding.ASCII.GetBytes(node)[0]) > Convert.ToInt32(Encoding.ASCII.GetBytes(nodes[presentNode])[0]))
                {
                    //if the first letter of the new node is before the first letter of the node at this position, then it will branch left
                    //if left branch is empty then assign node count
                    if (left[presentNode] == -1)
                    {
                        left[presentNode] = nodeCount;
                    }
                    presentNode = left[presentNode];
                }
                else
                {
                    //if right branch is empty then assign node count
                    if (right[presentNode] == -1)
                    {
                        right[presentNode] = nodeCount;
                    }
                    presentNode = right[presentNode];
                }
            }
            #endregion
        }
    }
}
```



```
    0 references
    public void DeleteNodeByName(string nodeName)
    {
        if (nodes.Contains(nodeName))
        {
            //algorithm to delete the node and its children
        }
        else
        {
            Console.WriteLine("Node not found!");
        }
    }
}
```



Quadratic Equation Solver

We can help you solve an equation of the form "$ax^2 + bx + c = 0$"
Just enter the values of a, b and c below:

a: 2, b: 5, c: -3

$2x^2 + 5x - 3 = 0$

Roots:
-3
0.5

Details:
$2x^2 + 5x - 3 = 0$

Roots: -3, 0.5
Root Pair: -5/4 ± 7/4
Factored: f(x) = 2(x + 3)(x - 0.5)

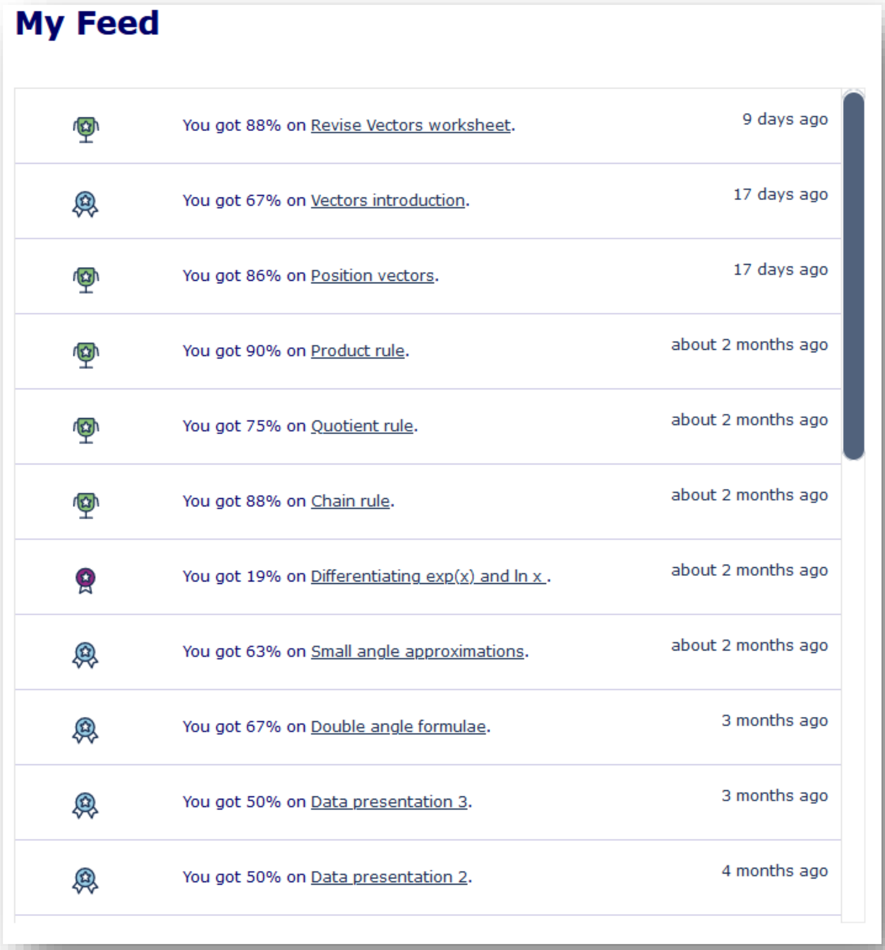
Discriminant: 49
Vertex: (-1.25, -6.125)
Sum of Roots (-b/a): -2.5
Product of Roots (c/a): -1.5

Values accurate to 12 places only, and 4 places in graph.

© 2018 MathsIsFun.com v0.876

This is a screenshot from the site https://www.mathsisfun.com/quadratic-equation-solver.html where the user can enter a quadratic's coefficients and determine the roots. In this example, it also draws a graph.

Each user has a login so that they can save their scores to their account and view them in future; logins and test scores are going to be stored in a multi-table SQLite database which will be local to the phone to allow for offline practice, which is effective if the user is practicing on their phone as they will be 'on-the-go' and may not have internet access for various reasons, e.g. no data or not enough data, no Wi-Fi, poor signal, etc.; overall, developing the app for a mobile device allows for convenient revision of certain topics through repetitive randomised practice. I have seen examples of being able to view test scores in a driving theory test app, where users can monitor progression over time. The site 'MyMaths' also has a similar system where you can see previous scores from tests and quizzes - see the screenshot below.


My Feed

You got 88% on Revise Vectors worksheet. 9 days ago

You got 67% on Vectors introduction. 17 days ago

You got 86% on Position vectors. 17 days ago

You got 90% on Product rule. about 2 months ago

You got 75% on Quotient rule. about 2 months ago

You got 88% on Chain rule. about 2 months ago

You got 19% on Differentiating exp(x) and ln x . about 2 months ago

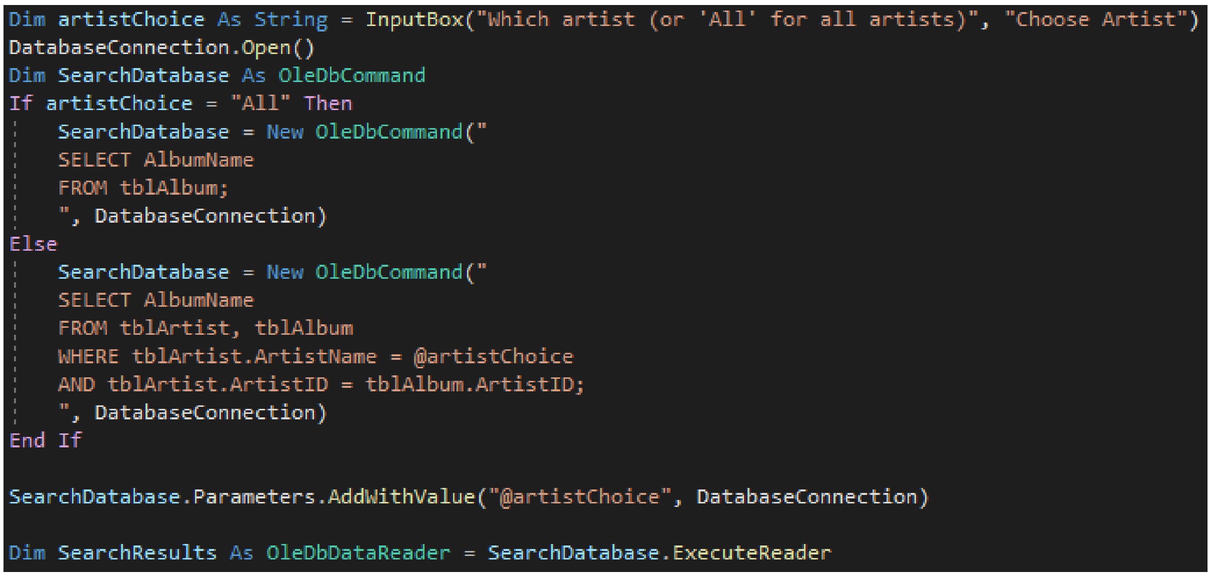
You got 63% on Small angle approximations. about 2 months ago

You got 67% on Double angle formulae. 3 months ago

You got 50% on Data presentation 3. 3 months ago

You got 50% on Data presentation 2. 4 months ago

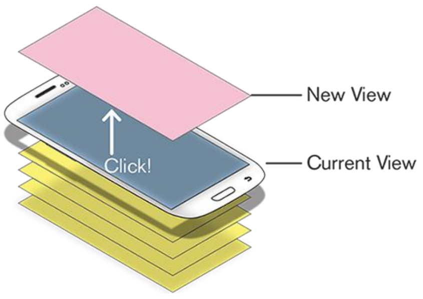
Below is an example of using parameterised SQL queries to search for results based on user input, which in the case of my app, will be based on the user clicking a button to load their personal leader board.


```
Dim artistChoice As String = InputBox("Which artist (or 'All' for all artists)", "Choose Artist")
DatabaseConnection.Open()
Dim SearchDatabase As OleDbCommand
If artistChoice = "All" Then
    SearchDatabase = New OleDbCommand("
    SELECT AlbumName
    FROM tblAlbum;
    ", DatabaseConnection)
Else
    SearchDatabase = New OleDbCommand("
    SELECT AlbumName
    FROM tblArtist, tblAlbum
    WHERE tblArtist.ArtistName = @artistChoice
    AND tblArtist.ArtistID = tblAlbum.ArtistID;
    ", DatabaseConnection)
End If

SearchDatabase.Parameters.AddWithValue("@artistChoice", DatabaseConnection)

Dim SearchResults As OleDbDataReader = SearchDatabase.ExecuteReader
```

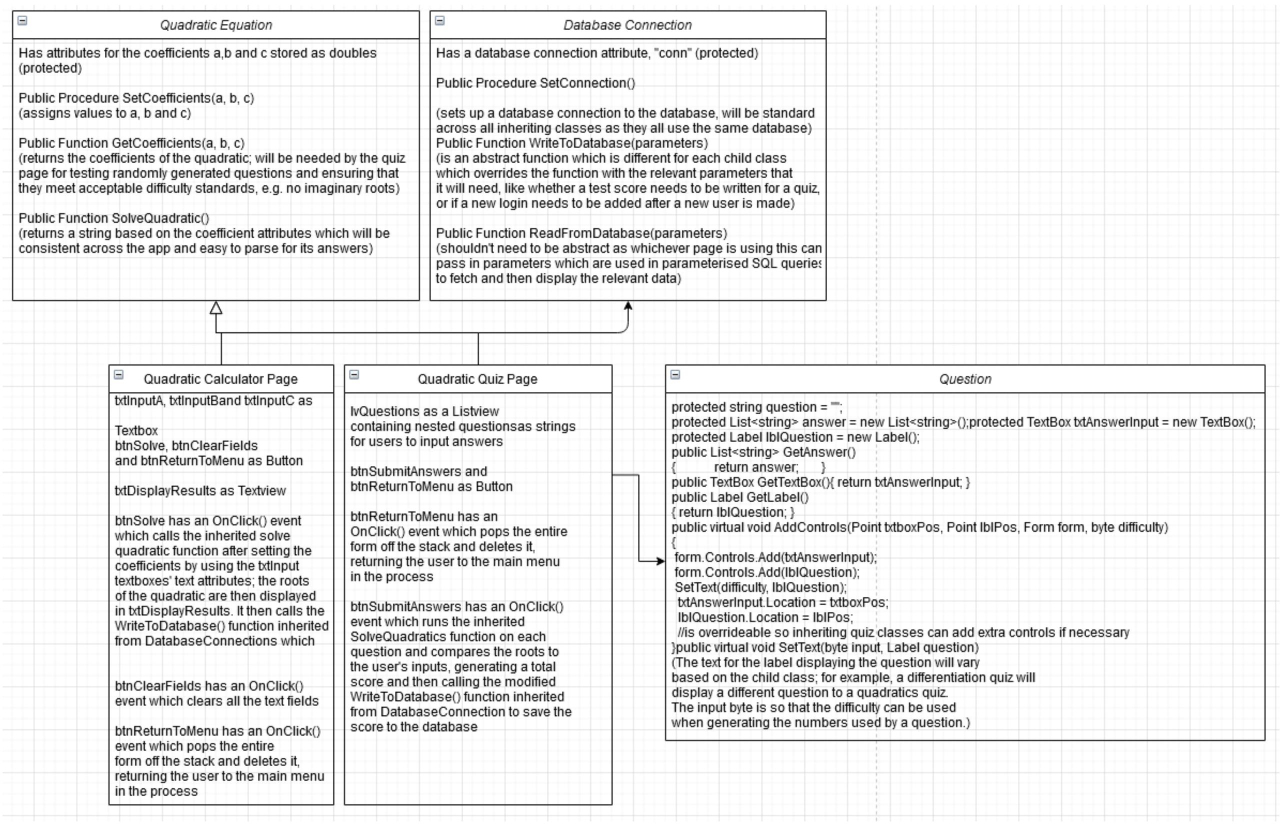
<u>More Specific Design and Modelling (still initial ideas)</u>

In Xamarin Forms, there are various classes for different widgets and UI tools, such as for buttons, list views and text entry fields. In a Xamarin Forms app, you design child classes that inherit from the Form class - arguably one of the most important classes in the libraries which is needed to create a page for your app on which you can put buttons and other widgets. When designing a page class, you oftentimes define it as a partial class whereby the UI elements of it, like buttons and other interactive items such as text entries, are written using a markup language (in this case, XAML) whilst the functions behind the buttons within the form are defined using C#; effectively, XAML is used to make the front-end (what the user sees and uses to input values and view outputs) whilst C# is used to make the back-end (the functionality that goes on behind the scenes, for example, clicking a button which is tied to a function to collect data from a database and display it in a listview for the user to view). https://docs.microsoft.com/en-us/xamarin/xamarin-forms/internals/class-hierarchy This link contains a class diagram for how all the objects in Xamarin Forms relate to each other and what they inherit from. Each page is a content page but also inherits from other classes like layout classes and view classes, making each type of page - e.g. a leader board page display, a quadratic solver, or a kinematics quiz generator – a unique class inheriting from the relevant classes (e.g. many pages will need to use text fields for user input as well as button inputs, whilst not every page may need to use scroll layouts and list views) which can be used as a template for creating more pages of the same type which get stacked onto each other and popped from the stack, explained in more detail in the paragraph below.

Once you have defined your classes, how the app works is by instantiating an object of a page class and then pushing the page object onto a stack, e.g. when you open an app, you might be met with a login page which is the only page on the stack; once logged in, the next page - which in our example could be an instance of a menu page for which part of the app to use (calculator, quiz questions, view test scores) - is pushed on top of the stack and shown to the user; if the user goes back, this page which is an instance of the menu class is then popped from the top of the stack and deleted and the user is taken to the item that is next on top of the stack, in this case, the login page. The app will be event-driven by the user whereby as they click buttons and enter different information (e.g. logging in and authenticating their details, submitting a quiz, requesting to view their previous scores), the relevant page objects with necessary data fed to them through parameters where needed will be pushed onto the stack frame. Below is a basic diagram of how it works, where new views literally get stacked onto the current view each time like the stack data structure.



Each page will need to inherit from a class for writing data to the SQLite database as a lot of the code will be the same or have the same method names, thus using abstract functions which can be overridden by each inheriting class as well as constant subroutines that are the same for each class all contained in one class will be a clean way of doing this – if the contents of the functions are different but the names are the same then incorporating an interface could work out better as how an interface works is it contains functions which don't actually do anything and are abstract, as in they are intended to be overridden by inheriting classes but all share the same name so that objects can interact with each other and call the methods that they share with each other. A similar occurrence will happen with the quiz pages and the calculator pages where both a quadratics quiz and quadratics solver will need to inherit the exact same method, "SolveQuadratic", as well as with the differentiation quiz and differentiation solver pages needing to inherit methods to solve differentials and integrals, so creating base classes that each calculator/quiz pair can derive from will be the cleanest and most efficient method for covering this. Below is the screenshot from the analysis of a class diagram showing how a page will inherit from a database connection class.



Quadratic Equation

Has attributes for the coefficients a,b and c stored as doubles (protected)

Public Procedure SetCoefficients(a, b, c)
(assigns values to a, b and c)

Public Function GetCoefficients(a, b, c)
(returns the coefficients of the quadratic; will be needed by the quiz page for testing randomly generated questions and ensuring that they meet acceptable difficulty standards, e.g. no imaginary roots)

Public Function SolveQuadratic()
(returns a string based on the coefficient attributes which will be consistent across the app and easy to parse for its answers)

Database Connection

Has a database connection attribute, "conn" (protected)

Public Procedure SetConnection()

(sets up a database connection to the database, will be standard across all inheriting classes as they all use the same database)
Public Function WriteToDatabase(parameters)
(is an abstract function which is different for each child class which overrides the function with the relevant parameters that it will need, like whether a test score needs to be written for a quiz, or if a new login needs to be added after a new user is made)

Public Function ReadFromDatabase(parameters)
(shouldn't need to be abstract as whichever page is using this can pass in parameters which are used in parameterised SQL queries to fetch and then display the relevant data)

Quadratic Calculator Page

txtInputA, txtInputBand txtInputC as

Textbox
btnSolve, btnClearFields
and btnReturnToMenu as Button

txtDisplayResults as Textview

btnSolve has an OnClick() event which calls the inherited solve quadratic function after setting the coefficients by using the txtInput textboxes' text attributes; the roots of the quadratic are then displayed in txtDisplayResults. It then calls the WriteToDatabase() function inherited from DatabaseConnections which

btnClearFields has an OnClick() event which clears all the text fields

btnReturnToMenu has an OnClick() event which pops the entire form off the stack and deletes it, returning the user to the main menu in the process

Quadratic Quiz Page

lvQuestions as a Listview containing nested questionsas strings for users to input answers

btnSubmitAnswers and btnReturnToMenu as Button

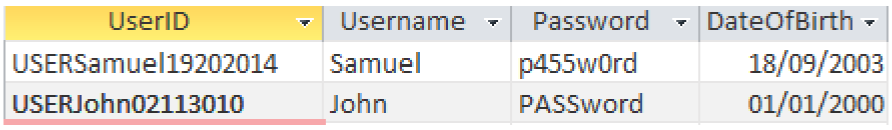
btnReturnToMenu has an OnClick() event which pops the entire form off the stack and deletes it, returning the user to the main menu in the process

btnSubmitAnswers has an OnClick() event which runs the inherited SolveQuadratics function on each question and compares the roots to the user's inputs, generating a total score and then calling the modified WriteToDatabase() function inherited from DatabaseConnection to save the score to the database

Question

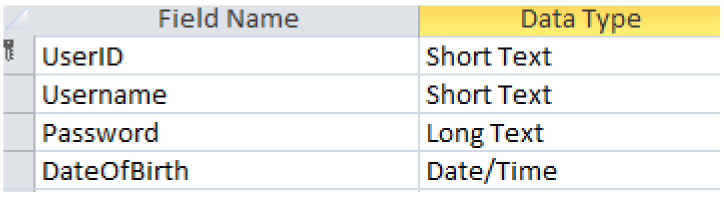
```
protected string question = "";
protected List<string> answer = new List<string>();protected TextBox txtAnswerInput = new TextBox();
protected Label lblQuestion = new Label();
public List<string> GetAnswer()
{          return answer;      }
public TextBox GetTextBox(){ return txtAnswerInput; }
public Label GetLabel()
{ return lblQuestion; }
public virtual void AddControls(Point txtboxPos, Point lblPos, Form form, byte difficulty)
{
 form.Controls.Add(txtAnswerInput);
 form.Controls.Add(lblQuestion);
 SetText(difficulty, lblQuestion);
  txtAnswerInput.Location = txtboxPos;
  lblQuestion.Location = lblPos;
  //is overrideable so inheriting quiz classes can add extra controls if necessary
}public virtual void SetText(byte input, Label question)
(The text for the label displaying the question will vary
based on the child class; for example, a differentiation quiz will
display a different question to a quadratics quiz.
The input byte is so that the difficulty can be used
when generating the numbers used by a question.)
```

In terms of the database format, it will be a multi-table relational database. The first table will be a user table, where each user has the following attributes: Username, DateOfBirth, Password and UserID. To create a UserID for a user, the program will take the first letter of their Username, convert it into its 7-bit ASCII value with an extra 0 at the start, e.g. in the following example, Samuel --> S --> 115 denary --> 1110011 binary --> 01110011 as a string; this string is then treated like a denary number and added to the user's DateOfBirth, e.g. if my date of birth is 18/09/2003 then this results in 01110011+ 18092003 = 19202014 which is to be my UserNumber; my final UserID is then "USERSamuel19202014" - "USER" + [Username] + [UserNumber]. If there are multiple users with the same final UserID then the later one to be added will have an extra "1" at the end of their UserID, e.g. "USERSamuel192020141" would be the UserID if another user named Samuel with a DateOfBirth of 18/09/2003 created a login. For a user called John born on 01/01/2000, generating their UserID would be as follows: John --> J --> 106 --> 1101010 --> 01110011; 01110011 + 01012000 = 02113010 --> (if this username and usernumber are unique) USERJohn02113010. **UPDATE: Realised that the user would have to remember their UserID to log in each time if there were multiple users with the same name; instead, if a new user is added with the same username as an existing user, their name will simply have a number after it, e.g. Samuel1, Samuel2, etc.**



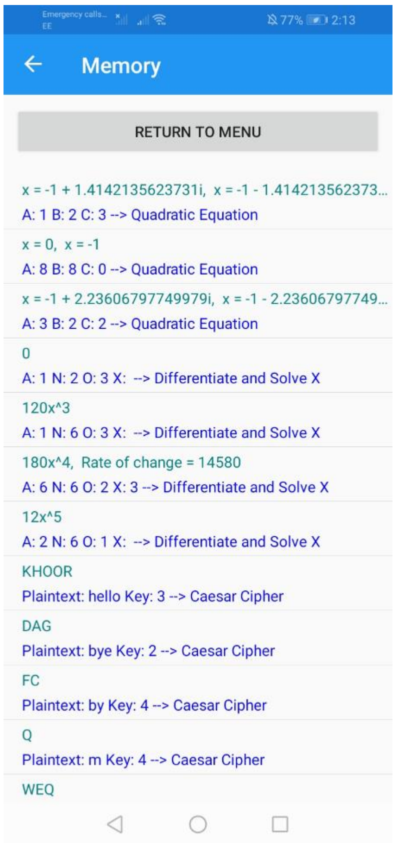
UserID	Username	Password	DateOfBirth
USERSamuel19202014	Samuel	p455w0rd	18/09/2003
USERJohn02113010	John	PASSword	01/01/2000

(sample data)



Field Name	Data Type
UserID	Short Text
Username	Short Text
Password	Long Text
DateOfBirth	Date/Time

The next table is the Calculations table. Each calculation has the following attributes: CalculationID, UserID, Inputs, Output and CalculationType. CalculationID is the primary key and autoincrement, whilst UserID is the foreign key which is tied to the UserID primary key in the User table. The CalculationType is needed by the app when reading the database and loading it into a page in order to allow it to know what to show, e.g. the field Inputs and Output will be parsed based on whether the calculation that was performed was a quadratic getting solved or an expression being integrated. My prototype app shows how this is done to some degree (see screenshot below, alongside the sample data for this table).



x = -1 + 1.4142135623731i, x = -1 - 1.414213562373...
A: 1 B: 2 C: 3 --> Quadratic Equation

x = 0, x = -1
A: 8 B: 8 C: 0 --> Quadratic Equation

x = -1 + 2.23606797749979i, x = -1 - 2.23606797749...
A: 3 B: 2 C: 2 --> Quadratic Equation

0
A: 1 N: 2 O: 3 X: --> Differentiate and Solve X

120x^3
A: 1 N: 6 O: 3 X: --> Differentiate and Solve X

180x^4, Rate of change = 14580
A: 6 N: 6 O: 2 X: 3 --> Differentiate and Solve X

12x^5
A: 2 N: 6 O: 1 X: --> Differentiate and Solve X

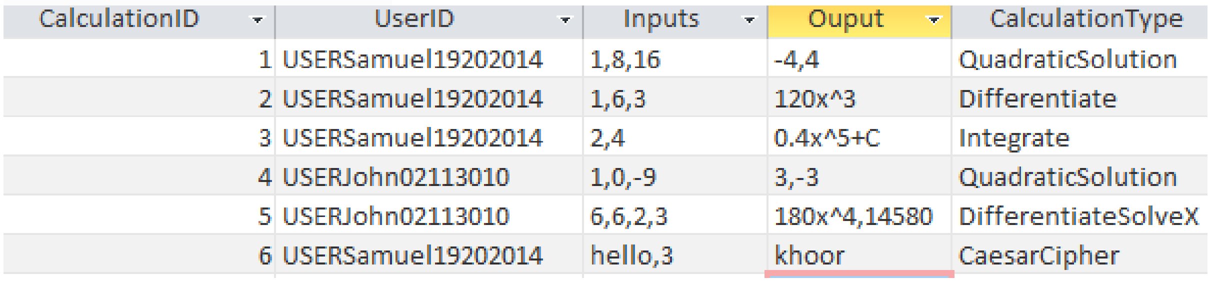
KHOOR
Plaintext: hello Key: 3 --> Caesar Cipher

DAG
Plaintext: bye Key: 2 --> Caesar Cipher

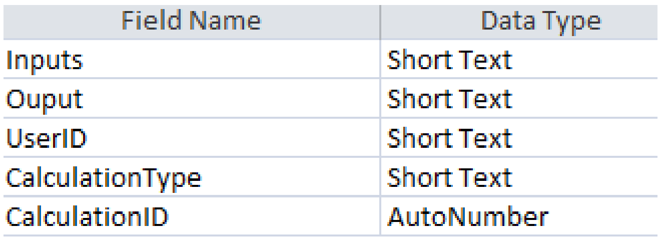
FC
Plaintext: by Key: 4 --> Caesar Cipher

Q
Plaintext: m Key: 4 --> Caesar Cipher

WEQ

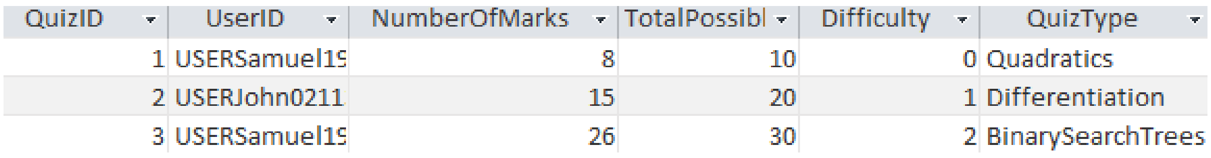


CalculationID	UserID	Inputs	Ouput	CalculationType
1	USERSamuel19202014	1,8,16	-4,4	QuadraticSolution
2	USERSamuel19202014	1,6,3	120x^3	Differentiate
3	USERSamuel19202014	2,4	0.4x^5+C	Integrate
4	USERJohn02113010	1,0,-9	3,-3	QuadraticSolution
5	USERJohn02113010	6,6,2,3	180x^4,14580	DifferentiateSolveX
6	USERSamuel19202014	hello,3	khoor	CaesarCipher

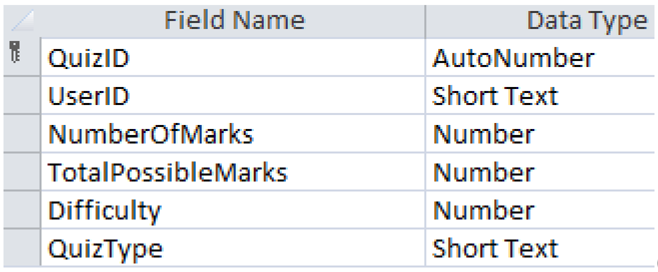


Field Name	Data Type
Inputs	Short Text
Ouput	Short Text
UserID	Short Text
CalculationType	Short Text
CalculationID	AutoNumber

The third table in the database is Quizzes and has the following attributes: QuizID, UserID, NumberOfMarks, TotalPossibleMarks, Difficulty, QuizType and isCompleted. The QuizID is the primary key and simply autoincrement while the UserID is the foreign key which links to the primary key UserID in the User table. The NumberOfMarks and TotalPossibleMarks are fairly self-explanatory field: the program will be able to generate a score or percentage using them when loading the table into a page for the user to view. QuizType is also straightforward: it tells the program what the quiz was on, e.g. quadratics, binary trees, differentiation, or maybe for a difficult mode, multiple topics that are combined in each question, like differentiating a cubic equation and then finding the value of x for the resulting quadratic. For quizzes of this nature, they will likely be composed of a common theme like, say, quadratics, and use other elements of maths to create difficult questions like using integrals, or finding a root of a polynomial and then applying that to a Caesar cipher to get an encrypted message. The last attribute in the table is difficulty, which will be 0 for easy, 1 for normal and 2 for hard mode; again, the app will parse the number and show the data is a user-friendly format, e.g. if(difficulty == 0) { txtDisplay.Text = "easy"; } The isCompleted attribute is a boolean set to true or false; On the app, when a button that says "Submit Answers" is clicked, isComplete is set to true, else it is always false. When generating a new quiz, you can click a button to go and view unfinished quizzes, i.e. Quizzes where submit answers wasn't clicked, to finish them, whereby it recreates the quiz on a new page pushed onto the stack frame but with input boxes filled in and questions grabbed from the questions table in the database (the last table) which has a one to one relationship with the quizzes table.

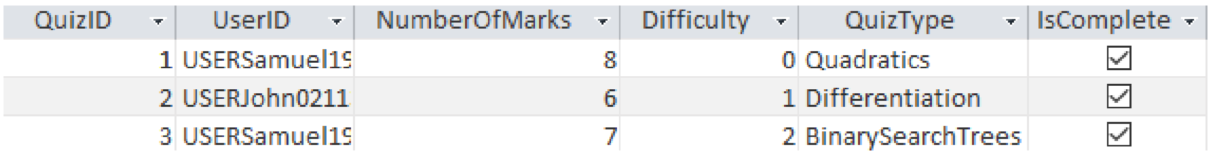


QuizID	UserID	NumberOfMarks	TotalPossibl	Difficulty	QuizType
1	USERSamuel19	8	10	0	Quadratics
2	USERJohn0211	15	20	1	Differentiation
3	USERSamuel19	26	30	2	BinarySearchTrees

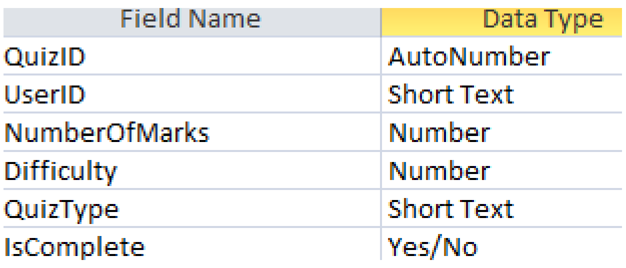


Field Name	Data Type
QuizID	AutoNumber
UserID	Short Text
NumberOfMarks	Number
TotalPossibleMarks	Number
Difficulty	Number
QuizType	Short Text

(these screenshots haven't been updated to show the attribute, "isCompleted". **~~UPDATE: In order to keep things consistent across the database, it makes more sense to make each quiz type the same number of questions regardless of difficulty, otherwise in the questions table, there would be blank columns for easy and regular mode questions with only advanced mode questions having all columns, e.g. question 30, so the column total possible marks has been removed, with only the number of marks being necessary to calculate the percentage scored if each quiz is the same size.~~ Thought that the question table would end up having 30 question attributes to accommodate for the total number of questions in each difficulty which would end up leaving loads of blank spaces for easy and regular questions, however I realised that this wouldn't be the case as each question has to have only its text and the required answer for attributes and then each unique question is tied to a quiz by its QuizID so as many questions as are necessary for a quiz can be inserted into the table with no unnecessary empty space. Below are updated screenshots which show the removal of TotalPossibleMarks (which isn't needed regardless of whether quizzes have equal numbers of questions regardless of difficulty due to the fact that the app can work out how many total marks should be in each quiz by fetching the difficulty attribute from the database) and the addition of isComplete :**

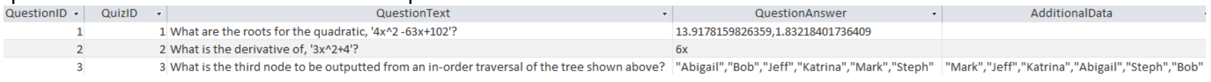


QuizID	UserID	NumberOfMarks	Difficulty	QuizType	IsComplete
1	USERSamuel19	8	0	Quadratics	✓
2	USERJohn0211	6	1	Differentiation	✓
3	USERSamuel19	7	2	BinarySearchTrees	✓

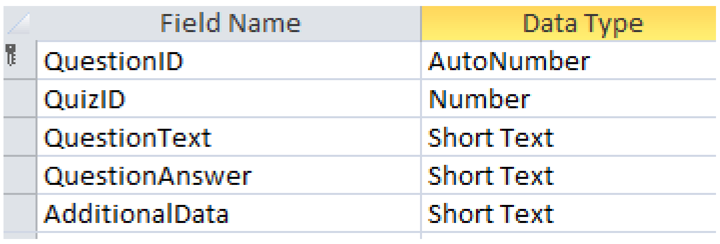


Field Name	Data Type
QuizID	AutoNumber
UserID	Short Text
NumberOfMarks	Number
Difficulty	Number
QuizType	Short Text
IsComplete	Yes/No

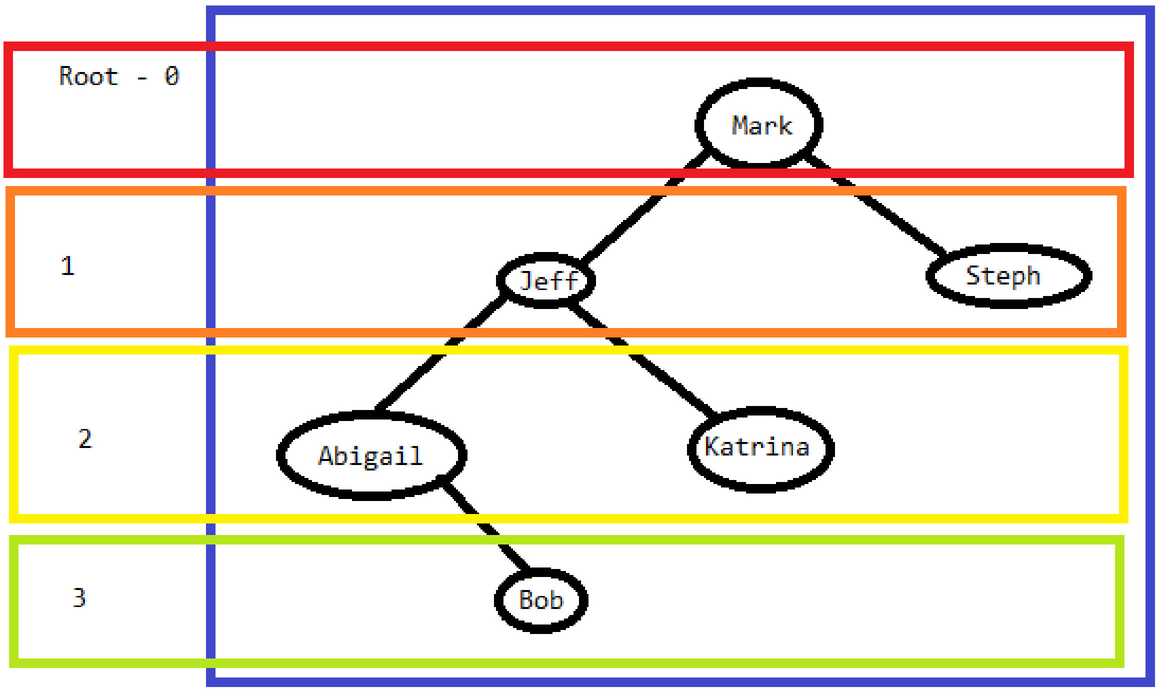
The last table, as aforementioned above, is called, "Questions". It has a primary key of QuestionID which is just AutoNumber. More importantly is its foreign key which is a QuizID; this will allow the app to regenerate an unfinished quiz if isComplete for a quiz is set to false by looking for all questions that have the same QuizID as the desired quiz using SQL along the lines of "SELECT QuestionText FROM tblQuizzes, tblQuestions, WHERE tblQuizzes.QuizID = tblQuestions.QuizID". There will also be an attribute called, "AdditionalData" where any extra information regarding the question is stored, for example, parameters for instantiating a binary search tree like a comma-separated list of names to insert which can be parsed to make an array once the app reads it from the database. Below is a demonstration of a binary search tree and the question that could be asked with it incorporated into a screenshot of the table's attributes.



QuestionID	QuizID	QuestionText	QuestionAnswer	AdditionalData
1	1	What are the roots for the quadratic, '4x^2 -63x+102'?	13.9178159826359,1.83218401736409	
2	2	What is the derivative of, '3x^2+4'?	6x	
3	3	What is the third node to be outputted from an in-order traversal of the tree shown above?	"Abigail","Bob","Jeff","Katrina","Mark","Steph"	"Mark","Jeff","Katrina","Abigail","Steph","Bob"

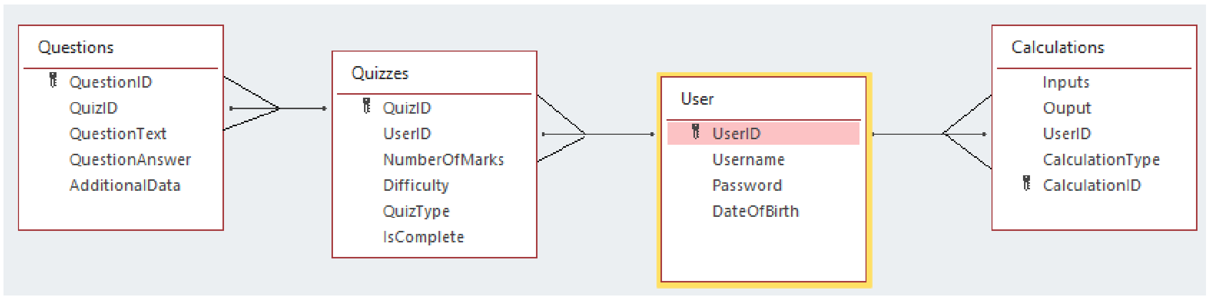


Field Name	Data Type
QuestionID	AutoNumber
QuizID	Number
QuestionText	Short Text
QuestionAnswer	Short Text
AdditionalData	Short Text

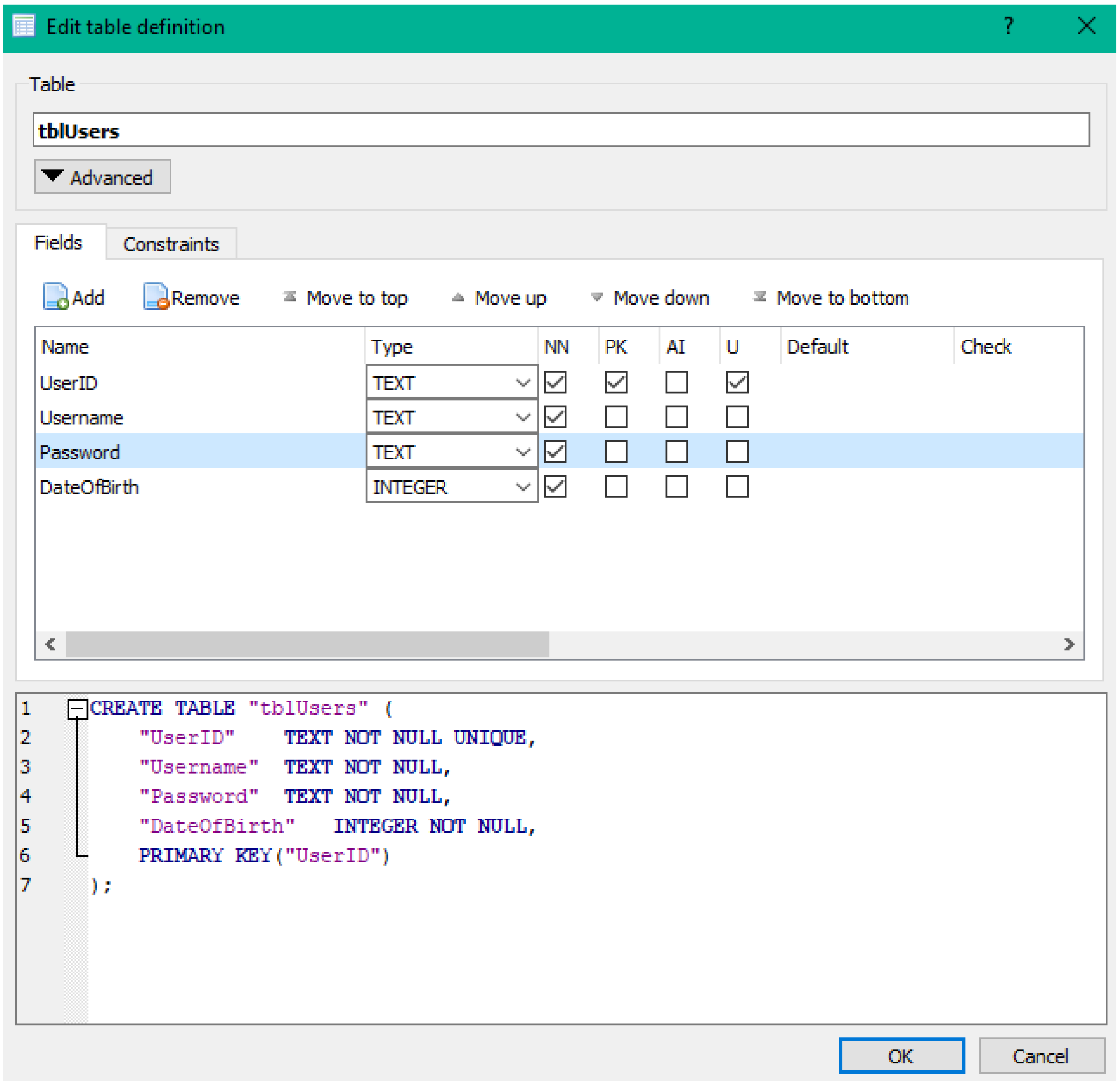


To sort the questions in order when regenerated from an unfinished quiz, all questions belonging to the same quiz will first be selected and then because it is a small set of only ten items, a bubble sort will be applied so that the questions with lower QuestionIDs (which were added to the database earlier than those with larger ones due to autoincrement) will be at the start of the list and then added sequentially to a listview as if the quiz were a new quiz, with the answers already submitted being put into the relevant text boxes either alongside the listview or below it.

The relationships between the tables will be as follows: User has one-to-many with Quizzes via UserID; and User has one-to-many with Calculations via UserID. This is because each user can have many quizzes whilst each randomly generated quiz can only be done by one user and each calculation performed is performed by one user at a time whilst one user can perform many calculations with their account. Quizzes then has a one-to-many relationship with Questions as each quiz can have many questions whereas each question belongs to only one quiz (technically you could have two questions generated on separate occasions which use the same exact parameters but that doesn't make them the same 'question' - it is part of a different quiz on a different day).



UPDATE: When developing the table using a SQLite database tool after modelling a prototype of what it would look like using Access, I found that Date/Time isn't a valid data type, as shown below:

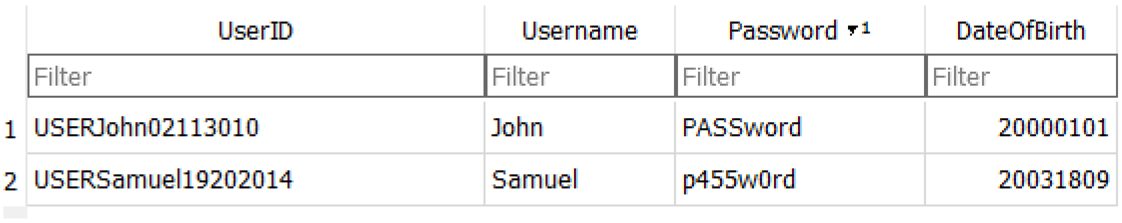


tblUsers

Name	Type	NN	PK	AI	U	Default	Check
UserID	TEXT	✓	✓		✓		
Username	TEXT	✓					
Password	TEXT	✓					
DateOfBirth	INTEGER	✓					

```
CREATE TABLE "tblUsers" (
    "UserID"    TEXT NOT NULL UNIQUE,
    "Username"  TEXT NOT NULL,
    "Password"  TEXT NOT NULL,
    "DateOfBirth"   INTEGER NOT NULL,
    PRIMARY KEY("UserID")
);
```

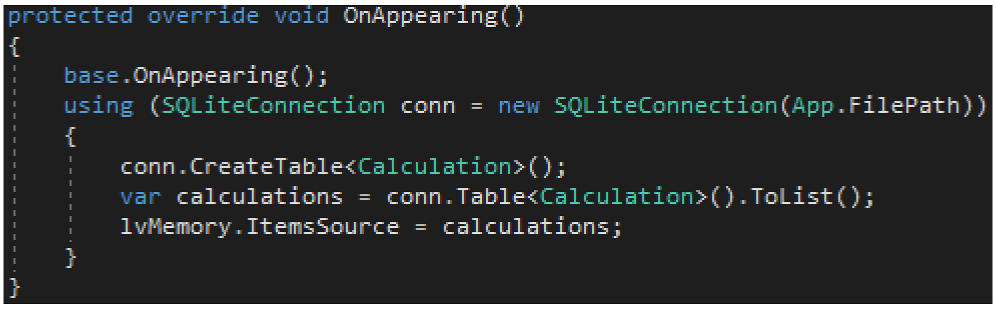
Thus, I decided I'd save dates as integers in the format YYYY/DD/MM like in the below example:



	UserID	Username	Password	DateOfBirth
1	USERJohn02113010	John	PASSword	20000101
2	USERSamuel19202014	Samuel	p455w0rd	20031809

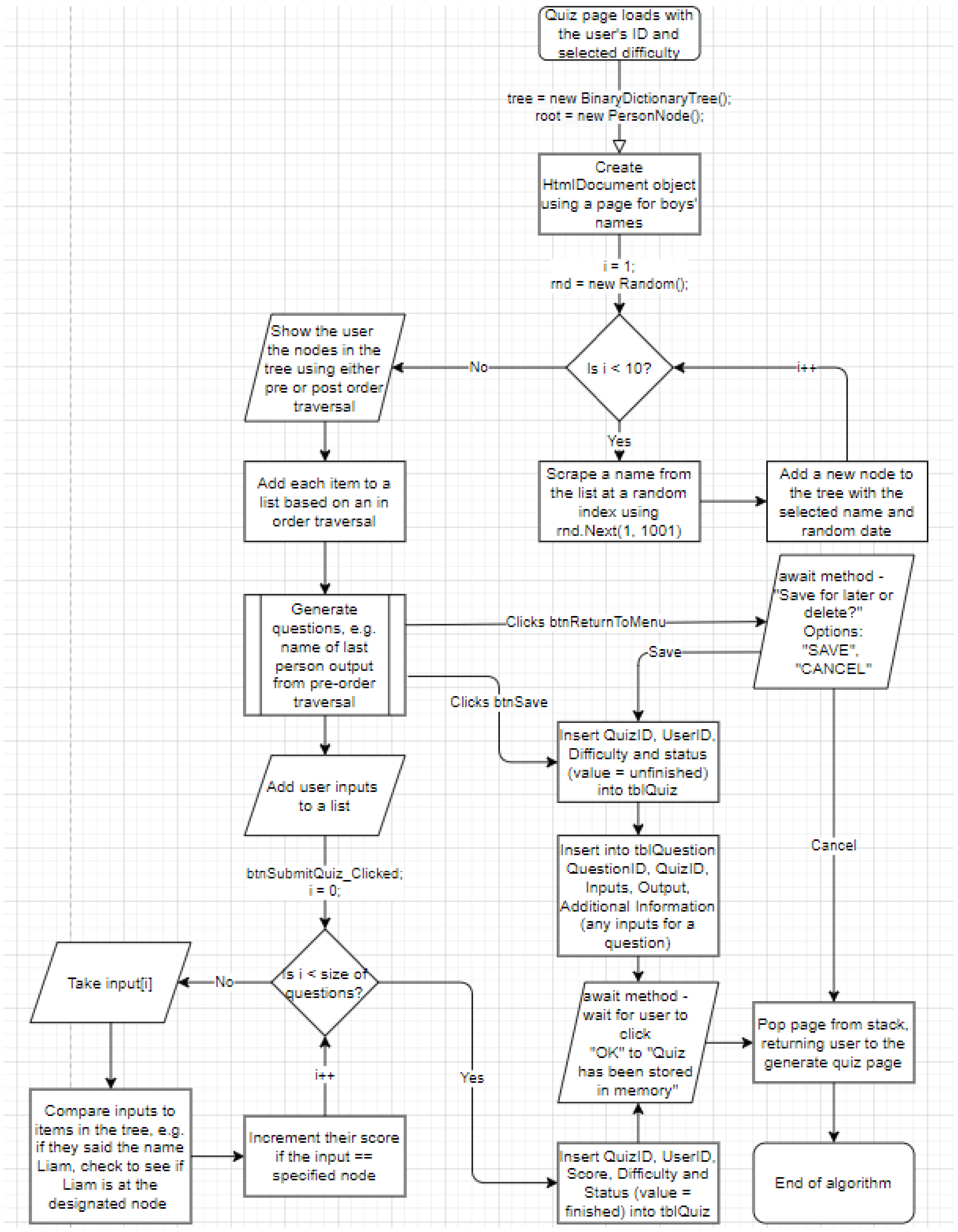
The reason for this is because if I didn't start with the year, somebody could have a birthday where the day or month is a single digit number, like in the format MM/DD/YYYY my birthday would be 09182003 which causes it to be stored as 9182003, which creates inconsistencies across data as some dates would be stored as 7 digits and others as 8 digits; keeping everything uniform by storing dates in this format is easier to parse and process.

Each quiz page will have a listview of the questions; how a listview works is that it uses data-binding whereby the collection of items in the listview is equal to a list of items. In this case, before the page is loaded and as it is being created, the function, "OnAppearing()" will be used to declare the ItemsSource attribute of the listview, "lvQuestions" to be set to a list of the relevant questions which will be either grabbed from the database based on the unfinished quiz that the user selected that they wanted to finish or a new quiz to be generated where the parameters from the QuizSettings page like QuizType and QuizDifficulty will be passed through to the Quiz page and used to create a randomised set of questions for the list. In a maths app prototype that I completed, I used this on the memory page to grab previous calculations performed for the user to view; **UPDATE: the data in the quiz settings page doesn't need to be bound to lists as the settings are the same and don't change, unlike memory where new scores get added, along with new users and new calculations, meaning binding data to lists created from a database is the most effective route to take. In the context of a pre-determined set of options for the user to choose from, a table view is best due to the lack of needing to bind data, however, I don't see much of a necessity to use a table view as its default data entries are limited to text boxes and switch cells (on or off buttons effectively); there is the option to make custom cells but ultimately all that would entail is using buttons for my scenario. I tried incorporating switch cells but there seemed to be issues with Visual Studio not being able to declare OnChanged() events so due to this and partially because of the extra data that would be taken up storing details about the table view and various custom cells within it that isn't ultimately benefitting user experience or performance, I decided to just make a grid of buttons as event handling for them is far more versatile and simple to pull off.** Below is a screenshot of the memory page in my prototype, where a connection to the database is made, the relevant data is grabbed (in this case, all the calculations in the table) and the ItemsSource is set to this list of data; this style of coding will not be obsolete simply because of a change to the QuizSettings that I have realised during developmental stages as it is still relevant to use it for reloading scores and calculations from memory where datasets update as new items get added.



```
protected override void OnAppearing()
{
    base.OnAppearing();
    using (SQLiteConnection conn = new SQLiteConnection(App.FilePath))
    {
        conn.CreateTable<Calculation>();
        var calculations = conn.Table<Calculation>().ToList();
        lvMemory.ItemsSource = calculations;
    }
}
```

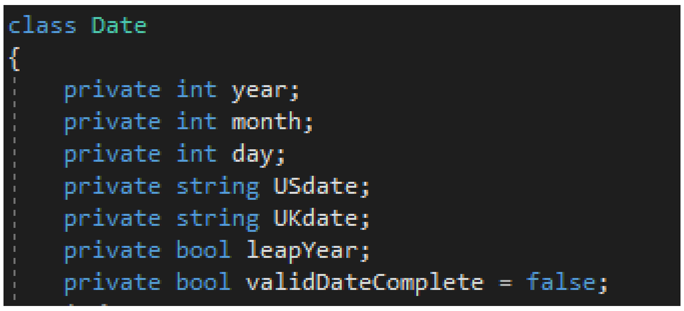
In the binary search tree quiz page, a binary tree is constructed with each node containing a dictionary (representing a person) for its data. In the dictionary, there will be only one key-value-pair, with the key being the person's name and the value being their date of birth, created using a Date class that I have written. When instantiating the tree, the names and dates need to be random to make sure new questions are generated each time a binary tree quiz page is loaded. Thus, I have used the NuGet package, 'HtmlAgilityPack', in order to perform webscraping and find lists of names from websites to use. The full X-path of the list of names within the page is taken and a random number between 1 and 1000 is generated to pick one of the thousand list item within the list, each of which obviously holds a name. The randomly selected name is then fed through to the function, 'InsertNode()', along with a date object which has the method, 'GenerateRandomDate()' applied to it. Below is a flowchart of the rough algorithm that will be followed to create a functional binary tree quiz page.


Quiz page loads with the user's ID and selected difficulty
tree = new BinaryDictionaryTree();
root = new PersonNode();
Create HtmlDocument object using a page for boys' names
i = 1;
rnd = new Random();
Is i < 10?
No
Yes
i++
Show the user the nodes in the tree using either pre or post order traversal
Scrape a name from the list at a random index using rnd.Next(1, 1001)
Add a new node to the tree with the selected name and random date
Add each item to a list based on an in order traversal
Generate questions, e.g. name of last person output from pre-order traversal
Clicks btnReturnToMenu
await method - "Save for later or delete?" Options: "SAVE", "CANCEL"
Save
Clicks btnSave
Insert QuizID, UserID, Difficulty and status (value = unfinished) into tblQuiz
Add user inputs to a list
Cancel
Insert into tblQuestion QuestionID, QuizID, Inputs, Output, Additional Information (any inputs for a question)
btnSubmitQuiz_Clicked;
i = 0;
Take input[i]
No
Is i < size of questions?
await method - wait for user to click "OK" to "Quiz has been stored in memory"
Pop page from stack, returning user to the generate quiz page
i++
Yes
Compare inputs to items in the tree, e.g. if they said the name Liam, check to see if Liam is at the designated node
Increment their score if the input == specified node
Insert QuizID, UserID, Score, Difficulty and Status (value = finished) into tblQuiz
End of algorithm

Final Design

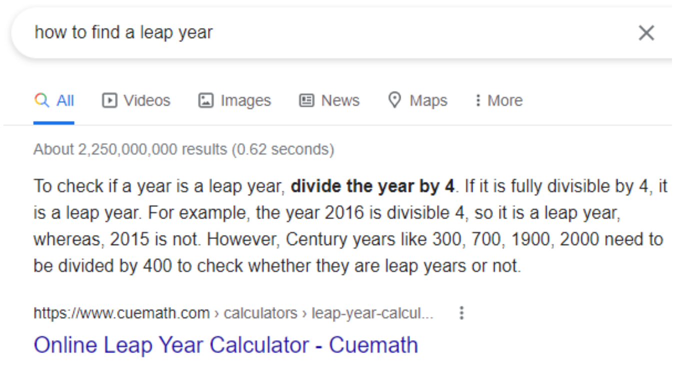
In my data structures quiz, I made several questions based on binary search trees. One of these uses a tree containing people based on a person node class that I made, where each person's main data consists of a single key-value pair: the key is their name, and their value is their date of birth. When inserting people into the tree in the quiz, they are ordered based on youngest to oldest. Using key value pairs makes it easy to find someone's birthday as it is directly associated to their name. It also allows the person's data to be stored in different formats easily, such as in dictionaries.

In order to store birthdays in a standardised way with any methods that I want to use on dates, I created a date class, with each date object containing the following attributes: int values { year, month, day } , string values { USdate, UKdate } , Boolean values { leapYear, validDateComplete }. These attributes have fairly self-explanatory names: year is the year part of the date, month is the month part, day is the day part, US and UK dates are the dates written in the forms, "mm/dd/yyyy" and "dd/mm/yyyy" respectively; leapYear is either true or false depending on whether the year attribute corresponds to a leap year or not; and validDateComplete is true if the date held by the Date object is a valid date (tested with an algorithm) and false if not – by default, it is false;



```
class Date
{
    private int year;
    private int month;
    private int day;
    private string USdate;
    private string UKdate;
    private bool leapYear;
    private bool validDateComplete = false;
```

One key thing the date class requires is a way of setting the values of year, month and day and then verifying that they are valid – SetDate() does just this. First of all, we need to determine if the year entered is a leap year.



how to find a leap year

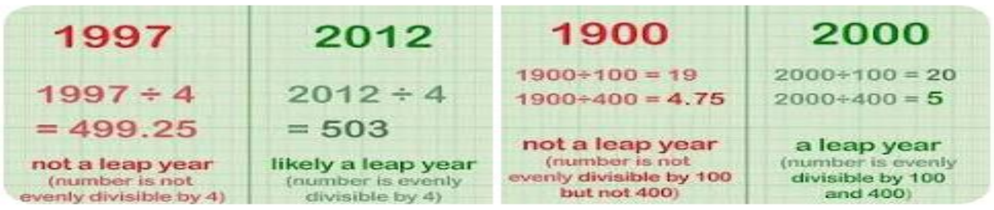
All Videos Images News Maps More

About 2,250,000,000 results (0.62 seconds)

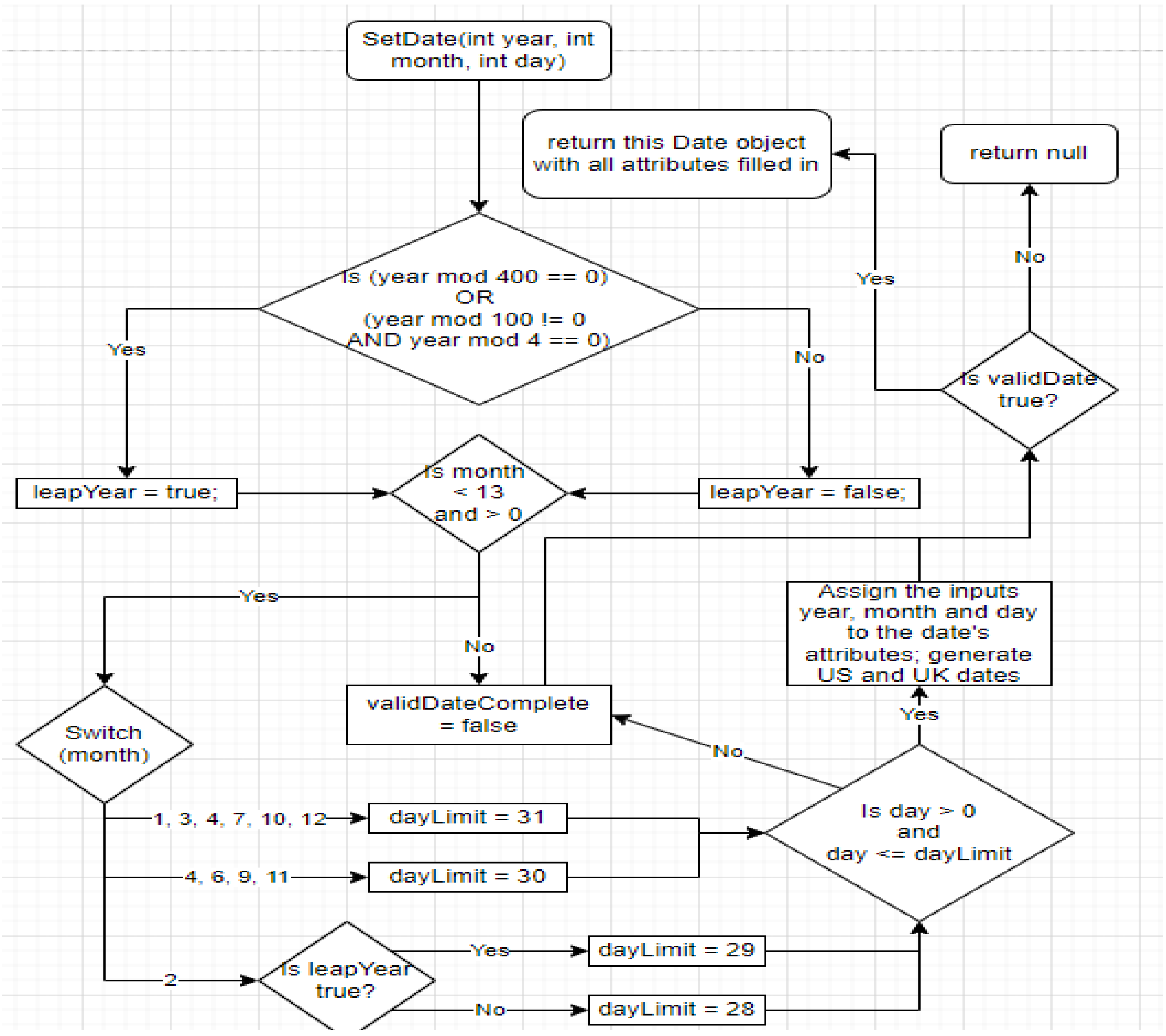
To check if a year is a leap year, **divide the year by 4**. If it is fully divisible by 4, it is a leap year. For example, the year 2016 is divisible 4, so it is a leap year, whereas, 2015 is not. However, Century years like 300, 700, 1900, 2000 need to be divided by 400 to check whether they are leap years or not.

https://www.cuemath.com › calculators › leap-year-calcul...

Online Leap Year Calculator - Cuemath

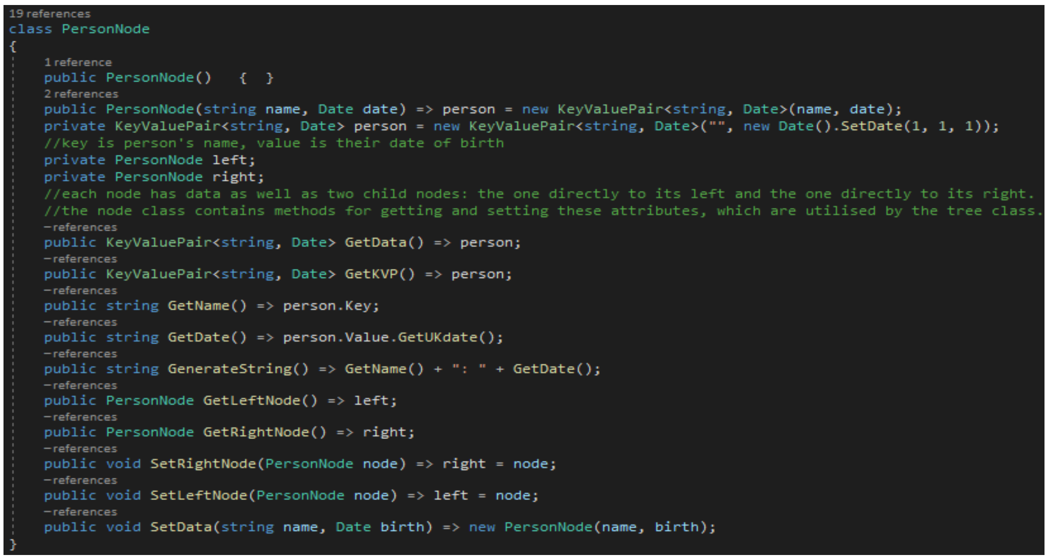

1997
1997 ÷ 4
= 499.25
not a leap year
(number is not evenly divisible by 4)
2012
2012 ÷ 4
= 503
likely a leap year
(number is evenly divisible by 4)
1900
1900÷100 = 19
1900÷400 = 4.75
not a leap year
(number is not evenly divisible by 100 but not 400)
2000
2000÷100 = 20
2000÷400 = 5
a leap year
(number is evenly divisible by 100 and 400)

After this, we need to check that the month entered is a valid month (between 1 and 12 inclusive) and determine the number of days in that month, using the information on whether the year is a leap year or not to determine February's days as there are 29 in a leap year and 28 usually. Next, we check that the number of days entered for the date are valid for the month. Once all the checks on the month and days have been performed, the values can be assigned to the date and the object is validated for use. Below is a flowchart of the SetDate algorithm.


SetDate(int year, int month, int day)
return this Date object with all attributes filled in
return null
Is (year mod 400 == 0) OR (year mod 100 != 0 AND year mod 4 == 0)
Yes
No
Yes
No
Is validDate true?
leapYear = true;
Is month < 13 and > 0
leapYear = false;
Yes
No
Assign the inputs year, month and day to the date's attributes; generate US and UK dates
validDateComplete = false
Yes
No
Switch (month)
Is day > 0 and day <= dayLimit
1, 3, 4, 7, 10, 12
dayLimit = 31
4, 6, 9, 11
dayLimit = 30
Is leapYear true?
Yes
dayLimit = 29
2
No
dayLimit = 28

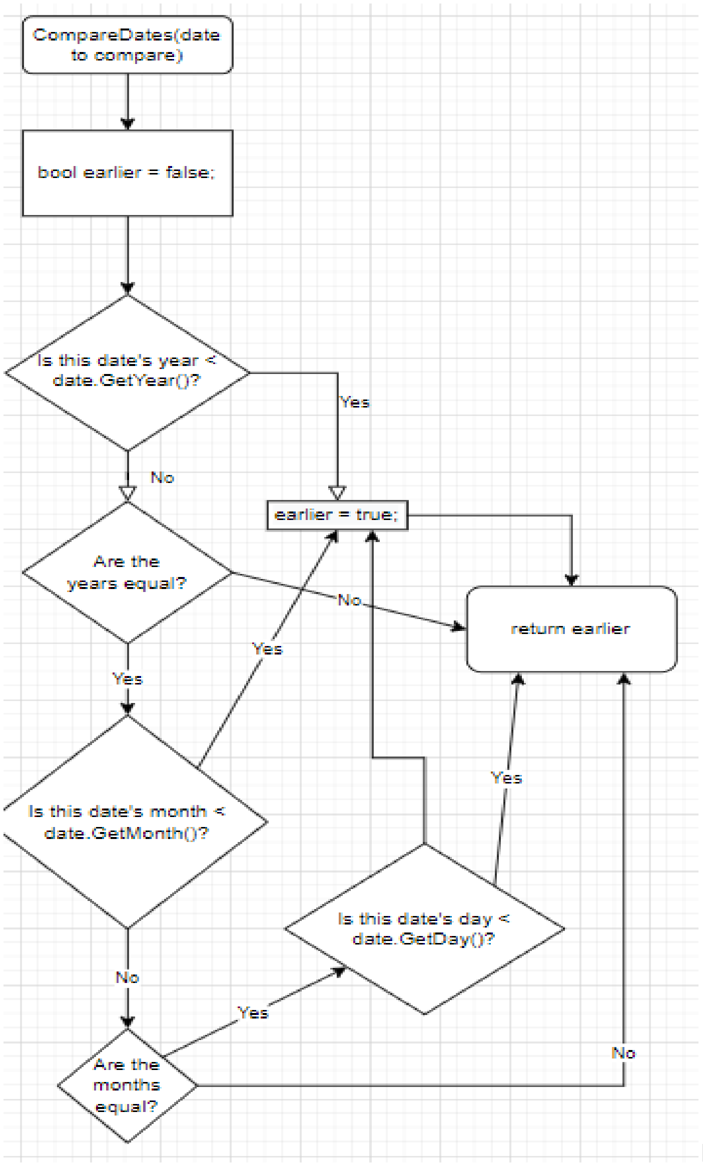
When inserting people into our binary tree, we are going to need a method of evaluating the date of the person to be inserted and the date of the person currently pointed to in the tree. Therefore, I made a CompareDates function in order to determine this. Further below is the relevant flowchart.

Immediately below is the class for PersonNode. Each PersonNode object has a key-value pair for their main data, holding their name as a string and their birthday as a Date object. They also have two PersonNode objects as further attributes: one is the left PersonNode (people who are born before them get inserted to their left) and one is the right PersonNode (people who are born after them get inserted to their right). The DictionaryTree class is what is used to manipulate PersonNodes and assign their left-right nodes to create a tree; the PersonNode class mainly has getters and setters in lambda expressions to take up less space on the screen when I am coding.



```
19 references
class PersonNode
{
    1 reference
    public PersonNode()   {  }
    2 references
    public PersonNode(string name, Date date) => person = new KeyValuePair<string, Date>(name, date);
    private KeyValuePair<string, Date> person = new KeyValuePair<string, Date>("", new Date().SetDate(1, 1, 1));
    //key is person's name, value is their date of birth
    private PersonNode left;
    private PersonNode right;
    //each node has data as well as two child nodes: the one directly to its left and the one directly to its right.
    //the node class contains methods for getting and setting these attributes, which are utilised by the tree class.
    -references
    public KeyValuePair<string, Date> GetData() => person;
    -references
    public KeyValuePair<string, Date> GetKVP() => person;
    -references
    public string GetName() => person.Key;
    -references
    public string GetDate() => person.Value.GetUKdate();
    -references
    public string GenerateString() => GetName() + ": " + GetDate();
    -references
    public PersonNode GetLeftNode() => left;
    -references
    public PersonNode GetRightNode() => right;
    -references
    public void SetRightNode(PersonNode node) => right = node;
    -references
    public void SetLeftNode(PersonNode node) => left = node;
    -references
    public void SetData(string name, Date birth) => new PersonNode(name, birth);
}
```

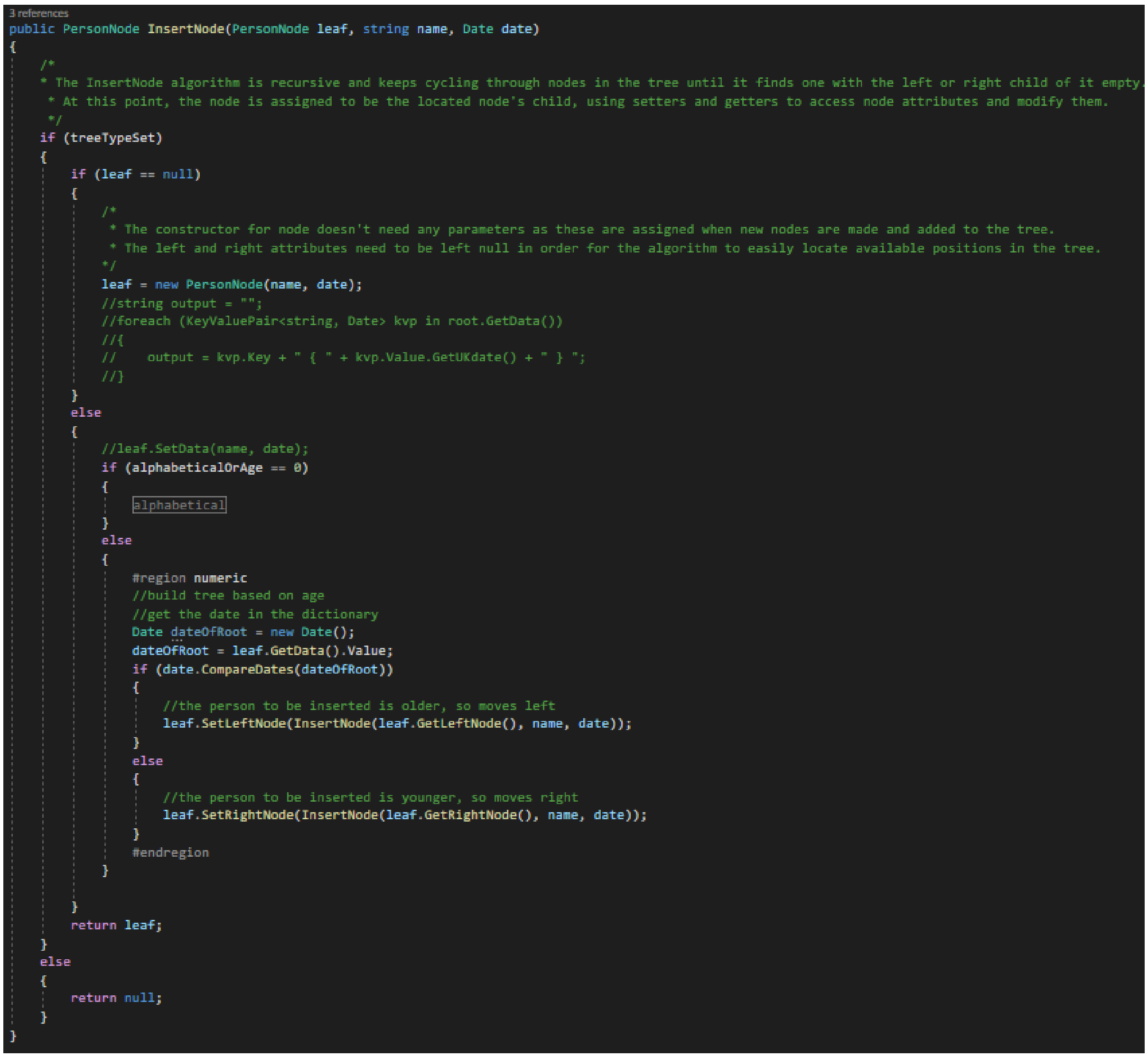
Above is the class for PersonNode.



CompareDates(date to compare)
bool earlier = false;
Is this date's year < date.GetYear()?
Yes
No
earlier = true;
Are the years equal?
No
return earlier
Yes
Yes
Is this date's month < date.GetMonth()?
Yes
No
Is this date's day < date.GetDay()?
Yes
No
Are the months equal?

Left is the CompareDates function.

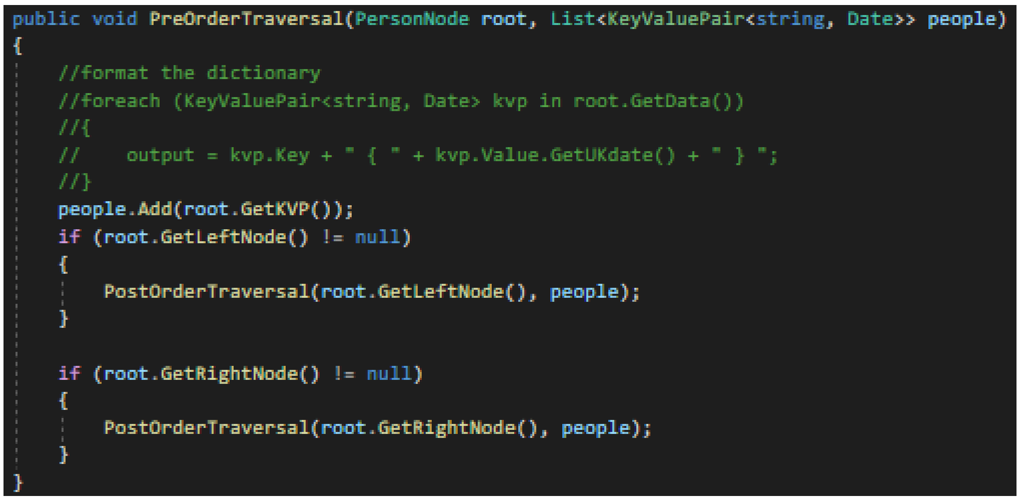
I made two different binary tree classes: one for organising numbers (Tree class) and one for organising people - DictionaryTree class, named so because it contains key-value pairs like a dictionary. It organises people based on their age, calling the CompareDates() function on new PersonNodes being inserted. The way the person insertion algorithm works is through recursion: a node acts as the root of the tree and the algorithm is called on the node to assign its left nodes and right nodes, each of which can have their own left nodes and right nodes. This essentially keeps going until a node has an empty slot for its left or right node, in which case the new person’s data is inserted at this point.



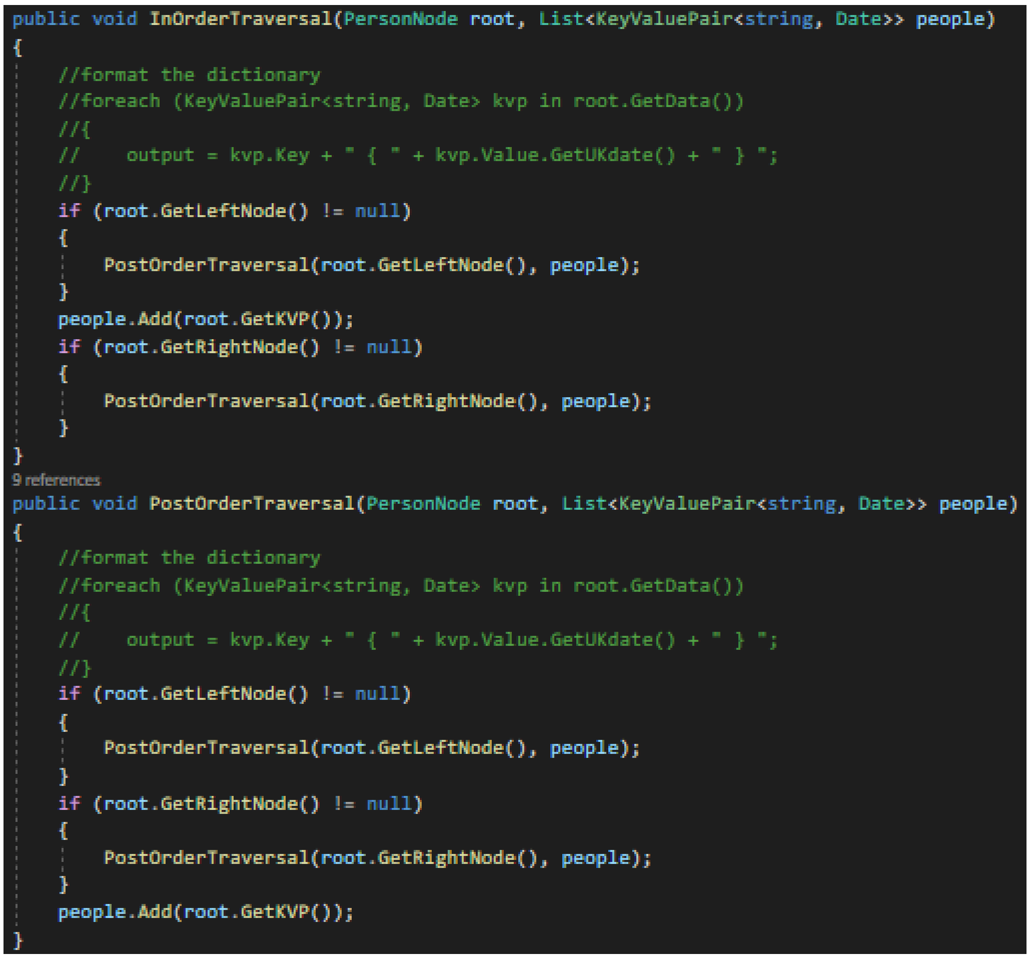
```
3 references
public PersonNode InsertNode(PersonNode leaf, string name, Date date)
{
    /*
    * The InsertNode algorithm is recursive and keeps cycling through nodes in the tree until it finds one with the left or right child of it empty.
     * At this point, the node is assigned to be the located node's child, using setters and getters to access node attributes and modify them.
     */
    if (treeTypeSet)
    {
        if (leaf == null)
        {
            /*
             * The constructor for node doesn't need any parameters as these are assigned when new nodes are made and added to the tree.
             * The left and right attributes need to be left null in order for the algorithm to easily locate available positions in the tree.
            */
            leaf = new PersonNode(name, date);
            //string output = "";
            //foreach (KeyValuePair<string, Date> kvp in root.GetData())
            //{
            //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";
            //}
        }
        else
        {
            //leaf.SetData(name, date);
            if (alphabeticalOrAge == 0)
            {
                alphabetical
            }
            else
            {
                #region numeric
                //build tree based on age
                //get the date in the dictionary
                Date dateOfRoot = new Date();
                dateOfRoot = leaf.GetData().Value;
                if (date.CompareDates(dateOfRoot))
                {
                    //the person to be inserted is older, so moves left
                    leaf.SetLeftNode(InsertNode(leaf.GetLeftNode(), name, date));
                }
                else
                {
                    //the person to be inserted is younger, so moves right
                    leaf.SetRightNode(InsertNode(leaf.GetRightNode(), name, date));
                }
                #endregion
            }

        }
        return leaf;
    }
    else
    {
        return null;
    }
}
```

This is the algorithm for inserting people into the DictionaryTree. It was going to have an option to build a tree alphabetically but in the final version, I made it only order people based on their age.

The three traversal types (pre order, in order and post order) are very similarly programmed. They all put the nodes from the tree into a list (specified in the functions’ input parameters) based on a node provided to the function for it to traverse. The functions then perform left and right traversals at different points before and after adding nodes to the list. The algorithms are largely inspired by the ones in the textbook.

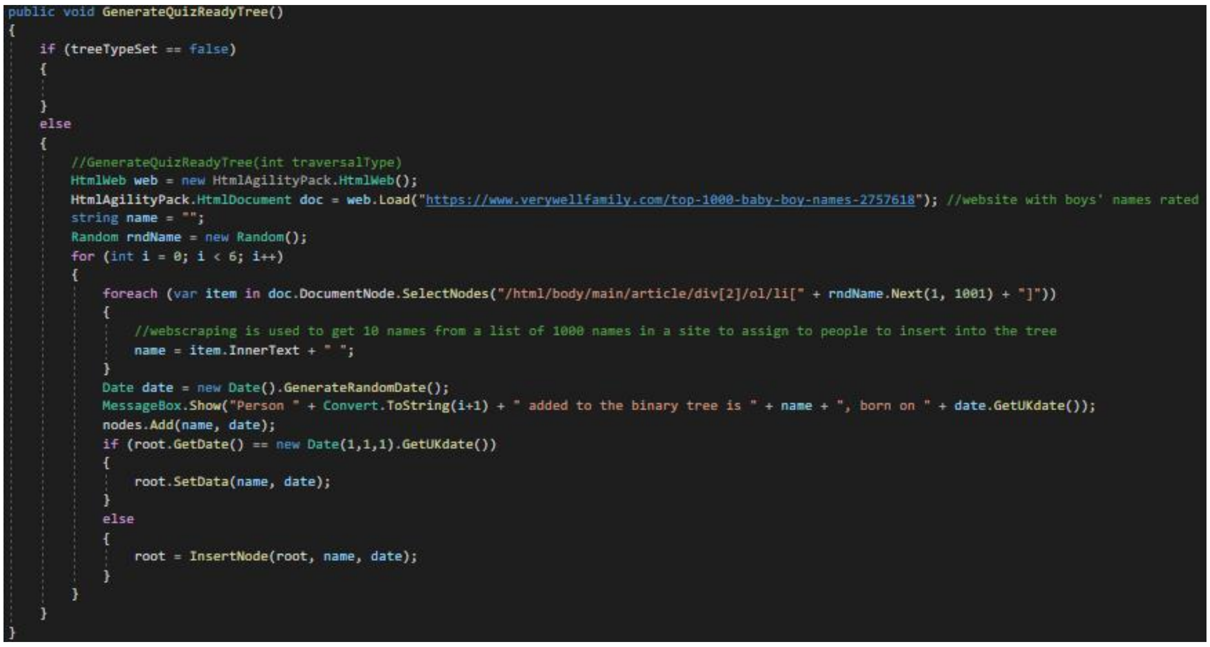


```
public void PreOrderTraversal(PersonNode root, List<KeyValuePair<string, Date>> people)
{
    //format the dictionary
    //foreach (KeyValuePair<string, Date> kvp in root.GetData())
    //{
    //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";
    //}
    people.Add(root.GetKVP());
    if (root.GetLeftNode() != null)
    {
        PostOrderTraversal(root.GetLeftNode(), people);
    }

    if (root.GetRightNode() != null)
    {
        PostOrderTraversal(root.GetRightNode(), people);
    }
}
```



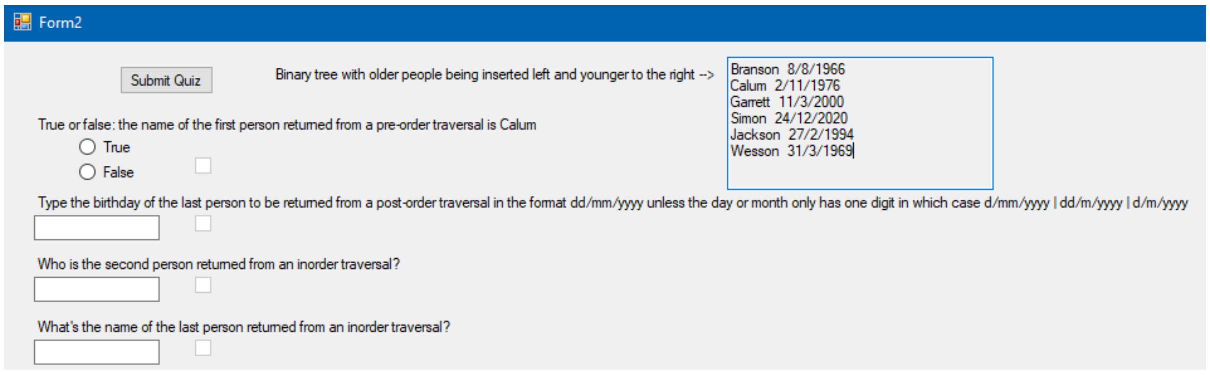
```
public void InOrderTraversal(PersonNode root, List<KeyValuePair<string, Date>> people)
{
    //format the dictionary
    //foreach (KeyValuePair<string, Date> kvp in root.GetData())
    //{
    //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";
    //}
    if (root.GetLeftNode() != null)
    {
        PostOrderTraversal(root.GetLeftNode(), people);
    }
    people.Add(root.GetKVP());
    if (root.GetRightNode() != null)
    {
        PostOrderTraversal(root.GetRightNode(), people);
    }
}
9 references
public void PostOrderTraversal(PersonNode root, List<KeyValuePair<string, Date>> people)
{
    //format the dictionary
    //foreach (KeyValuePair<string, Date> kvp in root.GetData())
    //{
    //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";
    //}
    if (root.GetLeftNode() != null)
    {
        PostOrderTraversal(root.GetLeftNode(), people);
    }
    if (root.GetRightNode() != null)
    {
        PostOrderTraversal(root.GetRightNode(), people);
    }
    people.Add(root.GetKVP());
}
```

When creating a quiz, the person tree is generated with random names and birthdays, on which questions are asked, e.g. what is the name of the third person returned from a pre order traversal; such questions develop fluency in dealing with tree structures and understanding them. In order to get random names, I used webscraping to access a website and gather random names from a set of 1000. My program does this by using the HtmlAgilityPack, which allows you to create a HtmlDocument object by using a link to a html file on the internet. Next, my code gets the Xpath of the list in the document which stores the names. It then generates random numbers and picks a random name from the list, repeating this until it has 6 names. In this same loop, a new date is created and has a random date set. For some reason, there were issues with the random function where it kept setting the same date for each subsequent date object, but using a message box object in the loop prevented the issue so I displayed the name being added to the tree as well as their birthday with each iteration of the loop, which in my opinion serves as a quality of life improvement as it allows the user to see what is happening to the tree they will be reading from.



```
public void GenerateQuizReadyTree()
{
    if (treeTypeSet == false)
    {

    }
    else
    {
        //GenerateQuizReadyTree(int traversalType)
        HtmlWeb web = new HtmlAgilityPack.HtmlWeb();
        HtmlAgilityPack.HtmlDocument doc = web.Load("https://www.verywellfamily.com/top-1000-baby-boy-names-2757618"); //website with boys' names rated
        string name = "";
        Random rndName = new Random();
        for (int i = 0; i < 6; i++)
        {
            foreach (var item in doc.DocumentNode.SelectNodes("/html/body/main/article/div[2]/ol/li[" + rndName.Next(1, 1001) + "]"))
            {
                //webscraping is used to get 10 names from a list of 1000 names in a site to assign to people to insert into the tree
                name = item.InnerText + " ";
            }
            Date date = new Date().GenerateRandomDate();
            MessageBox.Show("Person " + Convert.ToString(i+1) + " added to the binary tree is " + name + ", born on " + date.GetUKdate());
            nodes.Add(name, date);
            if (root.GetDate() == new Date(1,1,1).GetUKdate())
            {
                root.SetData(name, date);
            }
            else
            {
                root = InsertNode(root, name, date);
            }
        }
    }
}
```

The questions asked involve getting different types of data from the value in the tree, such as their name or their birthday, while the tree is sorted based on age. This means the user has to be wary and pay attention to the detail of the question, which serves as good practice for exams in any subject but especially computing and other sciences where accuracy is key in achieving the desired outcomes. Furthermore, if working in an IT field, you will need to have accuracy and clarity with what you write so that messages are conveyed effectively and for when sorting data and error-checking, so it helps with prep for potential careers of students using the revision tool. This aspect of it where you need to pick out the right data from the tree can also be used to keep existing IT employees sharp on analysis and observational skills.



Binary tree with older people being inserted left and younger to the right -->

```
Branson  8/8/1966
Calum  2/11/1976
Garrett  11/3/2000
Simon  24/12/2020
Jackson  27/2/1994
Wesson  31/3/1969
```

True or false: the name of the first person returned from a pre-order traversal is Calum

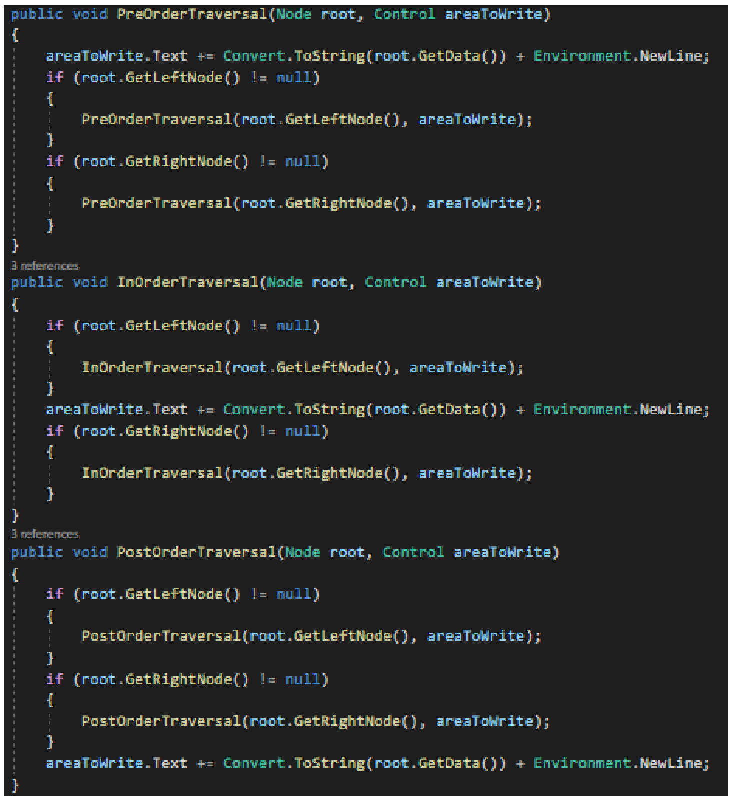
- ○ True
- ○ False

Type the birthday of the last person to be returned from a post-order traversal in the format dd/mm/yyyy unless the day or month only has one digit in which case d/mm/yyyy | dd/m/yyyy | d/m/yyyy

Who is the second person returned from an inorder traversal?

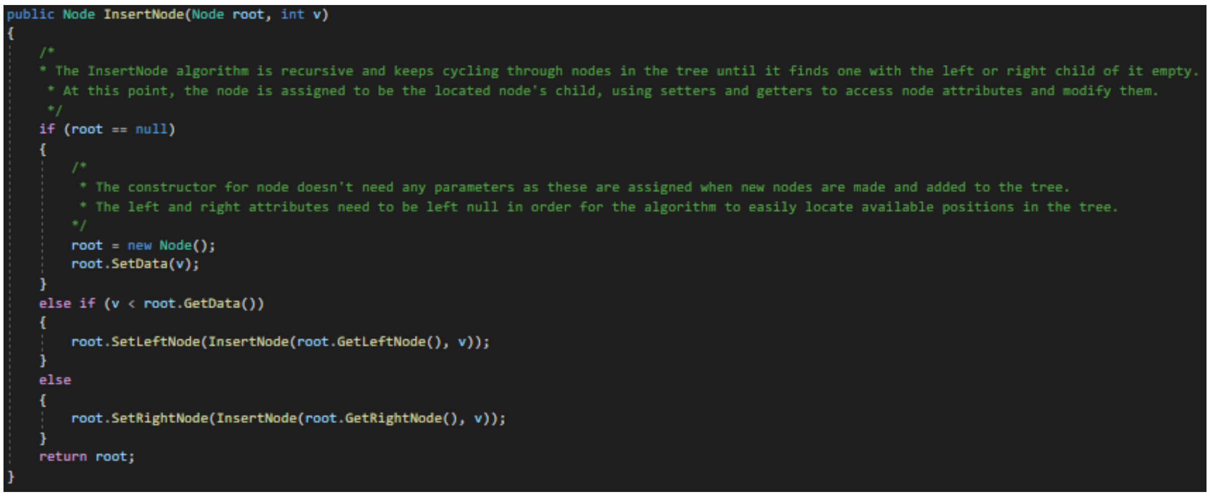
What's the name of the last person returned from an inorder traversal?

The numeric tree class is built quite similarly but with some differences. For one, there are two sets of the traversal functions: one returns a list of the nodes in the respective order for that traversal and the other directly writes the numbers to an output, such as a multiline textbox, provided in the functions' input parameters. This is used in the binary tree creator section of the program where you can insert integers into a tree and perform traversals on it to see how a binary tree operates.

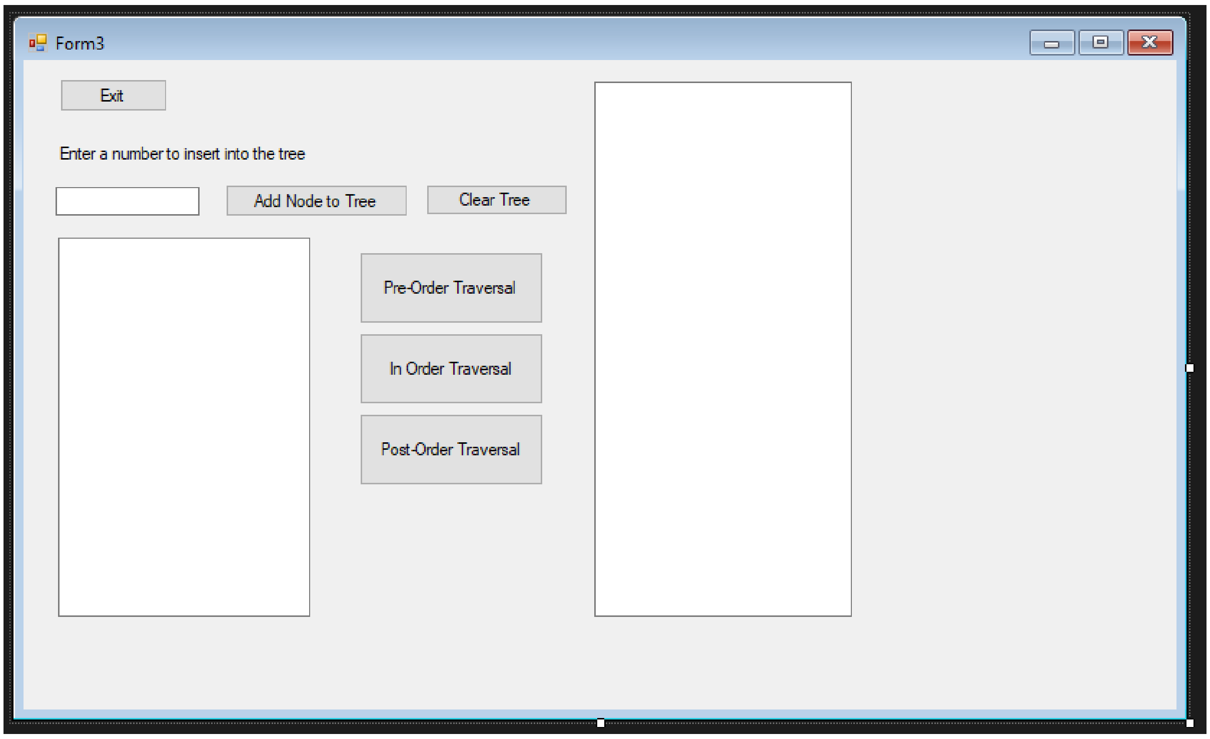


```
public void PreOrderTraversal(Node root, Control areaToWrite)
{
    areaToWrite.Text += Convert.ToString(root.GetData()) + Environment.NewLine;
    if (root.GetLeftNode() != null)
    {
        PreOrderTraversal(root.GetLeftNode(), areaToWrite);
    }
    if (root.GetRightNode() != null)
    {
        PreOrderTraversal(root.GetRightNode(), areaToWrite);
    }
}
3 references
public void InOrderTraversal(Node root, Control areaToWrite)
{
    if (root.GetLeftNode() != null)
    {
        InOrderTraversal(root.GetLeftNode(), areaToWrite);
    }
    areaToWrite.Text += Convert.ToString(root.GetData()) + Environment.NewLine;
    if (root.GetRightNode() != null)
    {
        InOrderTraversal(root.GetRightNode(), areaToWrite);
    }
}
3 references
public void PostOrderTraversal(Node root, Control areaToWrite)
{
    if (root.GetLeftNode() != null)
    {
        PostOrderTraversal(root.GetLeftNode(), areaToWrite);
    }
    if (root.GetRightNode() != null)
    {
        PostOrderTraversal(root.GetRightNode(), areaToWrite);
    }
    areaToWrite.Text += Convert.ToString(root.GetData()) + Environment.NewLine;
}
```

The insert node algorithm is far more straight forward than the person tree, as the data is simply an integer to compare, rather than a date which must be accessed through the object.

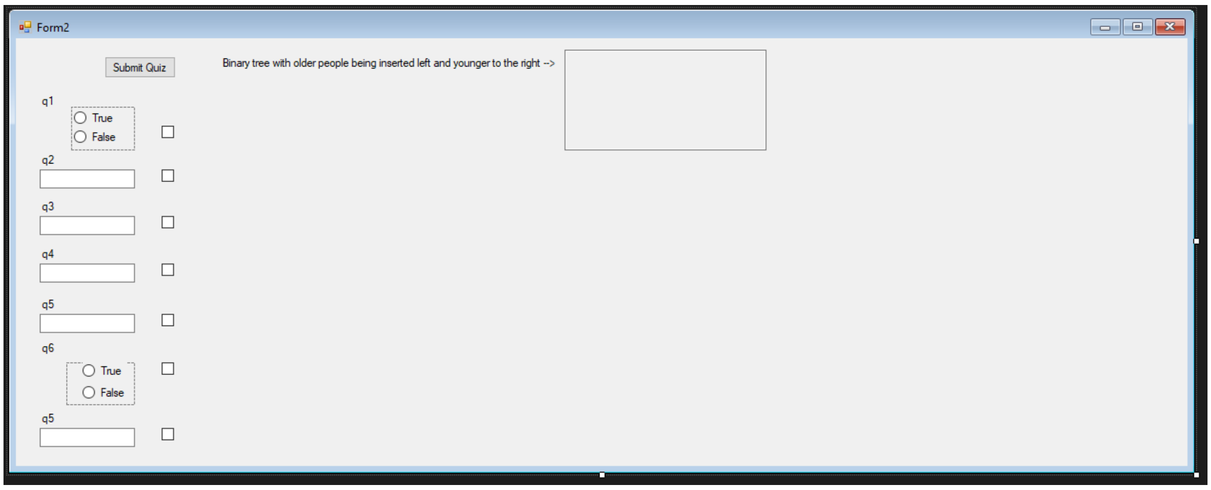


```
public Node InsertNode(Node root, int v)
{
    /*
    * The InsertNode algorithm is recursive and keeps cycling through nodes in the tree until it finds one with the left or right child of it empty.
     * At this point, the node is assigned to be the located node's child, using setters and getters to access node attributes and modify them.
     */
    if (root == null)
    {
        /*
         * The constructor for node doesn't need any parameters as these are assigned when new nodes are made and added to the tree.
         * The left and right attributes need to be left null in order for the algorithm to easily locate available positions in the tree.
        */
        root = new Node();
        root.SetData(v);
    }
    else if (v < root.GetData())
    {
        root.SetLeftNode(InsertNode(root.GetLeftNode(), v));
    }
    else
    {
        root.SetRightNode(InsertNode(root.GetRightNode(), v));
    }
    return root;
}
```

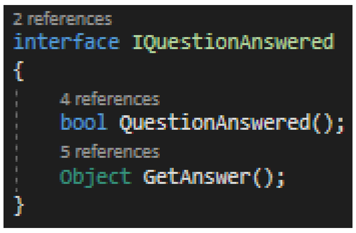


This is a screenshot of the binary tree creator; as aforementioned, it allows you to create integer-based trees. This is done by typing the number you want to add to the tree in the single line textbox and clicking the button to add it to the tree. The other buttons clear the tree and perform relevant traversals, whose results are shown in the multiline textbox to the right. The left multiline textbox shows the numbers you've inserted in order.

In the computer science quiz, there are a few different ways of answering questions, with a picture of the quiz form shown below:


Form2
Submit Quiz
Binary tree with older people being inserted left and younger to the right -->
q1
True
False
q2
q3
q4
q5
q6
True
False
q5

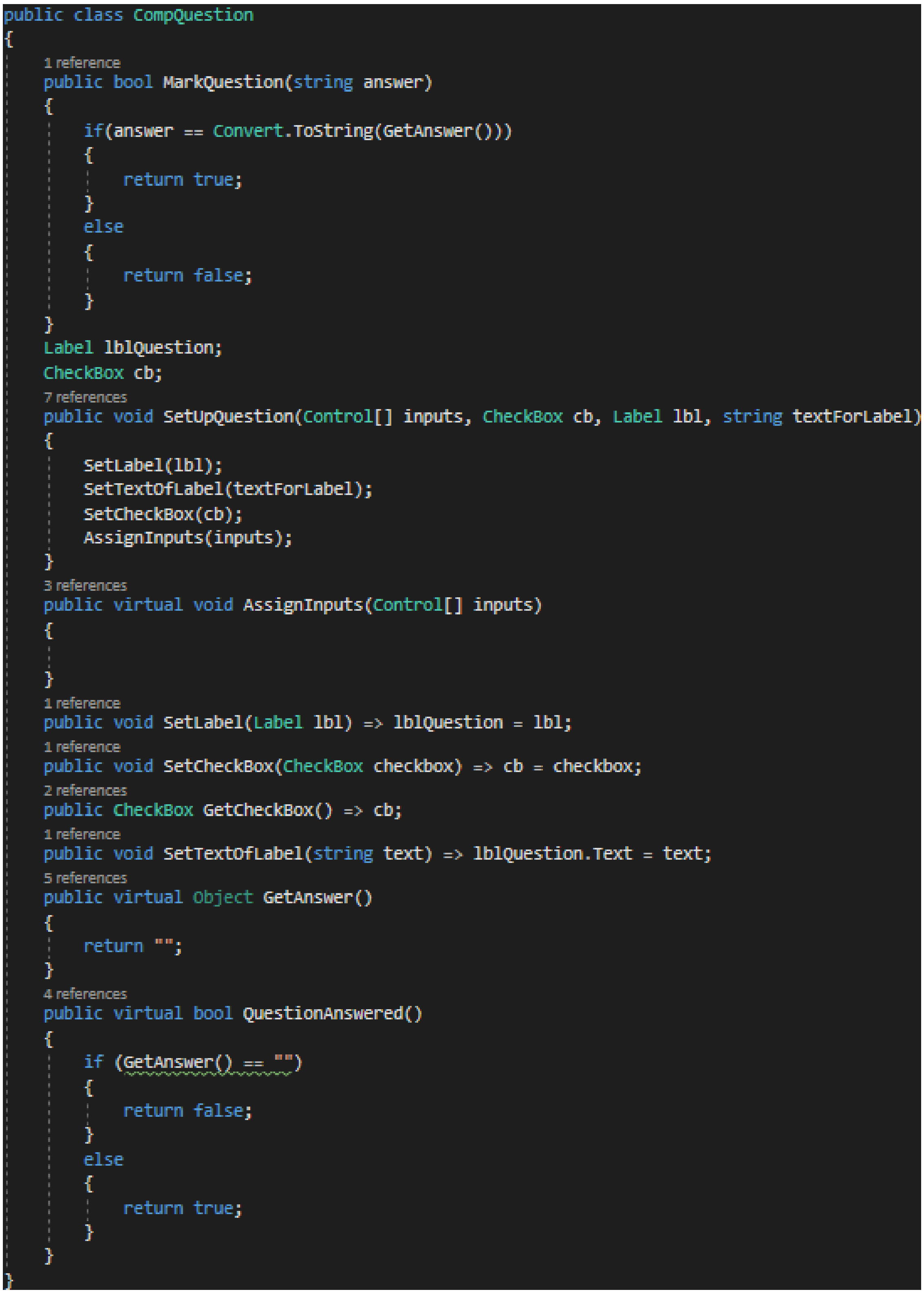
Due to this, writing the code for marking it was a bit tedious however by using an interface (effectively, a key word within the program which can have multiple definitions) it was easier to write as errors weren't thrown up for the wrong data types being returned from the functions QuestionAnswered() and GetAnswer().



```
2 references
interface IQuestionAnswered
{
    4 references
    bool QuestionAnswered();
    5 references
    Object GetAnswer();
}
```

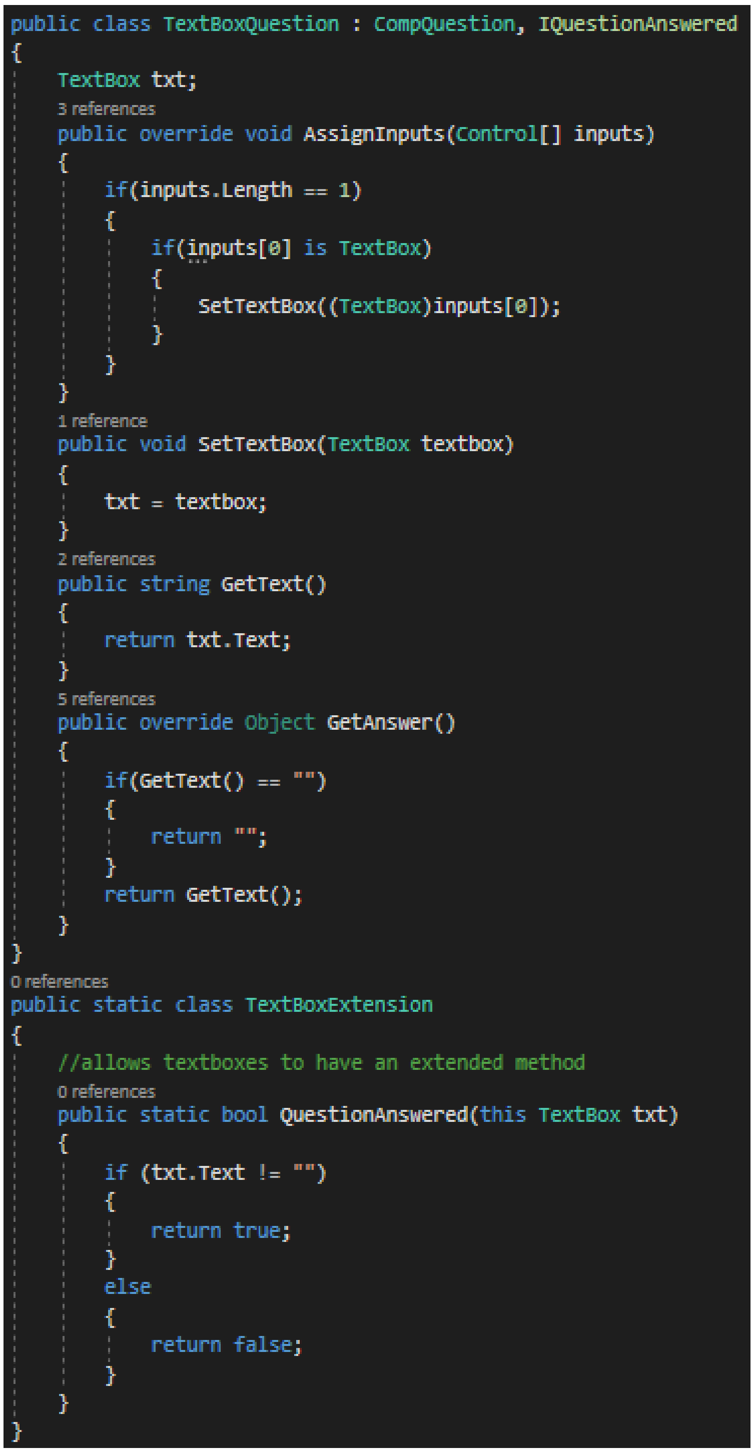
Having a standardised way of processing the different question types in the quiz makes it easier to code and debug. The different question types also had their own classes to encapsulate unique properties between them whilst all inheriting from a base parent class with their similarities.

CompQuestion is the base class with the ability to assign the relevant data input types (textboxes and radiobutton groups) with text and the check boxes for when marking the quiz.

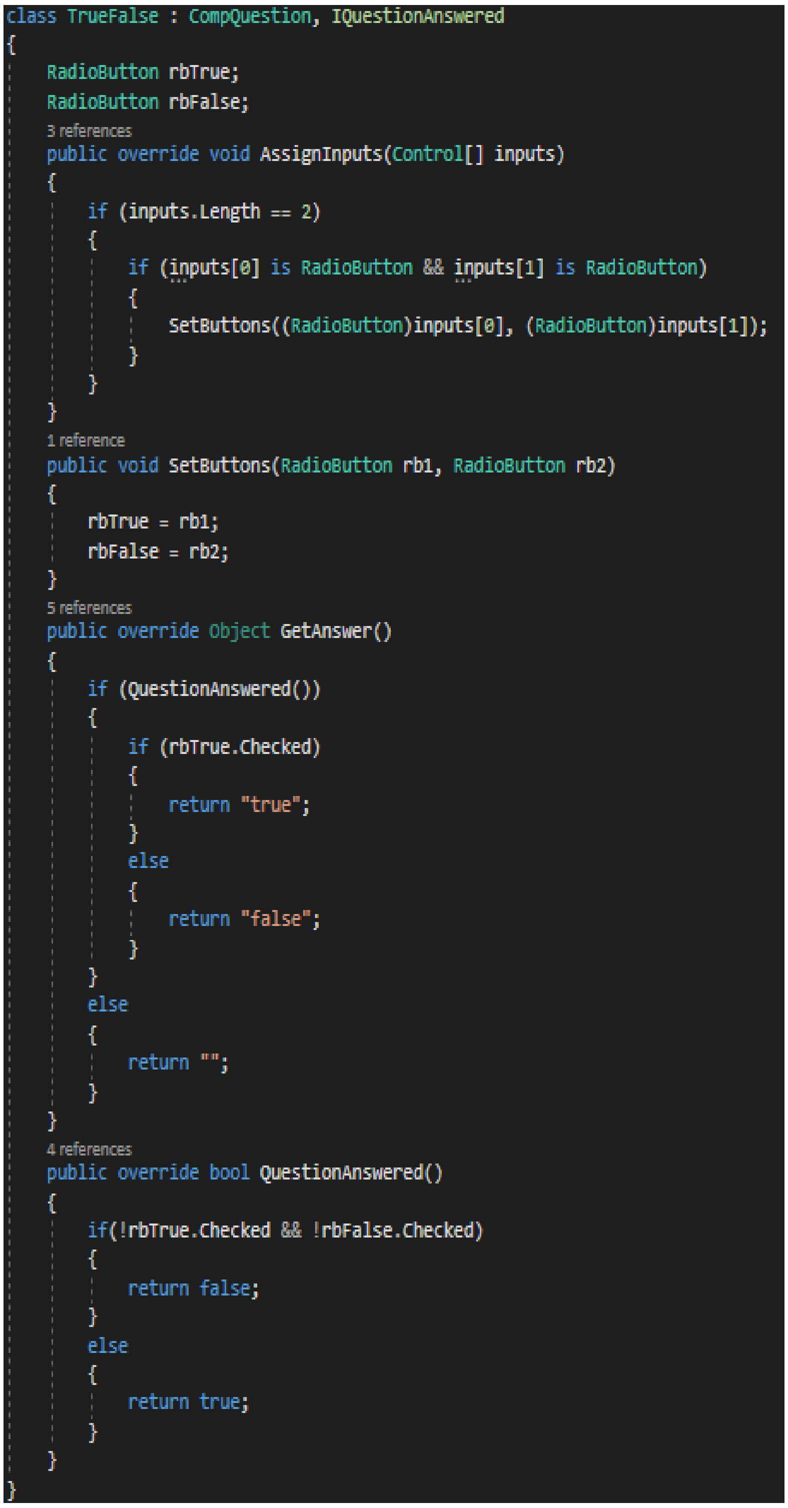


```
public class CompQuestion
{
    1 reference
    public bool MarkQuestion(string answer)
    {
        if(answer == Convert.ToString(GetAnswer()))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    Label lblQuestion;
    CheckBox cb;
    7 references
    public void SetUpQuestion(Control[] inputs, CheckBox cb, Label lbl, string textForLabel)
    {
        SetLabel(lbl);
        SetTextOfLabel(textForLabel);
        SetCheckBox(cb);
        AssignInputs(inputs);
    }
    3 references
    public virtual void AssignInputs(Control[] inputs)
    {

    }
    1 reference
    public void SetLabel(Label lbl) => lblQuestion = lbl;
    1 reference
    public void SetCheckBox(CheckBox checkbox) => cb = checkbox;
    2 references
    public CheckBox GetCheckBox() => cb;
    1 reference
    public void SetTextOfLabel(string text) => lblQuestion.Text = text;
    5 references
    public virtual Object GetAnswer()
    {
        return "";
    }
    4 references
    public virtual bool QuestionAnswered()
    {
        if (GetAnswer() == "")
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

One of its child classes is the TextBoxQuestion class, which inherits from CompQuestion and the interface mentioned above called IQuestionAnswered, which made coding the application smoother and error-free. As you can see, the AssignInputs() function in the parent class was empty whilst taking an array of form controls to assign to; the child classes make these specific to the question type, as seen below where textbox controls are set. I also used a static class for similar purposes as to why I used an interface: it keeps things standardised and cleaner to code. The static class allowed me to add methods to the textbox class like checking that a textbox has had text entered and thus is markable or not.



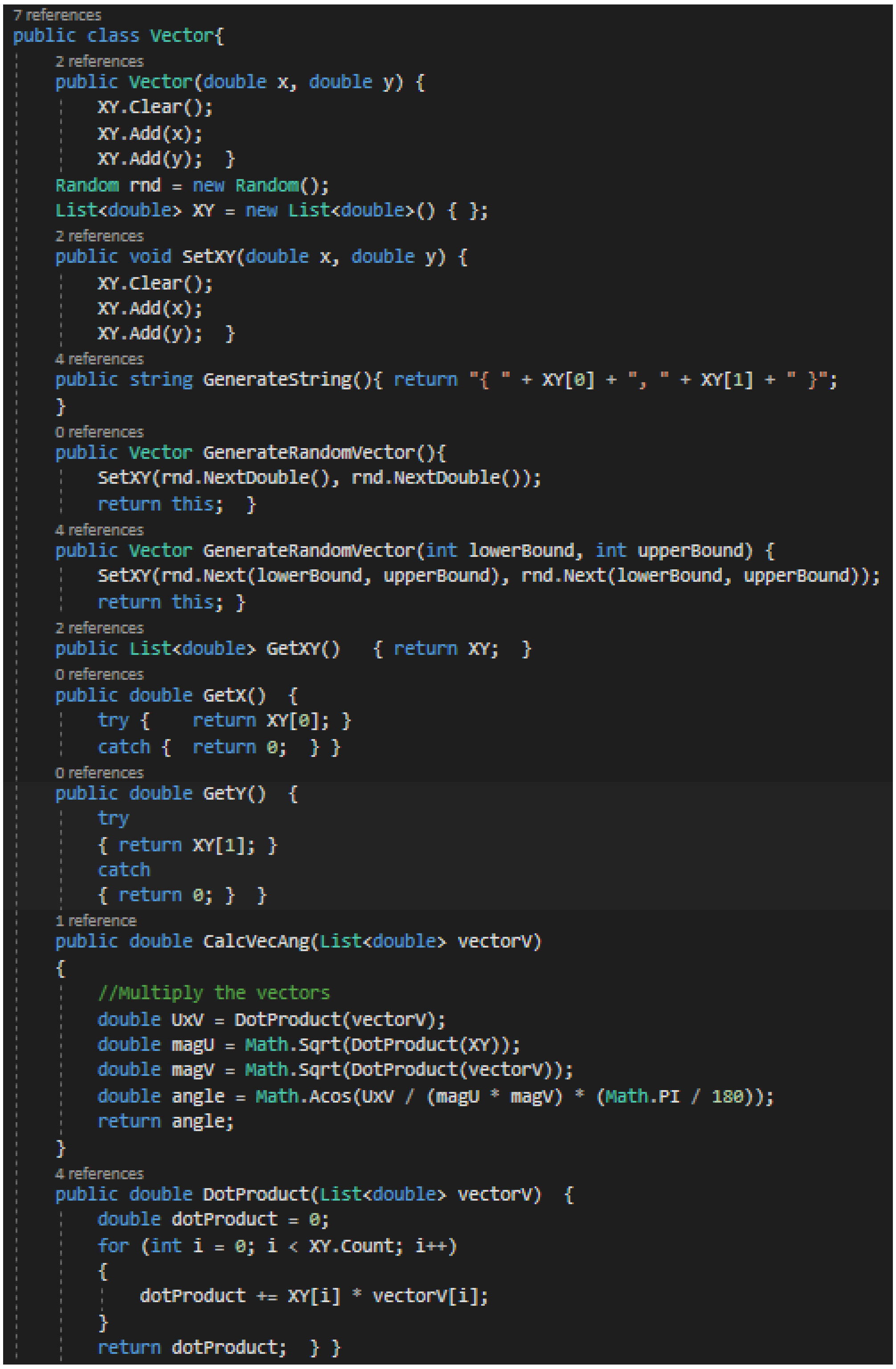
```
public class TextBoxQuestion : CompQuestion, IQuestionAnswered
{
    TextBox txt;
    3 references
    public override void AssignInputs(Control[] inputs)
    {
        if(inputs.Length == 1)
        {
            if(inputs[0] is TextBox)
            {
                SetTextBox((TextBox)inputs[0]);
            }
        }
    }
    1 reference
    public void SetTextBox(TextBox textbox)
    {
        txt = textbox;
    }
    2 references
    public string GetText()
    {
        return txt.Text;
    }
    5 references
    public override Object GetAnswer()
    {
        if(GetText() == "")
        {
            return "";
        }
        return GetText();
    }
}
0 references
public static class TextBoxExtension
{
    //allows textboxes to have an extended method
    0 references
    public static bool QuestionAnswered(this TextBox txt)
    {
        if (txt.Text != "")
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```



```
class TrueFalse : CompQuestion, IQuestionAnswered
{
    RadioButton rbTrue;
    RadioButton rbFalse;
    3 references
    public override void AssignInputs(Control[] inputs)
    {
        if (inputs.Length == 2)
        {
            if (inputs[0] is RadioButton && inputs[1] is RadioButton)
            {
                SetButtons((RadioButton)inputs[0], (RadioButton)inputs[1]);
            }
        }
    }
    1 reference
    public void SetButtons(RadioButton rb1, RadioButton rb2)
    {
        rbTrue = rb1;
        rbFalse = rb2;
    }
    5 references
    public override Object GetAnswer()
    {
        if (QuestionAnswered())
        {
            if (rbTrue.Checked)
            {
                return "true";
            }
            else
            {
                return "false";
            }
        }
        else
        {
            return "";
        }
    }
    4 references
    public override bool QuestionAnswered()
    {
        if(!rbTrue.Checked && !rbFalse.Checked)
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

Above is the other child class, TrueFalse; it utilises pairs of radiobuttons which were put inside panels so that they didn't interfere with eachother as when I tested my program, only one of the four radio buttons could be checked as they were being perceived as an entire group whereas it should be two pairs of buttons as there are two true/false questions.

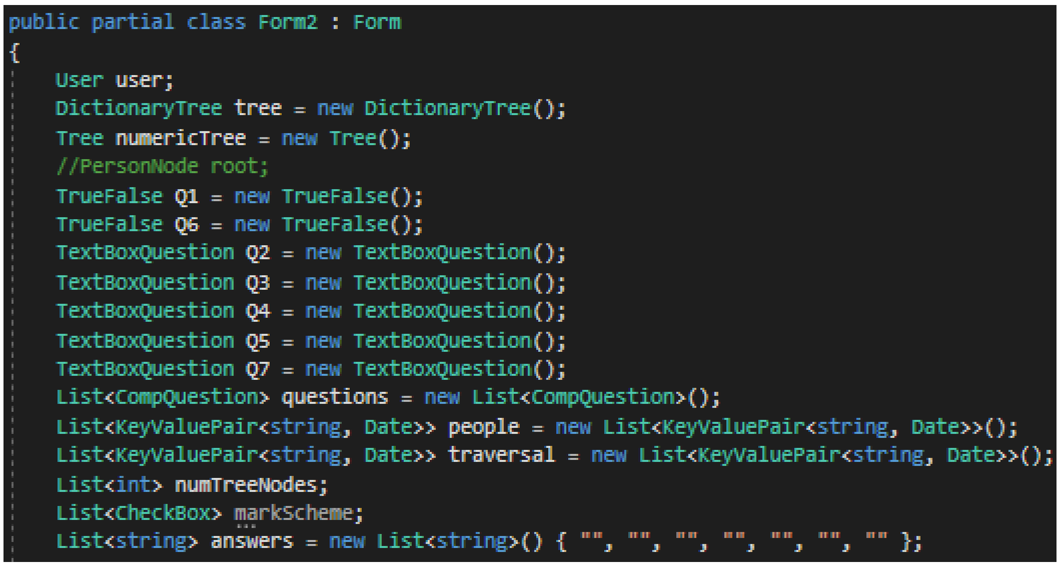
Another class used in this quiz page was one for vectors as there are two questions about vectors: one for finding the dot product and the other for finding the angle between two vectors.



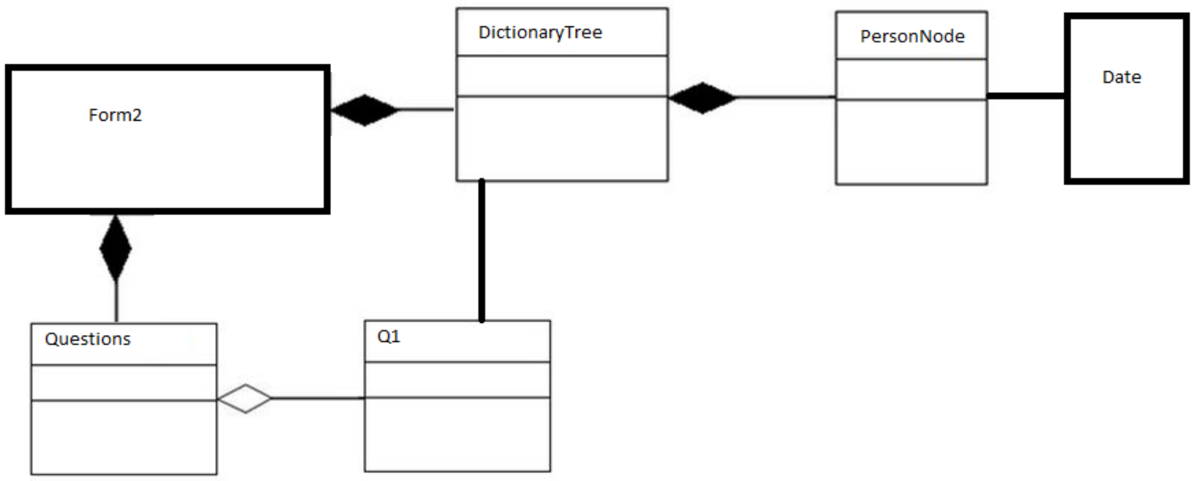
```
7 references
public class Vector{
    2 references
    public Vector(double x, double y) {
        XY.Clear();
        XY.Add(x);
        XY.Add(y);  }
    Random rnd = new Random();
    List<double> XY = new List<double>() { };
    2 references
    public void SetXY(double x, double y) {
        XY.Clear();
        XY.Add(x);
        XY.Add(y);  }
    4 references
    public string GenerateString(){ return "{ " + XY[0] + ", " + XY[1] + " }";
    }
    0 references
    public Vector GenerateRandomVector(){
        SetXY(rnd.NextDouble(), rnd.NextDouble());
        return this;  }
    4 references
    public Vector GenerateRandomVector(int lowerBound, int upperBound) {
        SetXY(rnd.Next(lowerBound, upperBound), rnd.Next(lowerBound, upperBound));
        return this; }
    2 references
    public List<double> GetXY()   { return XY;  }
    0 references
    public double GetX()  {
        try {    return XY[0]; }
        catch {  return 0;   } }
    0 references
    public double GetY()  {
        try
        { return XY[1]; }
        catch
        { return 0; }  }
    1 reference
    public double CalcVecAng(List<double> vectorV)
    {
        //Multiply the vectors
        double UxV = DotProduct(vectorV);
        double magU = Math.Sqrt(DotProduct(XY));
        double magV = Math.Sqrt(DotProduct(vectorV));
        double angle = Math.Acos(UxV / (magU * magV) * (Math.PI / 180));
        return angle;
    }
    4 references
    public double DotProduct(List<double> vectorV)  {
        double dotProduct = 0;
        for (int i = 0; i < XY.Count; i++)
        {
            dotProduct += XY[i] * vectorV[i];
        }
        return dotProduct;  } }
```

I learned about the dot product and how to find angles between vectors through using the A-Level Computer Science Textbook; vectors are effectively a 1x2, with the dot product being an application of matrix multiplication. Vectors and matrices are used to describe positions in 3D and 2D spaces, colours in images, the magnitude and direction of forces and so many other things so including a bit of practice into the quiz is useful for anyone with even the smallest interest in a science, especially computing and mathematics.

The quiz page for computing utilises several principles of object-oriented programming: association (objects have relationships to eachother, e.g. dates having relationships with person-nodes whereby one person-node can have one date attached while a single date could be associated with many person-nodes); aggregation (closer form of association, where one object contains another object, e.g. the list for the questions contains the individual question objects in the quiz, which in turn don't belong to any other collection, but are still independent of the list in that if the list is destroyed, the question objects won't be); and composition, the most notable principle applied. This is because the various classes such as TextBoxQuestion and TrueFalse are composed of form controls like textboxes and radiobuttons respectively as well as containing other objects, e.g. each TextBoxQuestion and TrueFalse is a child of CompQuestion and therefore inherit a label for the question to be written to and a checkbox for marking the question; ultimately, if the parent container of these objects, i.e. the question itself, is destroyed then the children to it won't exist. This is true for the person-nodes stored in the dictionary tree as well as they aren't stored anywhere other than inside the dictionary tree. Another thing to note is the entire form is composed of different objects too, with a special DictionaryTree for person nodes as aforementioned, a Tree for numbers, a User for the user, etc. If the form is deleted, the instances of these objects will also be deleted, thus demonstrating the strong ties of composition. Below is a screenshot of the global variables/objects used in the computing quiz form:

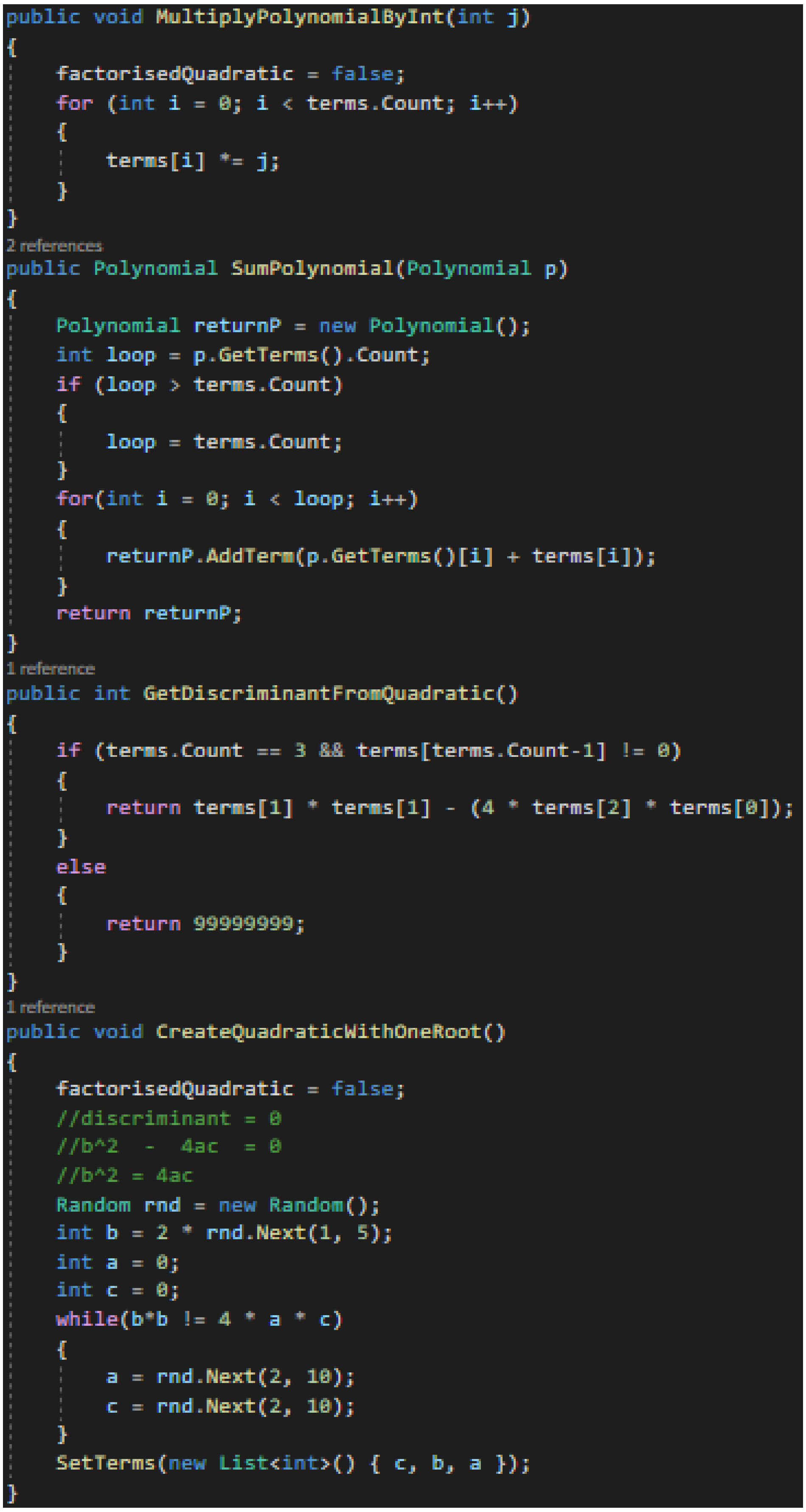


```
public partial class Form2 : Form
{
    User user;
    DictionaryTree tree = new DictionaryTree();
    Tree numericTree = new Tree();
    //PersonNode root;
    TrueFalse Q1 = new TrueFalse();
    TrueFalse Q6 = new TrueFalse();
    TextBoxQuestion Q2 = new TextBoxQuestion();
    TextBoxQuestion Q3 = new TextBoxQuestion();
    TextBoxQuestion Q4 = new TextBoxQuestion();
    TextBoxQuestion Q5 = new TextBoxQuestion();
    TextBoxQuestion Q7 = new TextBoxQuestion();
    List<CompQuestion> questions = new List<CompQuestion>();
    List<KeyValuePair<string, Date>> people = new List<KeyValuePair<string, Date>>();
    List<KeyValuePair<string, Date>> traversal = new List<KeyValuePair<string, Date>>();
    List<int> numTreeNodes;
    List<CheckBox> markScheme;
    List<string> answers = new List<string>() { "", "", "", "", "", "", "" };
```

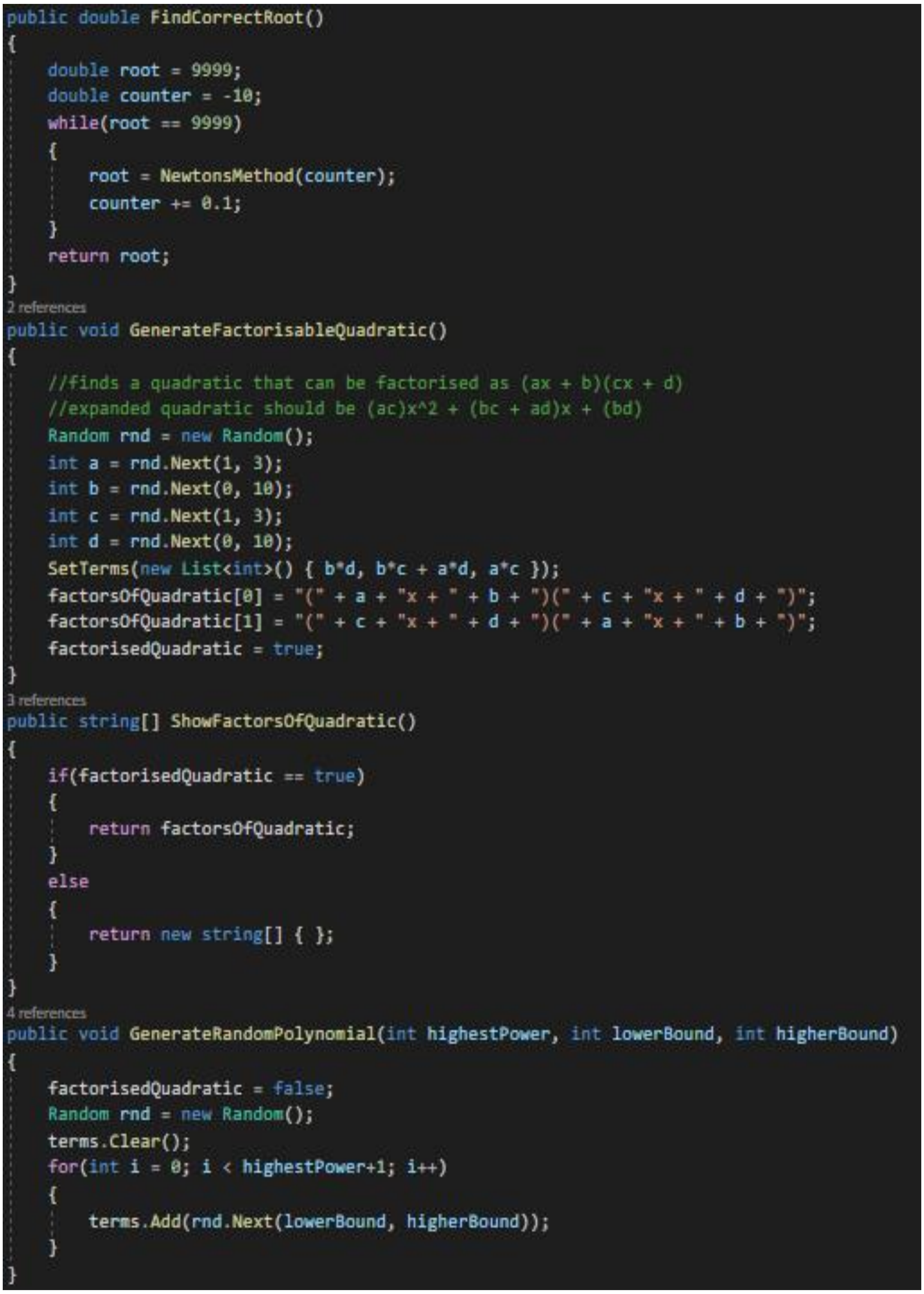


Above is a diagram showing some (not all) of the relationships between objects in the computing quiz page; a date is associated with a person-node, a person-node makes up the composition of a dictionary tree which is then associated with some of the questions, e.g. Q1 in the diagram; Q1 is aggregated with the list of questions. Both the list of questions and the dictionary tree are part of the composition for the overall quiz, Form2.

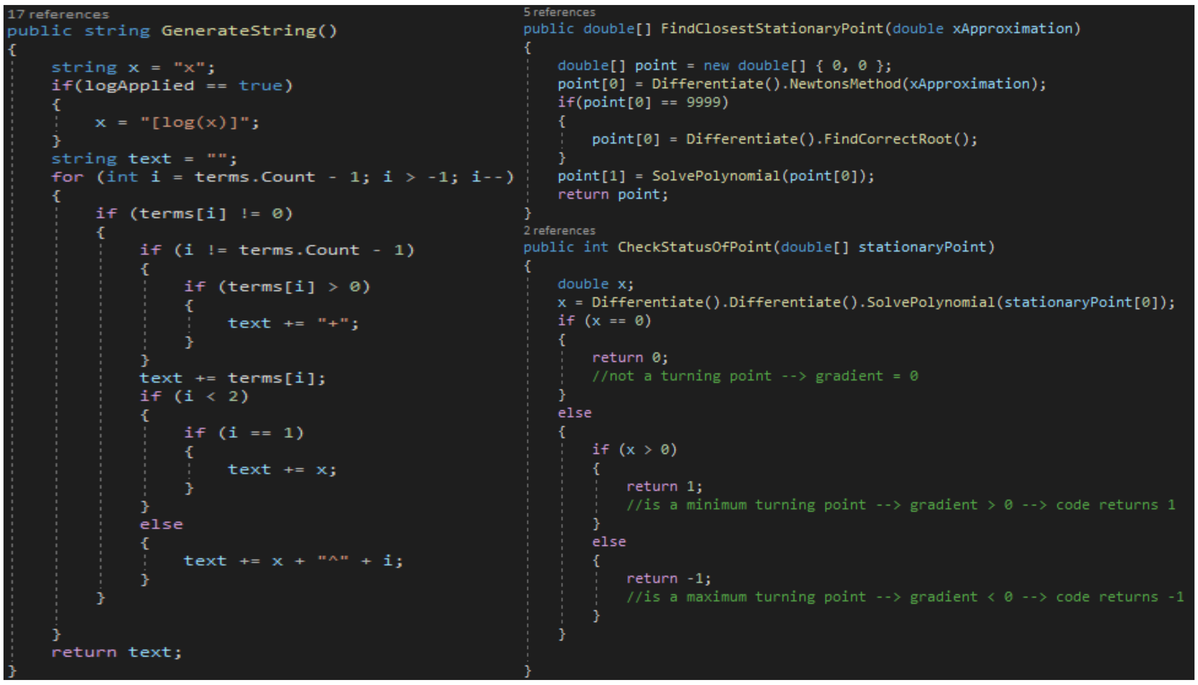
For the maths quizzes in my program, I developed a polynomial class which allows the page to dynamically create random polynomials and apply operations on them to create and mark questions, such as differentiating a cubic, finding the root of a quintic, adding polynomials together, multiplying polynomials by integers, etc. Below are screenshots of code for these various methods. Most of them are self-explanatory. The way my polynomials are handled is that they all contain a list of integers which represent coefficients of terms. The position of the coefficient in the list determines the power of x, e.g. the list { 10, 5, 1 } would represent [10 * x^0] + [5 * x^1] + [1 * x^2] -→ x^2 + 5x + 10.



```
public void MultiplyPolynomialByInt(int j)
{
    factorisedQuadratic = false;
    for (int i = 0; i < terms.Count; i++)
    {
        terms[i] *= j;
    }
}
2 references
public Polynomial SumPolynomial(Polynomial p)
{
    Polynomial returnP = new Polynomial();
    int loop = p.GetTerms().Count;
    if (loop > terms.Count)
    {
        loop = terms.Count;
    }
    for(int i = 0; i < loop; i++)
    {
        returnP.AddTerm(p.GetTerms()[i] + terms[i]);
    }
    return returnP;
}
1 reference
public int GetDiscriminantFromQuadratic()
{
    if (terms.Count == 3 && terms[terms.Count-1] != 0)
    {
        return terms[1] * terms[1] - (4 * terms[2] * terms[0]);
    }
    else
    {
        return 99999999;
    }
}
1 reference
public void CreateQuadraticWithOneRoot()
{
    factorisedQuadratic = false;
    //discriminant = 0
    //b^2  -  4ac  = 0
    //b^2 = 4ac
    Random rnd = new Random();
    int b = 2 * rnd.Next(1, 5);
    int a = 0;
    int c = 0;
    while(b*b != 4 * a * c)
    {
        a = rnd.Next(2, 10);
        c = rnd.Next(2, 10);
    }
    SetTerms(new List<int>() { c, b, a });
}
```

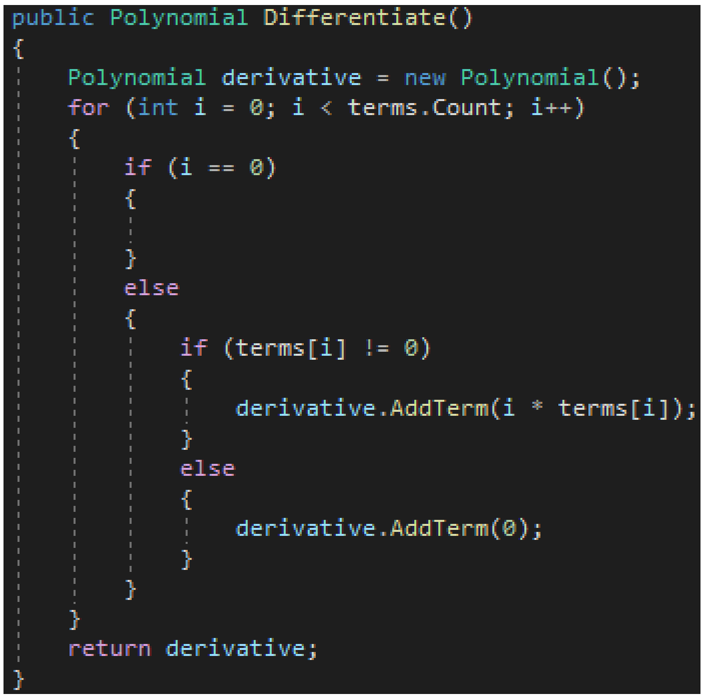


```
public double FindCorrectRoot()
{
    double root = 9999;
    double counter = -10;
    while(root == 9999)
    {
        root = NewtonsMethod(counter);
        counter += 0.1;
    }
    return root;
}
2 references
public void GenerateFactorisableQuadratic()
{
    //finds a quadratic that can be factorised as (ax + b)(cx + d)
    //expanded quadratic should be (ac)x^2 + (bc + ad)x + (bd)
    Random rnd = new Random();
    int a = rnd.Next(1, 3);
    int b = rnd.Next(0, 10);
    int c = rnd.Next(1, 3);
    int d = rnd.Next(0, 10);
    SetTerms(new List<int>() { b*d, b*c + a*d, a*c });
    factorsOfQuadratic[0] = "(" + a + "x + " + b + ")(" + c + "x + " + d + ")";
    factorsOfQuadratic[1] = "(" + c + "x + " + d + ")(" + a + "x + " + b + ")";
    factorisedQuadratic = true;
}
3 references
public string[] ShowFactorsOfQuadratic()
{
    if(factorisedQuadratic == true)
    {
        return factorsOfQuadratic;
    }
    else
    {
        return new string[] { };
    }
}
4 references
public void GenerateRandomPolynomial(int highestPower, int lowerBound, int higherBound)
{
    factorisedQuadratic = false;
    Random rnd = new Random();
    terms.Clear();
    for(int i = 0; i < highestPower+1; i++)
    {
        terms.Add(rnd.Next(lowerBound, higherBound));
    }
}
```



```
17 references
public string GenerateString()
{
    string x = "x";
    if(logApplied == true)
    {
        x = "[log(x)]";
    }
    string text = "";
    for (int i = terms.Count - 1; i > -1; i--)
    {
        if (terms[i] != 0)
        {
            if (i != terms.Count - 1)
            {
                if (terms[i] > 0)
                {
                    text += "+";
                }
            }
            text += terms[i];
            if (i < 2)
            {
                if (i == 1)
                {
                    text += x;
                }
            }
            else
            {
                text += x + "^" + i;
            }
        }

    }
    return text;
}
```

```
5 references
public double[] FindClosestStationaryPoint(double xApproximation)
{
    double[] point = new double[] { 0, 0 };
    point[0] = Differentiate().NewtonsMethod(xApproximation);
    if(point[0] == 9999)
    {
        point[0] = Differentiate().FindCorrectRoot();
    }
    point[1] = SolvePolynomial(point[0]);
    return point;
}
2 references
public int CheckStatusOfPoint(double[] stationaryPoint)
{
    double x;
    x = Differentiate().Differentiate().SolvePolynomial(stationaryPoint[0]);
    if (x == 0)
    {
        return 0;
        //not a turning point --> gradient = 0
    }
    else
    {
        if (x > 0)
        {
            return 1;
            //is a minimum turning point --> gradient > 0 --> code returns 1
        }
        else
        {
            return -1;
            //is a maximum turning point --> gradient < 0 --> code returns -1
        }
    }

}
```



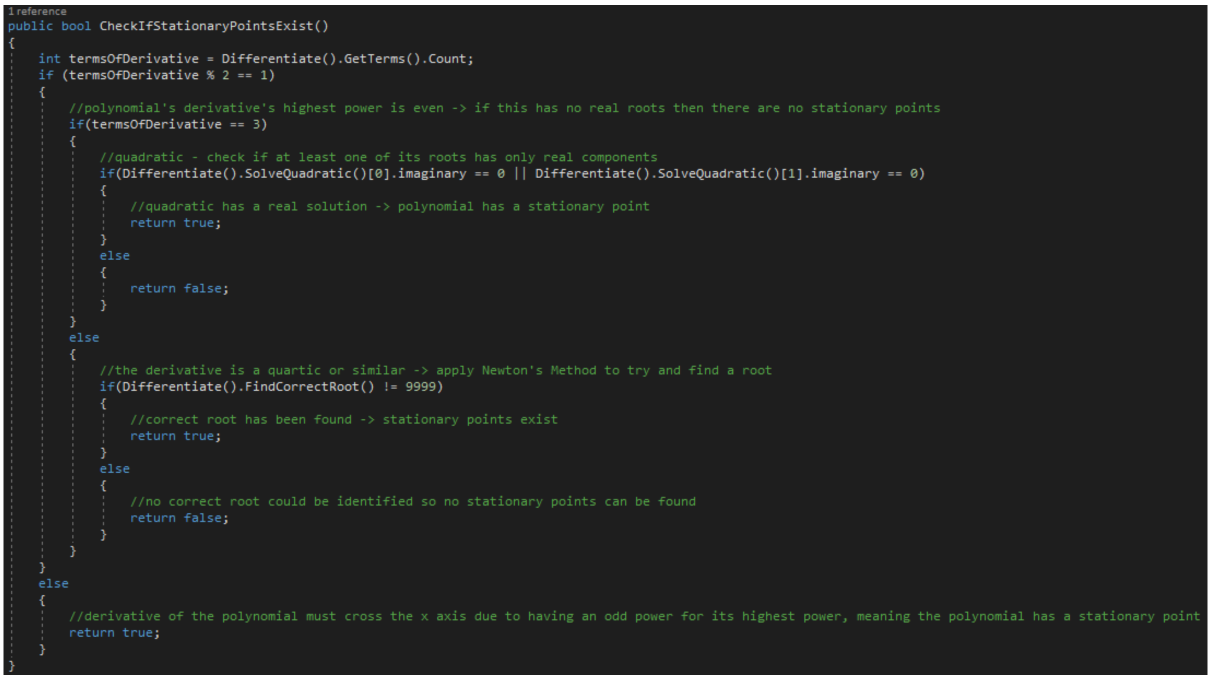
```
public Polynomial Differentiate()
{
    Polynomial derivative = new Polynomial();
    for (int i = 0; i < terms.Count; i++)
    {
        if (i == 0)
        {

        }
        else
        {
            if (terms[i] != 0)
            {
                derivative.AddTerm(i * terms[i]);
            }
            else
            {
                derivative.AddTerm(0);
            }
        }
    }
    return derivative;
}
```

Differentiating polynomials follows the formula where by the derivative of (ax^n) = nax^n-1; The function returns a polynomial as an object with the terms calculated by looping through the terms of the existing polynomial and applying this basic formula to each one. Differentiation is a very useful tool in maths as it can be used to find the gradient of a line/equation at a given point, which can in turn be used to find points in an equation/line which are stationary or turn; when differentiating these again, you can figure out if a point is a maximum point (peak of a hill) or a minimum point.

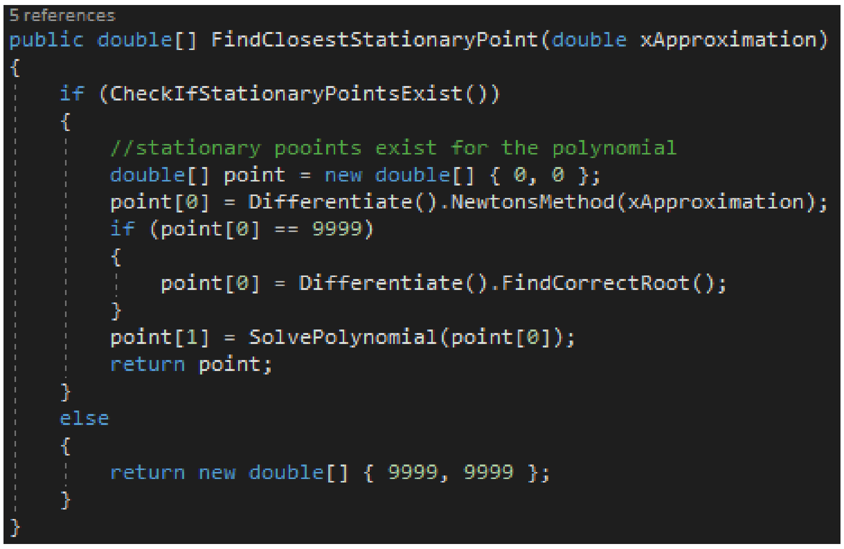
Above is an algorithm for this, CheckStatusOfPoint(), which finds the second derivative of the given polynomial and determines whether it is maximum, minimum or flat based on whether it is < 0, > 0 or = 0 respectively. Questions in the maths quizzes have then been asked on this, with this function applied to the polynomials in the questions to mark the quiz.

Another function shown is FindClosestStationaryPoint(), which, as its name suggests, finds one of the closest stationary points to a given x-coordinate. It does this by applying NewtonsMethod() (explained below) to the derivative of the polynomial, using the given x-coordinate. If no valid solution is found, it repeats; if a valid solution is found then it calculates the y-coordinate of the stationary point by using the valid x-coordinate found for it and substituting it into the polynomial to find what the y-coordinate at that point is, using the method SolvePolynomial() to do so.

CORRECTION: When testing the polynomial quiz, I found that when generating questions with random polynomials, they sometimes asked to find stationary points when none existed, e.g. for the cubic which would render the question. To fix this, I wrote an algorithm to check that stationary points exist first and then ran this at the start of FindClosestStationaryPoint(), which means when generating the polynomials, it checks that they are valid for the questions being asked and if not, generates new ones. The algorithms for finding the closest stationary points and checking them are shown below:



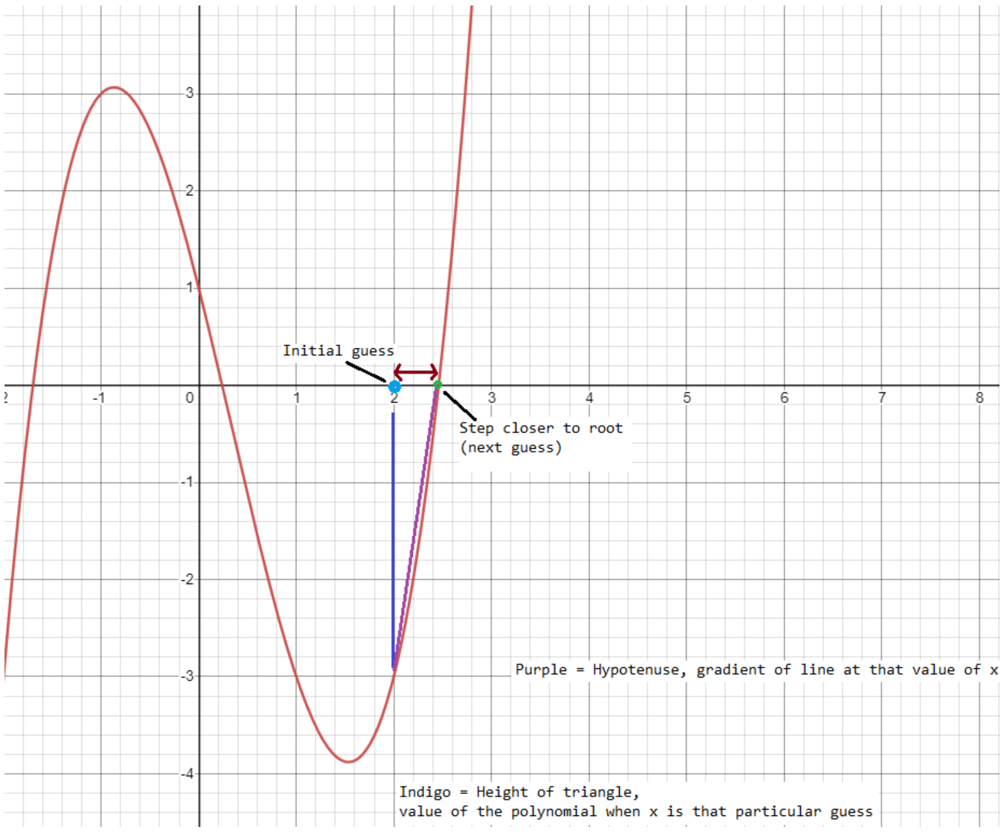
```
1 reference
public bool CheckIfStationaryPointsExist()
{
    int termsOfDerivative = Differentiate().GetTerms().Count;
    if (termsOfDerivative % 2 == 1)
    {
        //polynomial's derivative's highest power is even -> if this has no real roots then there are no stationary points
        if(termsOfDerivative == 3)
        {
            //quadratic - check if at least one of its roots has only real components
            if(Differentiate().SolveQuadratic()[0].imaginary == 0 || Differentiate().SolveQuadratic()[1].imaginary == 0)
            {
                //quadratic has a real solution -> polynomial has a stationary point
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            //the derivative is a quartic or similar -> apply Newton's Method to try and find a root
            if(Differentiate().FindCorrectRoot() != 9999)
            {
                //correct root has been found -> stationary points exist
                return true;
            }
            else
            {
                //no correct root could be identified so no stationary points can be found
                return false;
            }
        }
    }
    else
    {
        //derivative of the polynomial must cross the x axis due to having an odd power for its highest power, meaning the polynomial has a stationary point
        return true;
    }
}
```



```
5 references
public double[] FindClosestStationaryPoint(double xApproximation)
{
    if (CheckIfStationaryPointsExist())
    {
        //stationary pooints exist for the polynomial
        double[] point = new double[] { 0, 0 };
        point[0] = Differentiate().NewtonsMethod(xApproximation);
        if (point[0] == 9999)
        {
            point[0] = Differentiate().FindCorrectRoot();
        }
        point[1] = SolvePolynomial(point[0]);
        return point;
    }
    else
    {
        return new double[] { 9999, 9999 };
    }
}
```

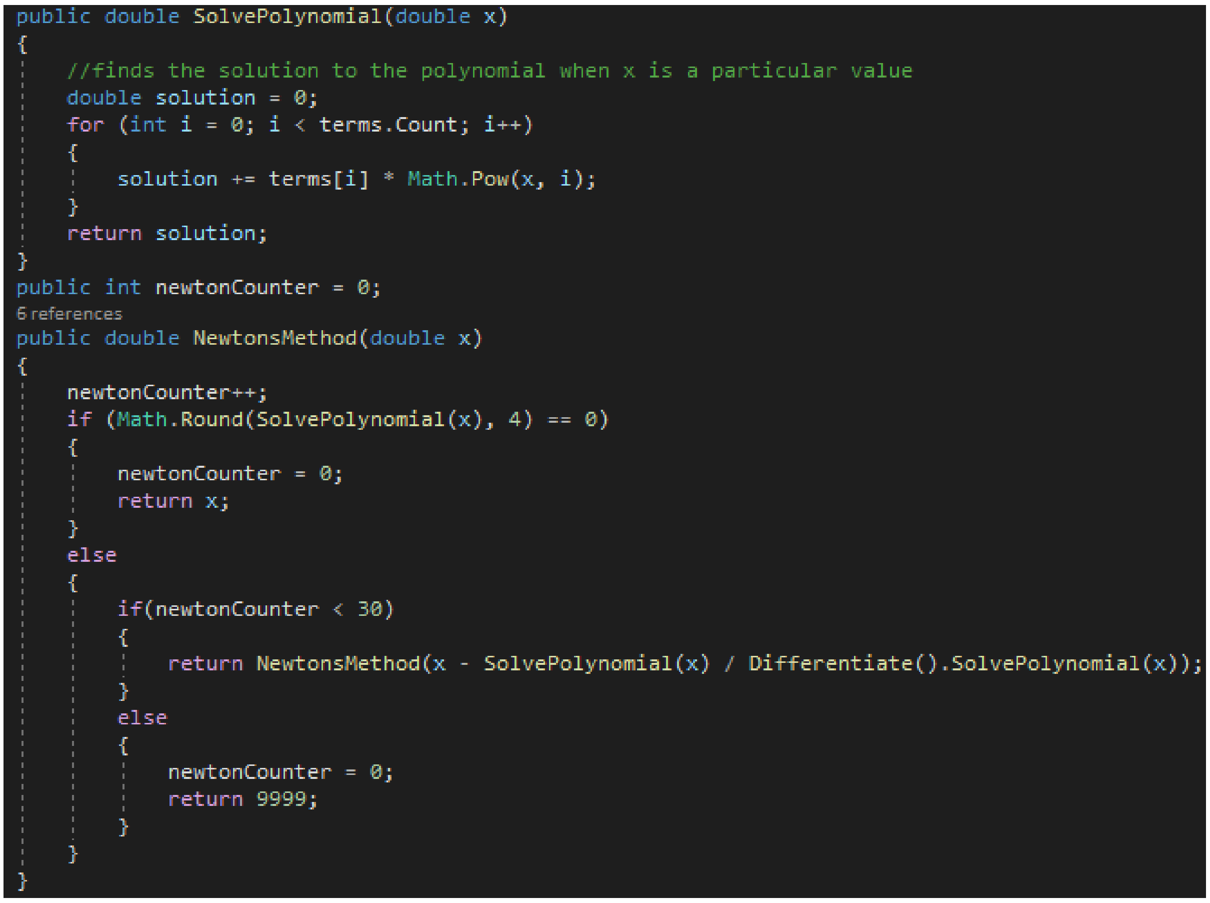
Differentiation was also particularly useful when incorporating NewtonsMethod() for finding roots to cubics, quartics, quintics and above. Basically, polynomials up to the power of 4 have formulas for calculating the roots, however only the quadratic formula is easy and reasonable to be able to memorise and do quickly, thus iterative numerical methods can be used to find incredibly close approximations for roots of polynomials with particularly high powers, especially quintics (x^5) and above which have no formulas discovered for solving them.

The Newton-Raphson Method is a key formula whereby you pick a random point along the x-axis for the graph of the polynomial, as a guess for the root. You then optimise this root by moving a small step closer to the actual root by finding the gradient of the line at your current value for x, then applying triangle theory to find the step. The diagram below, made using Desmos and MS Paint, demonstrates how this works, i.e. make a triangle whereby height is the value of the line at x = guess, hypotenuse is the gradient of the line at x = guess and base of the triangle is the distance between the height and hypotenuse. You can then draw a neat equation which tells you the next approximation for x, where $x_{n+1} = x_n - P(x) / P'(x)$ where P(x) is found by substituting your guess of x into the polynomial and P'(x) is where you substitute your value of x into the polynomial's derivative, which gets the gradient at this point on the line. Ultimately, you iterate this process until you reach a suitable level of accuracy for your root, i.e. where 5 significant figures remain consistent with each iteration – it is up to you how many iterations you continue for.



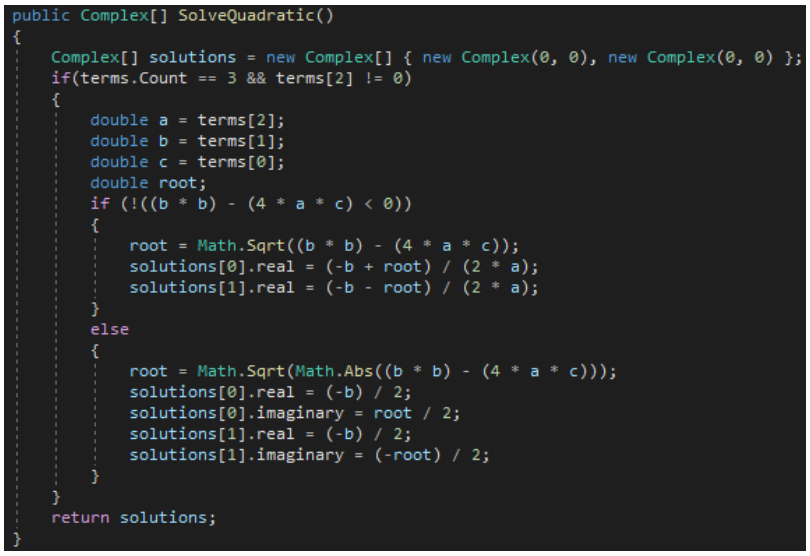
Initial guess
Step closer to root (next guess)
Purple = Hypotenuse, gradient of line at that value of x
Indigo = Height of triangle, value of the polynomial when x is that particular guess

To incorporate this, we need to be able to sub values into polynomials to generate numbers and differentiate polynomials. Below shows the code for solving polynomials with a set value and the function NewtonsMethod(); I decided to make it a recursive function as it is iterative and uses a previous output for the next input, eventually stopping when it meets a set criteria or 'base case', i.e. the solution to the polynomial with the current root when rounded to 4 decimal places is equal to 0 – when you input the root into the polynomial, it should be 0 so when rounded to some reasonable number of decimal places, such as 4 or 5, it is going to be somewhat accurate enough. If after 30 iterations or so, it hasn't found an accurate solution, then it returns 9999, which different parts of my program will interpret in their own relevant ways; for example, the Newton-Raphson method doesn't always converge to a root when the first initial guess is too far away from the root or can be caused to move around constantly due to being close to multiple roots, which prevents it converging to either. In this situation, I made the code handle it by testing different ranges of numbers for roots in a while loop (explained further on).

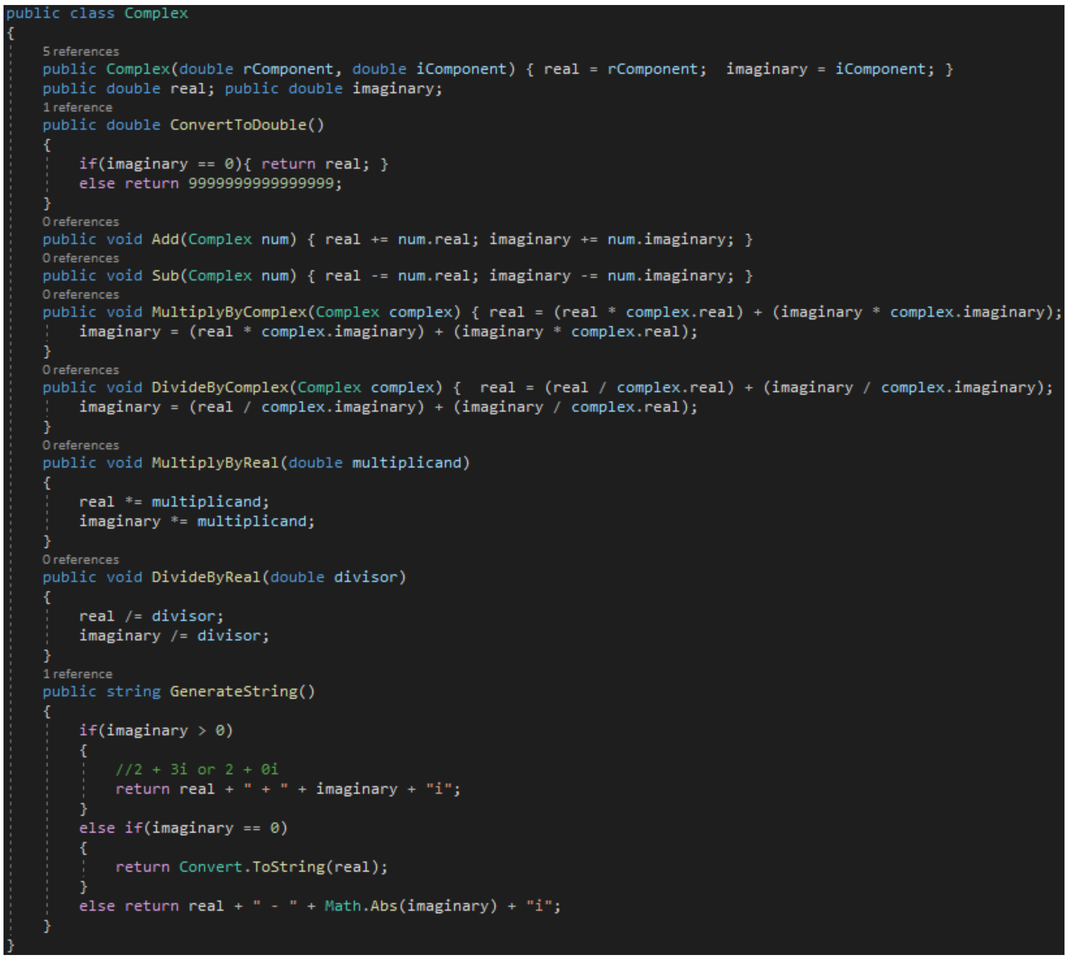


```
public double SolvePolynomial(double x)
{
    //finds the solution to the polynomial when x is a particular value
    double solution = 0;
    for (int i = 0; i < terms.Count; i++)
    {
        solution += terms[i] * Math.Pow(x, i);
    }
    return solution;
}
public int newtonCounter = 0;
6 references
public double NewtonsMethod(double x)
{
    newtonCounter++;
    if (Math.Round(SolvePolynomial(x), 4) == 0)
    {
        newtonCounter = 0;
        return x;
    }
    else
    {
        if(newtonCounter < 30)
        {
            return NewtonsMethod(x - SolvePolynomial(x) / Differentiate().SolvePolynomial(x));
        }
        else
        {
            newtonCounter = 0;
            return 9999;
        }
    }
}
```

An issue I had with NewtonsMethod() was that the recursion could lead to stackframe overflows often, which served as greater justification to only do 30 iterations of the algorithm before returning a base case value as such for if no solution can be found in those iterations – 9999, as mentioned above. Then, where the code has been implemented to find the root of a polynomial, like in the quiz for marking and in the polynomial solver section of my program, I have tested multiple ranges of values. In my polynomial solver for example, NewtonsMethod() serves as a component within the overall algorithm for solving polynomials, FindRootsOfPolynomial(); this is because quadratics are quicker and easier to solve using the quadratic formula and complex roots which contain an imaginary component can be found as well, unlike with the Newton-Raphson method. As a result, I developed a class to deal with complex numbers, shown below; it's a fairly straightforward class where each Complex has a real component and an imaginary component and can have basic operations like +,-,*,/ applied to it. A string can also be generated in the form, for example, "8 + 2i". How the FindRootsOfPolynomial() function works is it checks the number of terms in the polynomial to determine if it is a quadratic and thus to use the quadratic formula or if it has more terms and requires NewtonsMethod(). If it requires the quadratic formula, it applies the relevant method, which returns complex values even if it only has real roots, for uniform standardised processing.

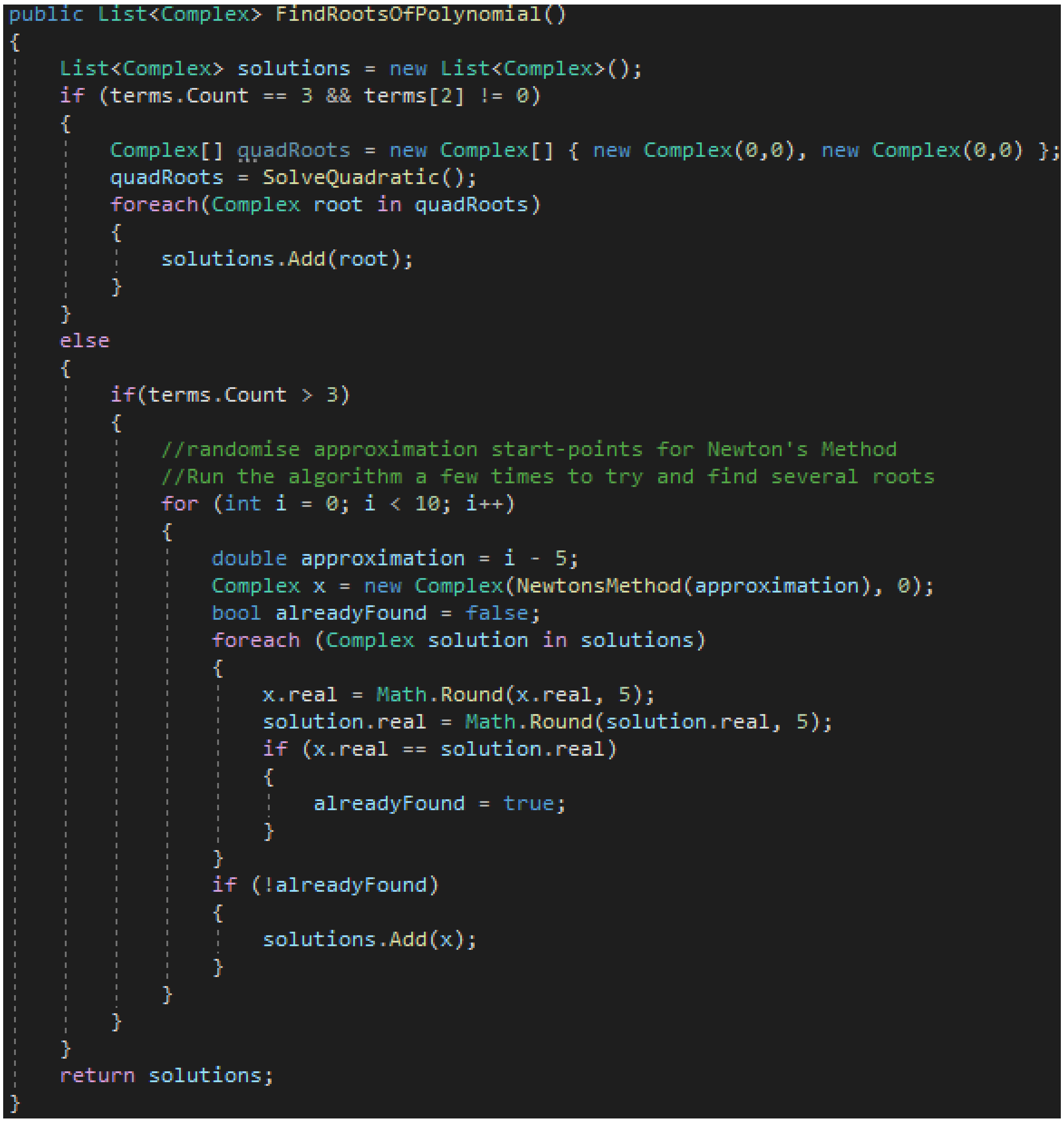


```
public Complex[] SolveQuadratic()
{
    Complex[] solutions = new Complex[] { new Complex(0, 0), new Complex(0, 0) };
    if(terms.Count == 3 && terms[2] != 0)
    {
        double a = terms[2];
        double b = terms[1];
        double c = terms[0];
        double root;
        if (!((b * b) - (4 * a * c) < 0))
        {
            root = Math.Sqrt((b * b) - (4 * a * c));
            solutions[0].real = (-b + root) / (2 * a);
            solutions[1].real = (-b - root) / (2 * a);
        }
        else
        {
            root = Math.Sqrt(Math.Abs((b * b) - (4 * a * c)));
            solutions[0].real = (-b) / 2;
            solutions[0].imaginary = root / 2;
            solutions[1].real = (-b) / 2;
            solutions[1].imaginary = (-root) / 2;
        }
    }
    return solutions;
}
```



```
public class Complex
{
    5 references
    public Complex(double rComponent, double iComponent) { real = rComponent;  imaginary = iComponent; }
    public double real; public double imaginary;
    1 reference
    public double ConvertToDouble()
    {
        if(imaginary == 0){ return real; }
        else return 9999999999999999;
    }
    0 references
    public void Add(Complex num) { real += num.real; imaginary += num.imaginary; }
    0 references
    public void Sub(Complex num) { real -= num.real; imaginary -= num.imaginary; }
    0 references
    public void MultiplyByComplex(Complex complex) { real = (real * complex.real) + (imaginary * complex.imaginary);
        imaginary = (real * complex.imaginary) + (imaginary * complex.real);
    }
    0 references
    public void DivideByComplex(Complex complex) {  real = (real / complex.real) + (imaginary / complex.imaginary);
        imaginary = (real / complex.imaginary) + (imaginary / complex.real);
    }
    0 references
    public void MultiplyByReal(double multiplicand)
    {
        real *= multiplicand;
        imaginary *= multiplicand;
    }
    0 references
    public void DivideByReal(double divisor)
    {
        real /= divisor;
        imaginary /= divisor;
    }
    1 reference
    public string GenerateString()
    {
        if(imaginary > 0)
        {
            //2 + 3i or 2 + 0i
            return real + " + " + imaginary + "i";
        }
        else if(imaginary == 0)
        {
            return Convert.ToString(real);
        }
        else return real + " - " + Math.Abs(imaginary) + "i";
    }
}
```

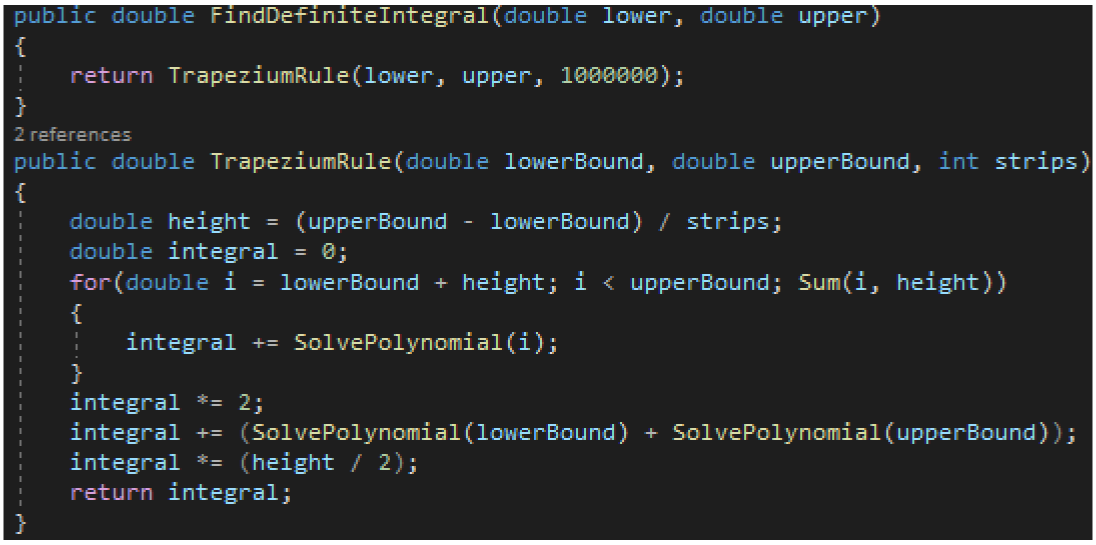
The rest of the FindRootsOfPolynomial() method applies NewtonsMethod() and creates the complex roots by making these the real components.



```
public List<Complex> FindRootsOfPolynomial()
{
    List<Complex> solutions = new List<Complex>();
    if (terms.Count == 3 && terms[2] != 0)
    {
        Complex[] quadRoots = new Complex[] { new Complex(0,0), new Complex(0,0) };
        quadRoots = SolveQuadratic();
        foreach(Complex root in quadRoots)
        {
            solutions.Add(root);
        }
    }
    else
    {
        if(terms.Count > 3)
        {
            //randomise approximation start-points for Newton's Method
            //Run the algorithm a few times to try and find several roots
            for (int i = 0; i < 10; i++)
            {
                double approximation = i - 5;
                Complex x = new Complex(NewtonsMethod(approximation), 0);
                bool alreadyFound = false;
                foreach (Complex solution in solutions)
                {
                    x.real = Math.Round(x.real, 5);
                    solution.real = Math.Round(solution.real, 5);
                    if (x.real == solution.real)
                    {
                        alreadyFound = true;
                    }
                }
                if (!alreadyFound)
                {
                    solutions.Add(x);
                }
            }
        }
    }
    return solutions;
}
```

As you can see, the constructor for Complex is used with NewtonsMethod to create a complex where the real part of it is NewtonsMethod applied to the first approximation, with the imaginary part being 0 each time as NewtonsMethod can only be used to find real roots. This is then checked against any solutions that have already been found, to ensure that there aren't repeats, as applying the algorithm in a close range of values, as shown here where the loop leads to the approximations used ranging from -5 to 5, results in multiple convergences to the same root.

Another function in my polynomial class is the TrapeziumRule(), which is a numerical method for finding a definite integral of an equation. As it finds a definite integral, it needs to have clear-cut discrete boundaries to integrate between, hence the method has inputs of a lowerBound and upperBound (as doubles). The trapezium rule can be used to varying degrees of accuracy as well, being based on creating numerous trapeziums along a curve, finding the areas of these and summing them up to find the area under a curve on a graph and thus a definite integral, hence you need to define how many trapeziums (or 'strips') you are using for your approximation of the definite integral in order for the formula to work, thus 'strips' is an input (as an integer). Below is a screenshot of the function, as well as FindDefiniteIntegral() which uses 1000000 strips in a trapezium rule to find the integral to such a high degree of accuracy that it cannot be distinguished from the genuine answer – numerical methods are approximations of results and not exact results, e.g. the value of pi can be approximated to good accuracy as 22/7 but it isn't exact, however the more iterations you do of a numerical method, the closer to the exact value it is, which is true for the trapezium rule whereby more trapeziums used means smaller and smaller intervals under the graph can be accounted for.



```
public double FindDefiniteIntegral(double lower, double upper)
{
    return TrapeziumRule(lower, upper, 1000000);
}
2 references
public double TrapeziumRule(double lowerBound, double upperBound, int strips)
{
    double height = (upperBound - lowerBound) / strips;
    double integral = 0;
    for(double i = lowerBound + height; i < upperBound; Sum(i, height))
    {
        integral += SolvePolynomial(i);
    }
    integral *= 2;
    integral += (SolvePolynomial(lowerBound) + SolvePolynomial(upperBound));
    integral *= (height / 2);
    return integral;
}
```

https://revisionmaths.com/advanced-level-maths-revision/pure-maths/calculus/trapezium-rule#:~:text=The%20trapezium%20rule%20works%20by,h%20(see%20diagram%20above).

The above link explains the trapezium rule with diagrams in great detail, with the main takeaway being the formula:

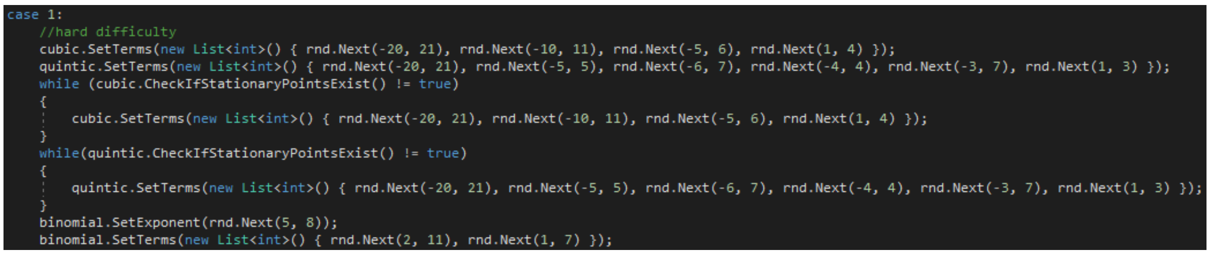
$$\int_{x_0}^{x_n} f(x)\,dx = \frac{1}{2}h[(y_0 + y_n) + 2(y_1 + y_2 + \ldots + y_{n-1})]$$

where $y_0 = f(x_0)$ and $y_1 = f(x_1)$ etc

. H = height of each strip, or the width of a single trapezium (each trapezium is equal in width). This can be calculated by finding the difference between the bounds and then dividing this by the number of strips to use.

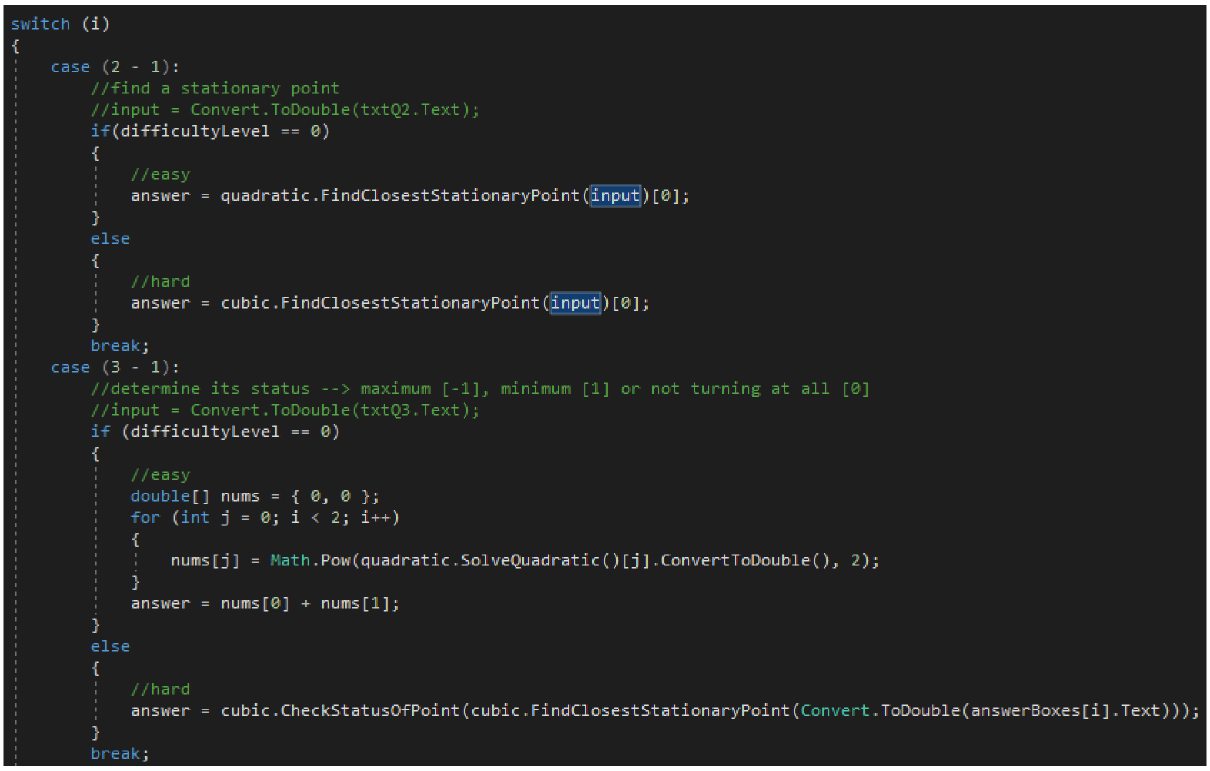
Overall, the reasoning behind why I designed my polynomial class in the way that I did was so that it is quick to get different bits of data from a polynomial and combine these with other methods to find other bits of data; for example, the Differentiate() method is utilised when finding stationary points, checking the status of stationary points and for finding roots in a polynomial, generating strings from polynomials can be applied to derivatives of polynomials, etc. The trapezium rule method, as described above this paragraph involves solving a polynomial over and over with different values of x so having a quick method to apply for this is incredibly efficient.

The maths quizzes utilise the polynomial class quite heavily and generate random polynomials multiple times, shown below:



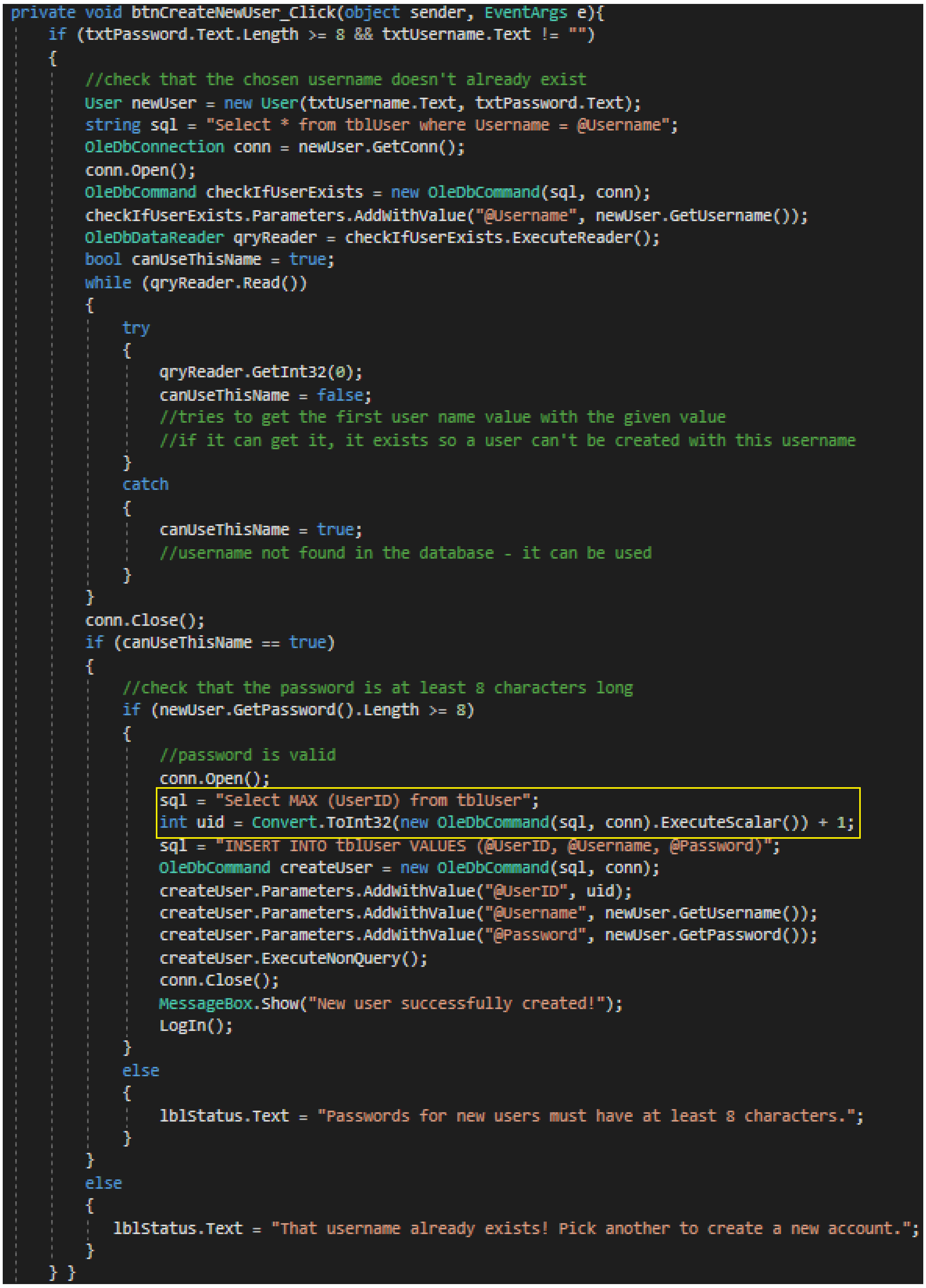
```
case 1:
    //hard difficulty
    cubic.SetTerms(new List<int>() { rnd.Next(-20, 21), rnd.Next(-10, 11), rnd.Next(-5, 6), rnd.Next(1, 4) });
    quintic.SetTerms(new List<int>() { rnd.Next(-20, 21), rnd.Next(-5, 5), rnd.Next(-6, 7), rnd.Next(-4, 4), rnd.Next(-3, 7), rnd.Next(1, 3) });
    while (cubic.CheckIfStationaryPointsExist() != true)
    {
        cubic.SetTerms(new List<int>() { rnd.Next(-20, 21), rnd.Next(-10, 11), rnd.Next(-5, 6), rnd.Next(1, 4) });
    }
    while(quintic.CheckIfStationaryPointsExist() != true)
    {
        quintic.SetTerms(new List<int>() { rnd.Next(-20, 21), rnd.Next(-5, 5), rnd.Next(-6, 7), rnd.Next(-4, 4), rnd.Next(-3, 7), rnd.Next(1, 3) });
    }
    binomial.SetExponent(rnd.Next(5, 8));
    binomial.SetTerms(new List<int>() { rnd.Next(2, 11), rnd.Next(1, 7) });
```

Question variety helps keep quizzes fresh, with the same format for practising the same skills but with different numbers. I used a switch statement for the difficulty handling whereby both the easy and hard quiz use the same form but based on which difficulty is passed to the form, they assign the questions differently; this applies for the marking too, where a switch statement is used for the iterative variable 'i', which cycles through each question and gets the answer from each one based on if the difficulty level is easy or hard which impacts what the answer gets stored as.



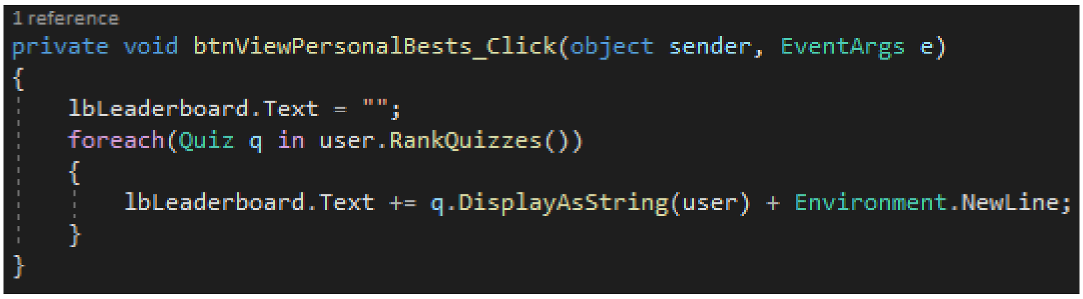
```
switch (i)
{
    case (2 - 1):
        //find a stationary point
        //input = Convert.ToDouble(txtQ2.Text);
        if(difficultyLevel == 0)
        {
            //easy
            answer = quadratic.FindClosestStationaryPoint(input)[0];
        }
        else
        {
            //hard
            answer = cubic.FindClosestStationaryPoint(input)[0];
        }
        break;
    case (3 - 1):
        //determine its status --> maximum [-1], minimum [1] or not turning at all [0]
        //input = Convert.ToDouble(txtQ3.Text);
        if (difficultyLevel == 0)
        {
            //easy
            double[] nums = { 0, 0 };
            for (int j = 0; i < 2; i++)
            {
                nums[j] = Math.Pow(quadratic.SolveQuadratic()[j].ConvertToDouble(), 2);
            }
            answer = nums[0] + nums[1];
        }
        else
        {
            //hard
            answer = cubic.CheckStatusOfPoint(cubic.FindClosestStationaryPoint(Convert.ToDouble(answerBoxes[i].Text)));
        }
        break;
```

The SQL used in my final design allows users create accounts, log in, do quizzes which, once completed, have the percentages saved to their accounts, and view a leader board of their personal best scores, which shows them the quiz type and percentage scored in the quiz. To do this, I created an old version Access database (2002 era – file extension = .mdb) and used OleDb in my program to access it. It has two tables: tblUser (for storing usernames and passwords as strings, with an autonumber UserID as the primary key) and tblQuiz (for storing quiz type, the UserID of the person who did the quiz, the percentage they scored and a QuizID for the primary key). It is in first normal form as data is atomic and there is a primary key per table to make records unique to access, with no repeating values; e.g. each username must be unique, with each user id being unique due to an algorithm I implemented; I was going to use the built-in autonumber data type for the primary keys of my tables as this would allow new records to by default have a new ID each time however my program encountered issues with this so I instead decided to make the primary keys (UserID in the user table and QuizID in the quiz table) just number data types and then when inserting new records, I used SQL to find the current largest ID to which I added 1 to before inserting a new record with that primary key value. Below is an example of this in the CreateUser() method.



```
private void btnCreateNewUser_Click(object sender, EventArgs e){
    if (txtPassword.Text.Length >= 8 && txtUsername.Text != "")
    {
        //check that the chosen username doesn't already exist
        User newUser = new User(txtUsername.Text, txtPassword.Text);
        string sql = "Select * from tblUser where Username = @Username";
        OleDbConnection conn = newUser.GetConn();
        conn.Open();
        OleDbCommand checkIfUserExists = new OleDbCommand(sql, conn);
        checkIfUserExists.Parameters.AddWithValue("@Username", newUser.GetUsername());
        OleDbDataReader qryReader = checkIfUserExists.ExecuteReader();
        bool canUseThisName = true;
        while (qryReader.Read())
        {
            try
            {
                qryReader.GetInt32(0);
                canUseThisName = false;
                //tries to get the first user name value with the given value
                //if it can get it, it exists so a user can't be created with this username
            }
            catch
            {
                canUseThisName = true;
                //username not found in the database - it can be used
            }
        }
        conn.Close();
        if (canUseThisName == true)
        {
            //check that the password is at least 8 characters long
            if (newUser.GetPassword().Length >= 8)
            {
                //password is valid
                conn.Open();
                sql = "Select MAX (UserID) from tblUser";
                int uid = Convert.ToInt32(new OleDbCommand(sql, conn).ExecuteScalar()) + 1;
                sql = "INSERT INTO tblUser VALUES (@UserID, @Username, @Password)";
                OleDbCommand createUser = new OleDbCommand(sql, conn);
                createUser.Parameters.AddWithValue("@UserID", uid);
                createUser.Parameters.AddWithValue("@Username", newUser.GetUsername());
                createUser.Parameters.AddWithValue("@Password", newUser.GetPassword());
                createUser.ExecuteNonQuery();
                conn.Close();
                MessageBox.Show("New user successfully created!");
                LogIn();
            }
            else
            {
                lblStatus.Text = "Passwords for new users must have at least 8 characters.";
            }
        }
        else
        {
           lblStatus.Text = "That username already exists! Pick another to create a new account.";
        }
    } }
```

I decided to create a class for User which allows you to very easily pass information from the database to other parts of the application once you have logged in; for example, a single user object can store their username, password and quiz scores, as well as have methods within it which makes getting any other data from the database quick and efficient to do. One example of this is the personal leader-board generator, shown below:



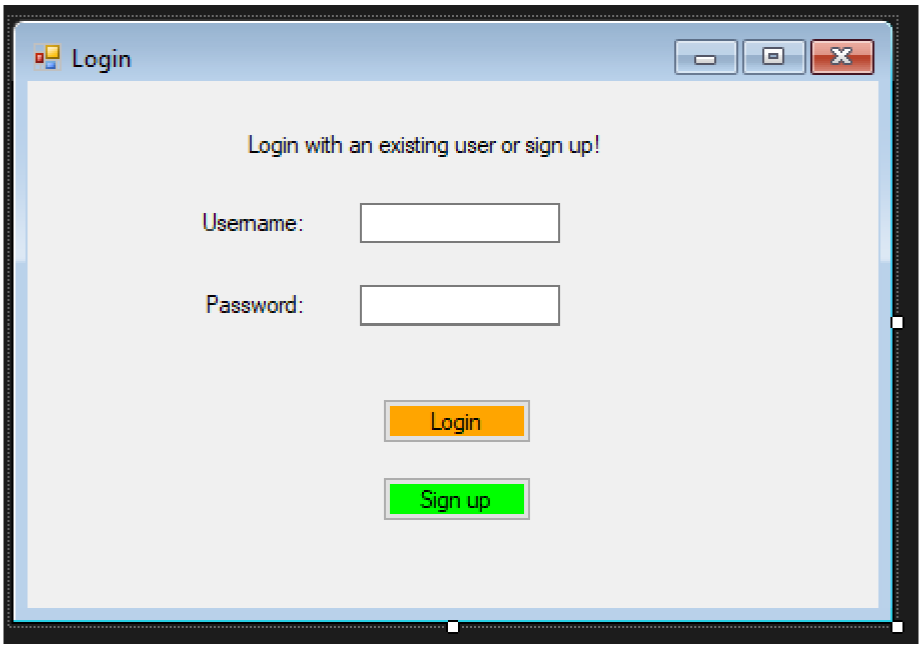
```
1 reference
private void btnViewPersonalBests_Click(object sender, EventArgs e)
{
    lbLeaderboard.Text = "";
    foreach(Quiz q in user.RankQuizzes())
    {
        lbLeaderboard.Text += q.DisplayAsString(user) + Environment.NewLine;
    }
}
```

We can simply click the button and it calls functions on the user to get the quizzes from that specific user through a SQL connection contained in the user, rank them and then write them to the text display.

Below are screenshots of final UI designs for all the forms

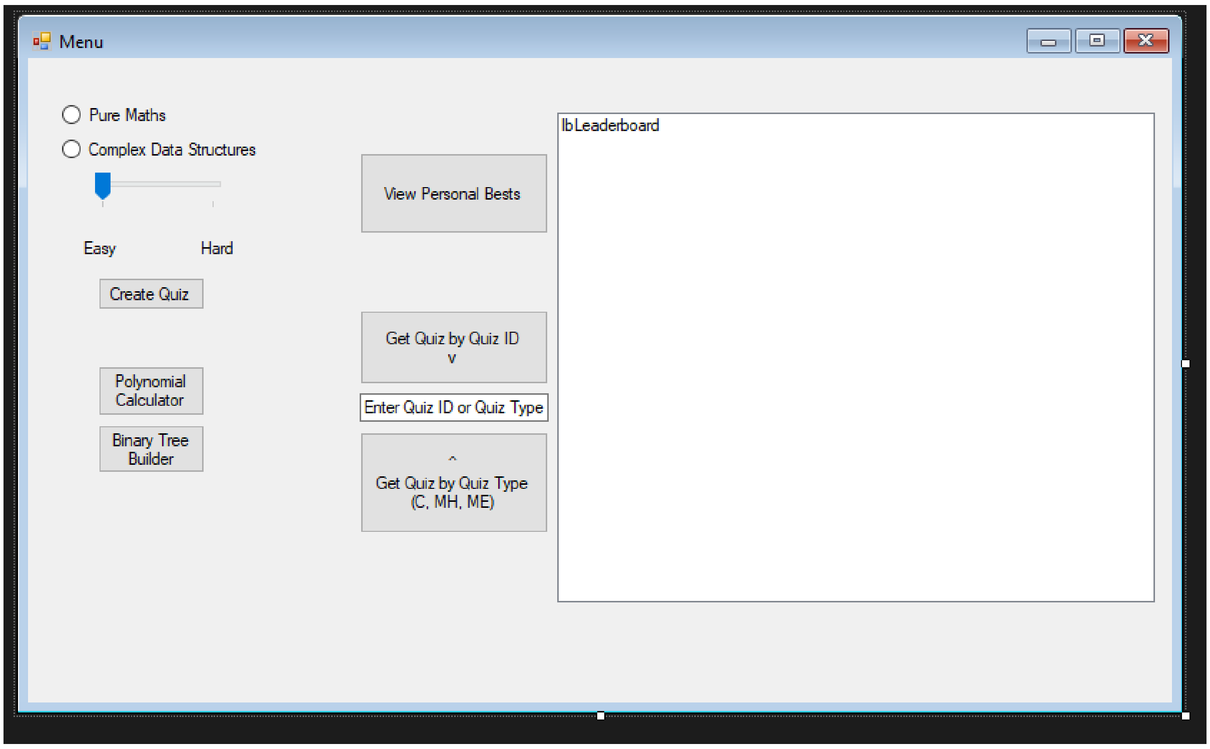
==

Login



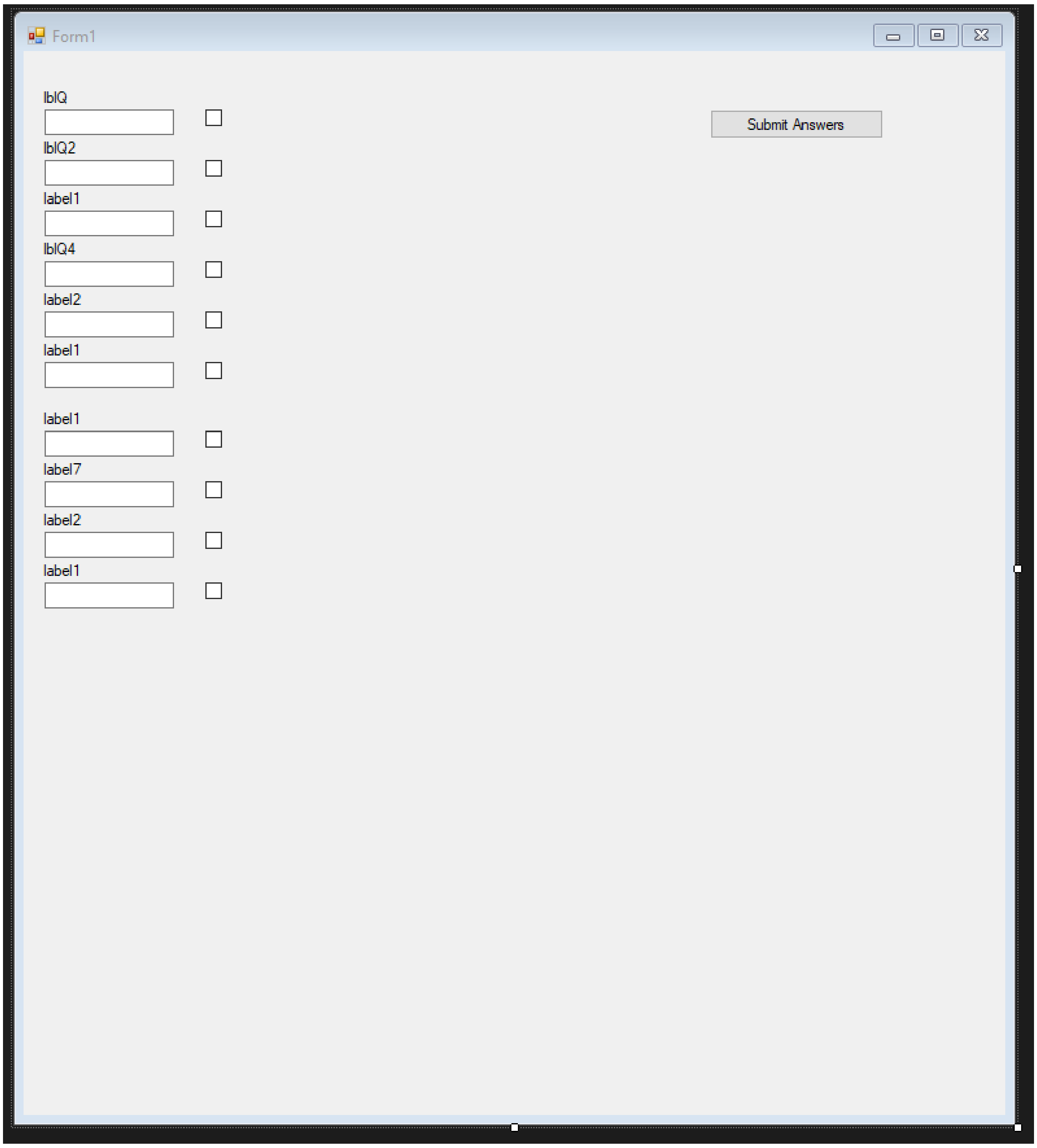
==

Menu (allows you to choose a quiz to do, view your personal leader board and pick polynomial/tree creation forms to load and use for educational purposes)



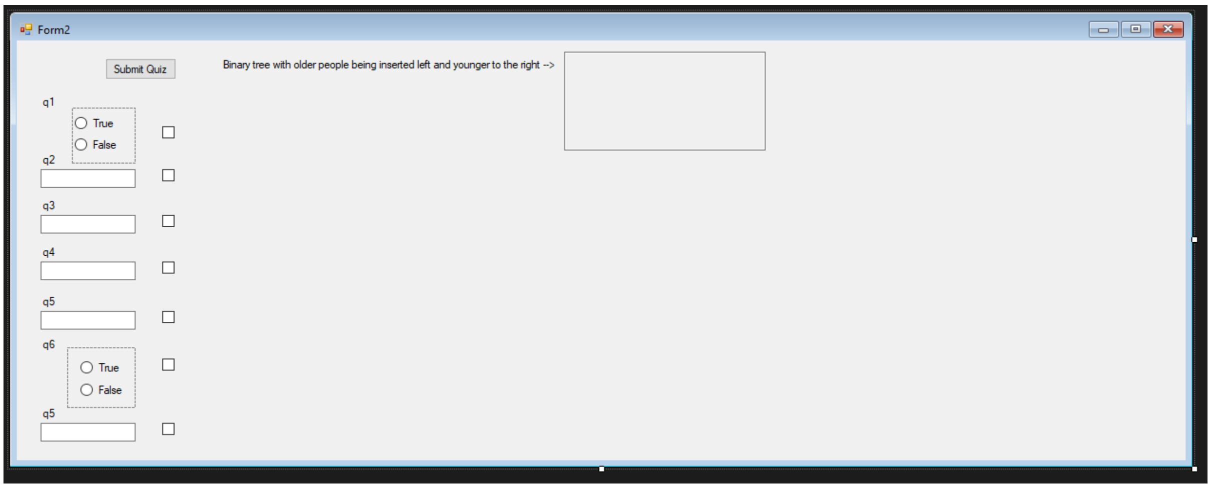
==

Form1 (the maths quiz) – it is the same form for the easy and hard modes, with different questions and marking methods depending on which mode you selected



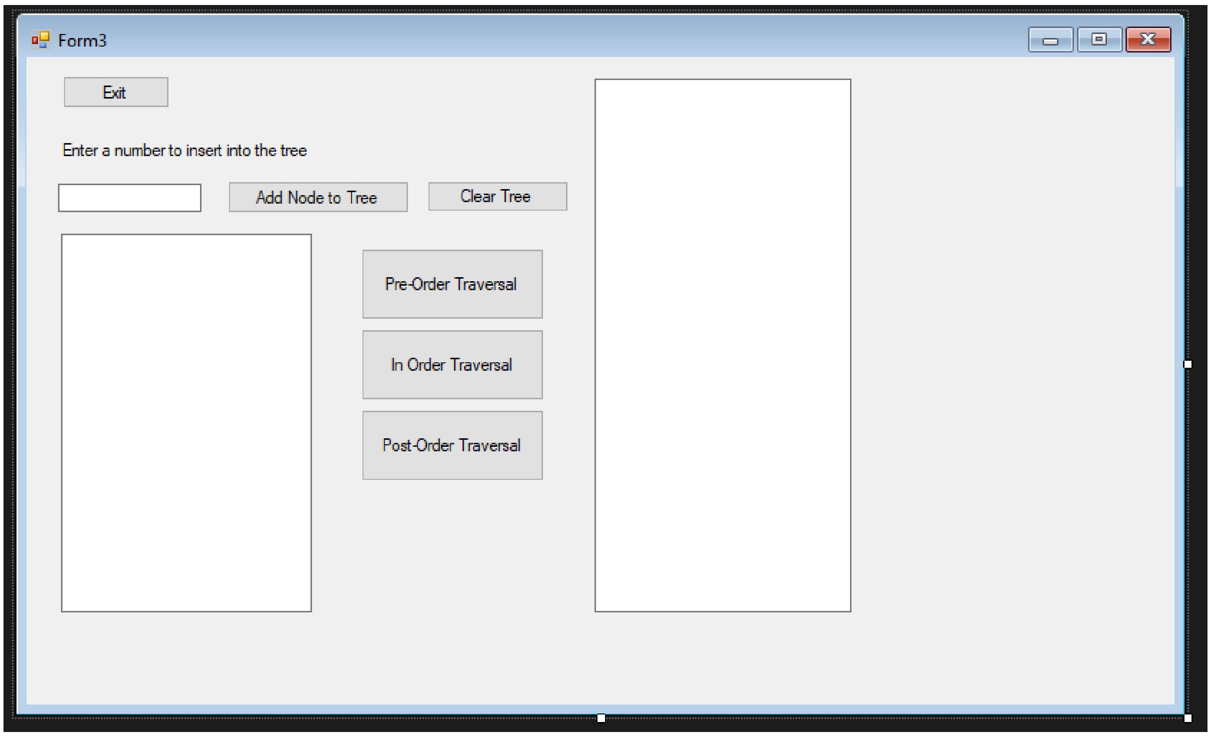
==

Form2 (the computing quiz)


Form2
Submit Quiz
Binary tree with older people being inserted left and younger to the right -->
q1
True
False
q2
q3
q4
q5
q6
True
False
q5

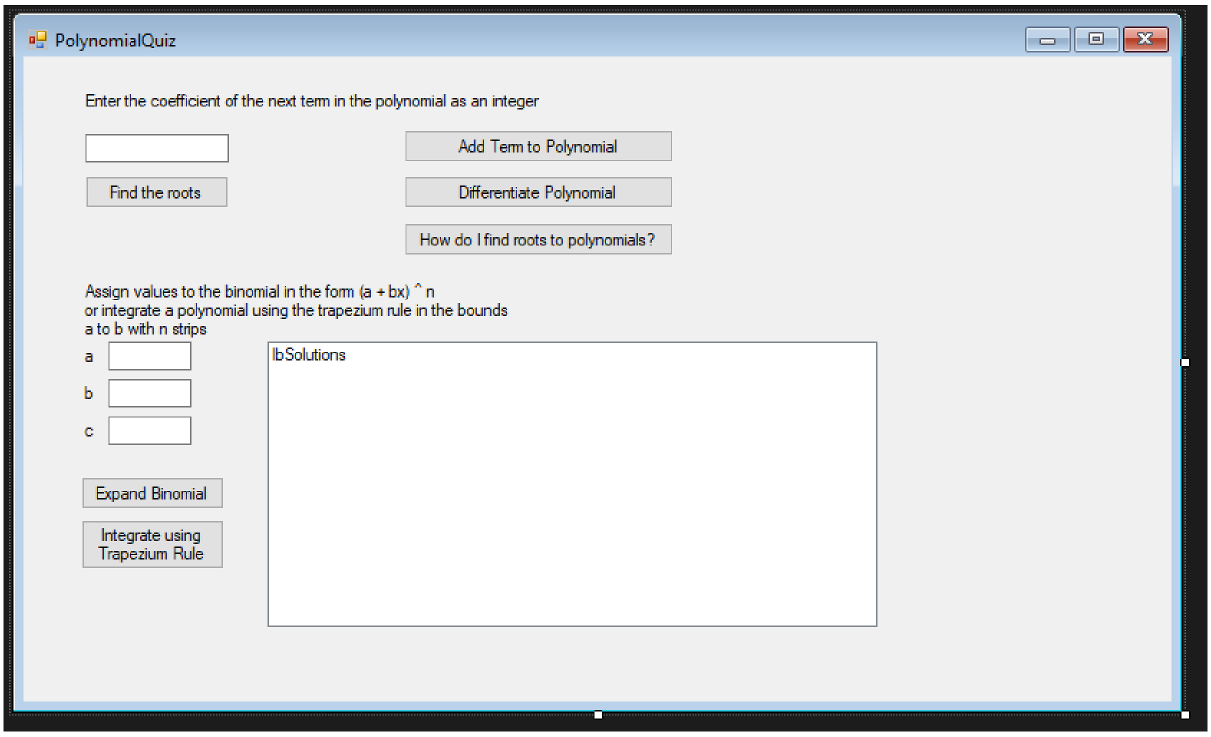
==

Form3 (the form for creating numeric binary trees and performing different traversals on them)



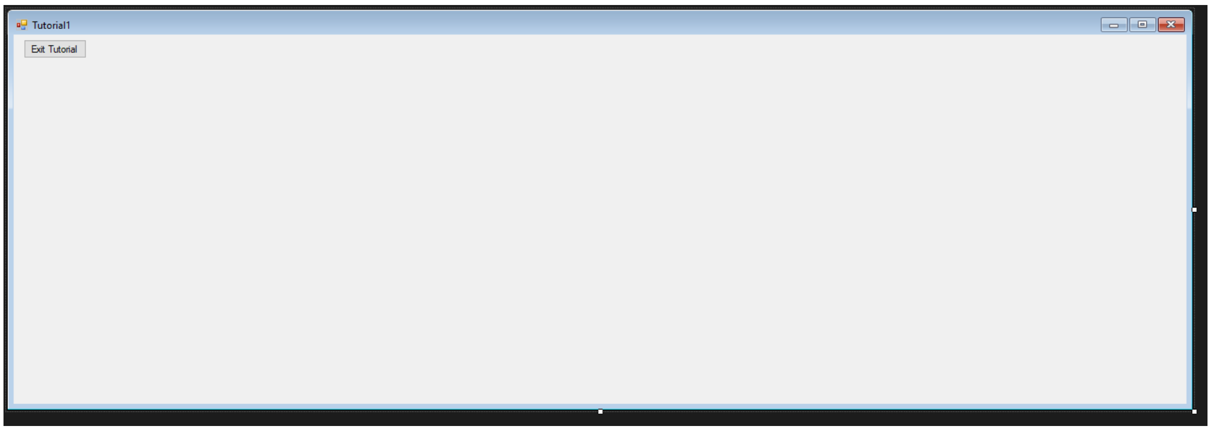
==

PolynomialQuiz (isn't used for quizzes, is actually used to create polynomials and solve them, differentiate them, integrate them, etc.)


PolynomialQuiz
Enter the coefficient of the next term in the polynomial as an integer
Add Term to Polynomial
Find the roots
Differentiate Polynomial
How do I find roots to polynomials?
Assign values to the binomial in the form (a + bx) ^ n
or integrate a polynomial using the trapezium rule in the bounds
a to b with n strips
a
b
c
lbSolutions
Expand Binomial
Integrate using Trapezium Rule

==

Tutorial1 (has a label which gets text loaded into it telling the user how different mathematical methods like differentiation works, can be loaded from the PolynomialQuiz form)



Technical Solution

Form1 aka the maths quiz form

```
using System;

using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PolynomialQuiz
{
    public partial class Form1 : Form
    {
        double[] bounds = { 0, 0};
        User user;
        int score = 0;
        Polynomial quadratic = new Polynomial();
        string[] q1Answer = { "", "" };
        Polynomial[] TrueOrFalse = { new Polynomial(), new Polynomial() };
        Polynomial quadraticToAdd = new Polynomial();
        Polynomial discQuestionQuad = new Polynomial();
        int TrueOrFalseXValue;
        Polynomial cubic = new Polynomial();
        Polynomial quintic = new Polynomial();
        Binomial binomial = new Binomial();
        List<Label> questions = new List<Label>();
        List<TextBox> answerBoxes = new List<TextBox>();
        List<CheckBox> markScheme = new List<CheckBox>();
        List<Label> correctAnswers = new List<Label>();
        int difficultyLevel;
        public Form1(int difficulty, User player)
        {
            //difficulty --> 0 = easy, 1 = hard
            InitializeComponent();
            difficultyLevel = difficulty;
            user = player;

        }

        private void Form1_Load(object sender, EventArgs e)

        {

            quadratic.GenerateFactorisableQuadratic();

            q1Answer = quadratic.ShowFactorsOfQuadratic();

            questions = new List<Label>() { lblQ1, lblQ2, lblQ3, lblQ4, lblQ5,
lblQ6, lblQ7, lblQ8, lblQ9, lblQ10 };

            answerBoxes = new List<TextBox>() { txtQ1, txtQ2, txtQ3, txtQ4, txtQ5,
txtQ6, txtQ7, txtQ8, txtQ9, txtQ10 };

            markScheme = new List<CheckBox>() { cb1, cb2, cb3, cb4, cb5, cb6, cb7,
cb8, cb9, cb10 };

            correctAnswers = new List<Label>() { lblCA1, lblCA2, lblCA3, lblCA4,
lblCA5, lblCA6, lblCA7, lblCA8, lblCA9, lblCA10 };

            foreach (CheckBox cb in markScheme)

            {

                cb.Enabled = false;

                //users cannot unlawfully check the checkboxes and give themselves
higher scores

            }

            Random rnd = new Random();

            switch (difficultyLevel)

            {

                case 0:

                    //easy difficulty

                    for (int i = 0; i < questions.Count; i++)

                    {

                        switch (i)

                        {

                            case 1 - 1:

                                //Factorise the quadratic

                                questions[i].Text = "Find the factors of the
quadratic in the exact format of (ax + b)(cx + d) - don't omit 1s and 0s from your
answer, e.g. (1x + 0)(2x + 3): " + quadratic.GenerateString();

                                break;

                            case 2 - 1:
                                //Find the quadratic's stationary point

                                questions[i].Text = "Find the x-coordinate of the
quadratic's stationary point";

                                break;

                            case 3 - 1:

                                //Find the roots of the quadratic, square them and
sum them

                                questions[i].Text = "The roots of the quadratic
are the opposite and adjacent sides of a triangle. What is the square of the
hypotenuse?";

                                break;

                            case 4 - 1:

                                //Find the quadratic's derivative and find the
highest power of x's coefficient

                                questions[i].Text = "Find the derivative of the
quadratic and type its highest power of x's coefficient";

                                break;

                            case 5 - 1:

                                //Integrate the quadratic and find its x^3
coefficient --> a/3

                                questions[i].Text = "Integrate the quadratic and
type the coefficient of x^3 in the box to a max of 3.d.p";

                                break;

                            case 6 - 1:

                                //Sum two quadratics

                                quadraticToAdd.GenerateRandomPolynomial(2, 1, 4);

                                questions[i].Text = "Add the following quadratics:
" + quadraticToAdd.GenerateString() + " , " + quadratic.GenerateString() + "; What
is the value of the coefficient of x?";

                                break;

                            case 7 - 1:

                                //find definite integral

                                bounds[0] = rnd.Next(0, 4);

                                bounds[1] = rnd.Next(5, 9);

                                questions[i].Text = "If y is acceleration and x is
time, find the velocity of the graph y = " + quadratic.GenerateString() + "
between " + bounds[0] + " and " + bounds[1] + " seconds to the nearest integer";

                                break;
                            case 8 - 1:

                                //find coordinates of the y-intercept --> x = 0

                                questions[i].Text = "What is the y-coordinate of
the y-intercept of f'(x) if f(x) = " +
quadratic.MultiplyPolynomial(4).GenerateString(); ;

                                break;

                            case 9 - 1:

                                //True or False

                                TrueOrFalse[0].GenerateRandomPolynomial(3, 0, 10);

                                TrueOrFalse[1].GenerateRandomPolynomial(4, 1, 5);

                                TrueOrFalseXValue = rnd.Next(2, 5);

                                questions[i].Text = "True (1) or False (0): " +
TrueOrFalse[0].GenerateString() + " is larger than " +
TrueOrFalse[1].GenerateString() + "when X is " + TrueOrFalseXValue;

                                break;

                            case 10 - 1:

                                //Discriminant True or False

                                discQuestionQuad.CreateQuadraticWithOneRoot();

                                discQuestionQuad.MultiplyPolynomialByInt(3);

                                discQuestionQuad =
discQuestionQuad.SumPolynomial(quadraticToAdd);

                                questions[i].Text = "True (1) or False (0): the
following quadratic only has one real root --> " +
discQuestionQuad.GenerateString();

                                break;

                        }

                    }

                    break;

                case 1:

                    //hard difficulty

                    cubic.SetTerms(new List<int>() { rnd.Next(-20, 21), rnd.Next(-
10, 11), rnd.Next(-5, 6), rnd.Next(1, 4) });

                    quintic.SetTerms(new List<int>() { rnd.Next(-20, 21),
rnd.Next(-5, 5), rnd.Next(-6, 7), rnd.Next(-4, 4), rnd.Next(-3, 7), rnd.Next(1, 3)
});

                    while (cubic.CheckIfStationaryPointsExist() != true)

                    {
                        cubic.SetTerms(new List<int>() { rnd.Next(-20, 21),
rnd.Next(-10, 11), rnd.Next(-5, 6), rnd.Next(1, 4) });

                    }

                    while(quintic.CheckIfStationaryPointsExist() != true)

                    {

                        quintic.SetTerms(new List<int>() { rnd.Next(-20, 21),
rnd.Next(-5, 5), rnd.Next(-6, 7), rnd.Next(-4, 4), rnd.Next(-3, 7), rnd.Next(1, 3)
});

                    }

                    binomial.SetExponent(rnd.Next(5, 8));

                    binomial.SetTerms(new List<int>() { rnd.Next(2, 11),
rnd.Next(1, 7) });

                    for (int i = 0; i < questions.Count; i++)

                    {

                        switch (i)

                        {

                            case 1 - 1:

                                //find a correct root

                                questions[i].Text = "Find a real root of the
following polynomial (for all answers, find them upto 3.d.p if the answer is a
decimal: " + cubic.GenerateString();

                                break;

                            case 2 - 1:

                                //find a stationary point

                                questions[i].Text = "For the above polynomial,
find the x-coordinate of one of its stationary points";

                                break;

                            case 3 - 1:

                                //determine its status

                                questions[i].Text = "Assuming you found a
stationary point, determine whether it is maximum [-1], minimum [1] or not turning
at all [0]";

                                break;

                            case 4 - 1:

                                //find a correct root

                                questions[i].Text = "Find a real root of the
following polynomial upto a max of 3.d.p if the answer is a decimal: " +
quintic.GenerateString();
```

```
                                break;

                            case 5 - 1:

                                //find a stationary point

                                questions[i].Text = "For the above polynomial,
find the x-coordinate of one of its stationary points";

                                break;

                            case 6 - 1:

                                //determine its status

                                questions[i].Text = "Assuming you found a
stationary point, determine whether it is maximum [-1], minimum [1] or not turning
at all [0]";

                                break;

                            case 7 - 1:

                                //find definite integral

                                bounds[0] = rnd.Next(0, 4);

                                bounds[1] = rnd.Next(5, 9);

                                questions[i].Text = "If y is velocity and x is
time, find the displacement of the graph y = " + quintic.GenerateString() + "
between "

                                    + bounds[0] +" and " + bounds[1] + " seconds,
truncated to the nearest integer";

                                break;

                            case 8 - 1:

                                //determine the coefficient of a term in an
expanded binomial

                                questions[i].Text = "Find the coefficient of x^2
in the expansion of the following binomial: " + binomial.ShowFactorisedBinomial();

                                break;

                            case 9 - 1:

                                //find gradient at specific point

                                questions[i].Text = "What is the gradient of f(x)
at when x = 6 if f(x) = " + quintic.GenerateString();

                                break;

                            case 10 - 1:

                                //find the gradient of the quintic and the
coefficient of its x^4

                                questions[i].Text = "What is the coefficient of
x^4 in f'(x) if f(x) = " + quintic.GenerateString();
```

```
                                break;
                        }
                    }
                    break;
            }
        }

        private void btnSubmit_Click(object sender, EventArgs e)
        {
            score = 0;
            bool correctFormat = true;
            double test = 0;
            foreach(TextBox txt in answerBoxes)
            {
                //verify all inputs are valid --> can be converted to doubles and
not just random characters that are only strings
                try
                {
                    if(test != 0)
                    {
                        test = Convert.ToDouble(txt.Text);
                    }
                }
                catch
                {
                    correctFormat = false;
                }
                test++;
            }
            if (correctFormat)
            {
                //all inputs are valid --> start checking answers
                double input = 0;
                double answer = 0;
                for (int i = 0; i < questions.Count; i++)
                {
                    if(i != (1-1))
                    {
                        input = Math.Round(Convert.ToDouble(answerBoxes[i].Text),
3);
                    }
                    if(i == (1 - 1))
                    {
                        if(difficultyLevel == 0)
                        {
                            //easy
                            //Factorise the quadratic --> (ax + b)(cx + d)
                            if (answerBoxes[i].Text == q1Answer[0] ||
answerBoxes[i].Text == q1Answer[1])
                            {
                                //correct
                                markScheme[i].Checked = true;
                                score++;
                            }
                            else
                            {
                                markScheme[i].Checked = false;
                                correctAnswers[i].Text = q1Answer[0];
                                correctAnswers[i].ForeColor = Color.IndianRed;
                            }
                        }
                        else
                        {
                            //hard
                            //find a correct root --> apply Newton's Method to
their answer
                            //input = Convert.ToDouble(answerBoxes[i].Text);
                            input =
cubic.SolvePolynomial(Convert.ToDouble(answerBoxes[i].Text));
```

```
            if(-0.01 < input && input < 0.01)
            {
                //answer is correct
                markScheme[i].Checked = true;
                score++;
            }
            else
            {
                //answer is incorrect
                markScheme[i].Checked = false;
                answer = cubic.FindCorrectRoot();
                correctAnswers[i].Text = Convert.ToString(answer);
                correctAnswers[i].ForeColor = Color.IndianRed;
            }
            //answer = cubic.NewtonsMethod(input);
            //if (Math.Round(answer, 3) == input)
            //{
            //    //answer is correct
            //    markScheme[i].Checked = true;
            //    score++;
            //}
            //else
            //{

            //}
        }
    }
    else
    {
        switch (i)
        {
            case (2 - 1):
                //find a stationary point
                //input = Convert.ToDouble(txtQ2.Text);
```

```
                                if(difficultyLevel == 0)
                                {
                                    //easy
                                    answer =
quadratic.FindClosestStationaryPoint(input)[0];
                                }
                                else
                                {
                                    //hard
                                    answer =
cubic.FindClosestStationaryPoint(input)[0];
                                }
                                break;
                            case (3 - 1):
                                //determine its status --> maximum [-1], minimum
[1] or not turning at all [0]
                                //input = Convert.ToDouble(txtQ3.Text);
                                if (difficultyLevel == 0)
                                {
                                    //easy
                                    double[] nums = { 0, 0 };
                                    for (int j = 0; i < 2; i++)
                                    {
                                        nums[j] =
Math.Pow(quadratic.SolveQuadratic()[j].ConvertToDouble(), 2);
                                    }
                                    answer = nums[0] + nums[1];
                                }
                                else
                                {
                                    //hard
                                    answer =
cubic.CheckStatusOfPoint(cubic.FindClosestStationaryPoint(Convert.ToDouble(answerB
oxes[i].Text)));
                                }
                                break;
                            case (4 - 1):
                                //input = Convert.ToDouble(txtQ4.Text);
                                switch (difficultyLevel)
                                {
                                    case 0:
                                        //easy
                                        //Find the coefficient of its derivative's
highest power
                                        answer =
quadratic.Differentiate().GetTerms()[1];
                                        break;
                                    case 1:
                                        //hard
                                        //find a correct root
                                        answer = quintic.NewtonsMethod(input);
                                        if(answer == 9999)
                                        {
                                            answer = quintic.FindCorrectRoot();
                                        }
                                        break;
                                }
                                break;
                            case (5 - 1):
                                //input = Convert.ToDouble(txtQ5.Text);
                                switch (difficultyLevel)
                                {
                                    case 0:
                                        //easy
                                        //integrate quadratic and find coefficient
of x^3 --> a/3 to 3.d.p
                                        answer =
Convert.ToDouble(quadratic.GetTerms()[2]) / 3;
                                        break;
                                    case 1:
                                        //hard
                                        //find a stationary point

                                        answer = 
quintic.FindClosestStationaryPoint(input)[0];

                                        break;

                                    }

                                    break;

                                case (6 - 1):

                                    //input = Convert.ToDouble(txtQ6.Text);

                                    switch (difficultyLevel)

                                    {

                                        case 0:

                                        //easy

                                        //sum two quadratics

                                        answer = 
quadratic.SumPolynomial(quadraticToAdd).GetTerms()[1];

                                        break;

                                    case 2:

                                        //hard

                                        //determine its status

                                        double userInput = 
Convert.ToDouble(txtQ5.Text);

                                        answer = 
quintic.CheckStatusOfPoint(quintic.FindClosestStationaryPoint(userInput));

                                        //for some reason, the above line kept on 
throwing decimal values so I added the code below to debug

                                        if((answer > 0 && userInput > 0 )||
(answer < 0 && userInput < 0) || (answer == 0 && userInput == 0))

                                        {

                                            answer = input;

                                        }

                                        break;

                                    }

                                    break;

                                case (7 - 1):

                                    //find definite integral

                                    if (difficultyLevel == 0)
                                    {
                                        //apply the trapezium rule to a quadratic
                                        answer =
Math.Floor(quadratic.FindDefiniteIntegral(bounds[0], bounds[1]));
                                    }
                                    else
                                    {
                                        //apply the trapezium rule to a quintic
                                        answer =
Math.Floor(quintic.FindDefiniteIntegral(bounds[0], bounds[1]));
                                    }
                                    break;
                                case (8 - 1):
                                    //input = Convert.ToDouble(txtQ8.Text);
                                    if(difficultyLevel == 0)
                                    {
                                        //easy - find y coordinate of the y-intercept
of a derivative --> x = 0
                                        // y = ax^2 + bx + c
                                        answer =
quadratic.MultiplyPolynomial(4).Differentiate().SolvePolynomial(0);
                                    }
                                    else
                                    {
                                        //hard - determine the coefficient of a term
in an expanded binomial
                                        answer =
binomial.DetermineCoefficientOfSpecificTerm(binomial.GetExponent() - 2);
                                    }
                                    break;
                                case 9 - 1:
                                    if(difficultyLevel == 0)
                                    {
                                        //easy --> is [0] > [1], True (1) or False (0)
                                        if
(TrueOrFalse[0].SolvePolynomial(TrueOrFalseXValue) >
TrueOrFalse[1].SolvePolynomial(TrueOrFalseXValue))
```

```
                                        {
                                            answer = 1;
                                        }
                                        else
                                        {
                                            answer = 0;
                                        }
                                    }
                                    else
                                    {
                                        //hard --> find gradient at specific point
                                        answer =
quintic.Differentiate().SolvePolynomial(6);
                                    }
                                    break;
                                case 10 - 1:
                                    if(difficultyLevel == 0)
                                    {
                                        //easy --> does the quadratic have only one
root? 1 - true, 0 - false

if(discQuestionQuad.GetDiscriminantFromQuadratic() == 0)
                                        {
                                            answer = 1;
                                        }
                                        else
                                        {
                                            answer = 0;
                                        }
                                    }
                                    else
                                    {
                                        //hard --> find coefficient from derivative
                                        answer =
quintic.Differentiate().GetTerms()[4];
```

```
                }
                break;
        }
        if (Math.Round(answer, 3) == input)
        {
            //answer is correct

            markScheme[i].Checked = true;
            score++;
        }
        else
        {
            //answer is incorrect
            markScheme[i].Checked = false;
            correctAnswers[i].Text = Convert.ToString(answer);
            correctAnswers[i].ForeColor = Color.IndianRed;
        }
    }
}
//show user their score out of 10
//save score and user to the database as well as quiz type
score *= 10;
MessageBox.Show("Great attempt! You got " + score + "%");
//save score to database
string quizType = "";
if(difficultyLevel == 0)
{
    quizType = "ME";
}
else
{
    quizType = "MH";

}
```

```
                //saves score as a percentage -> score / 10 * 100 --> score x 10

                Quiz q = new Quiz(user.GetUserID(), quizType, score);

                user.SaveQuizScore(q);

                MessageBox.Show("Result stored! QuizID = " + q.GetQuizID());

            }

            else

            {

                //an input is a string --> alert user that they need to make sure
all answers are numeric

                MessageBox.Show("An error was detected with one of your answers;
please make sure they are all numbers!");

            }


        }

    }

}
```

Polynomial class

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Drawing;


namespace PolynomialQuiz

{

    //class Term : MathsMethods

    //{

    //    public Term(Object o, string operation, bool hasChild)

    //    {

    //        if(Convert.ToString(o) == "x" && operation == "" && hasChild ==
false)

    //        {
```

```
    //            isJustX = true;
    //            num = "x";
    //            code = "";
    //            hasChildTerm = false;
    //        }
    //        else
    //        {
    //            num = o;
    //            code = operation;
    //            if (!ValidateOperation())
    //            {
    //                code = "";
    //            }
    //            else
    //            {
    //                hasChildTerm = hasChild;
    //            }
    //        }
    //    }
    //    public bool isJustX;
    //    Object num;
    //    string code;
    //    Term child;
    //    public bool hasChildTerm;
    //    public static string[] operations = { "log", "ln", "sin", "cos", "tan",
"sec", "csc", "cot", "e" };
    //    public string GetCode()
    //    {
    //        return code;
    //    }
    //    public Object GetNum()
    //    {
    //        return num;
    //    }
    //    public Term GetChild()
    //    {
    //        return child;
    //    }
    //    public bool ValidateOperation()
    //    {
    //        bool valid = false;
    //        byte b = 0;
    //        while (b < operations.Length && valid == false)
    //        {
    //            if (operations[b] == code)
    //            {
    //                valid = true;
    //            }
    //            b++;
    //        }
    //        return valid;
    //    }
    //    public void AddChild(Term t)
    //    {
    //        if(hasChildTerm == true)
    //        {
    //            if (ValidateOperation())
    //            {
    //                child = t;
    //            }
    //        }
    //    }
    //    public double GenerateNumWithXSubstituted(double x)
    //    {
    //        Term t = this;
    //        t.isJustX = false;
    //        t.num = x;
    //        return t.GenerateNum();
    //    }
    //    public double GenerateNum()
    //    {
    //        if (isJustX)
    //        {
    //            return 0;
    //        }
    //        else
    //        {
    //            double number;
    //            try
    //            {
    //                number = Convert.ToDouble(num);
    //            }
    //            catch
    //            {
    //                number = 1;
    //            }
    //            if (ValidateOperation() && hasChildTerm)
    //            {
    //                switch (code)
    //                {
    //                    case "sin":
    //                        try
    //                        {
    //                            number *= Math.Sin(child.GenerateNum());
    //                        }
    //                        catch
    //                        {

    //                        }
    //                        break;
    //                    case "cos":
    //                        try
    //                          {
    //                              number *= Math.Cos(child.GenerateNum());
    //                          }
    //                          catch
    //                          {

    //                          }
    //                          break;
    //                      case "tan":
    //                          try
    //                          {
    //                              number *= Math.Tan(child.GenerateNum());
    //                          }
    //                          catch
    //                          {

    //                          }
    //                          break;
    //                      case "sec":
    //                          try
    //                          {
    //                              number *= Sec(child.GenerateNum());
    //                          }
    //                          catch
    //                          {

    //                          }
    //                          break;
    //                      case "csc":
    //                          try
    //                          {
    //                              number *= Csc(child.GenerateNum());
    //                          }
    //                          catch
    //                        {

    //                        }
    //                        break;
    //                    case "cot":
    //                        try
    //                        {
    //                            number *= Cot(child.GenerateNum());
    //                        }
    //                        catch
    //                        {

    //                        }
    //                        break;
    //                    case "e":
    //                        number *= Math.E;
    //                        break;
    //                    case "log":
    //                        number = Math.Log10(number);
    //                        break;
    //                    case "ln":
    //                        number = Math.Log(number);
    //                        break;
    //                }
    //            }
    //            try
    //            {
    //                return Convert.ToDouble(num);
    //            }
    //            catch
    //            {
    //                return 1;
    //            }
    //        }
```

```
    //    }
    //    public string GenerateText()
    //    {
    //        if (isJustX)
    //        {
    //            return "x";
    //        }
    //        if (hasChildTerm == false)
    //        {
    //            return Convert.ToString(num);
    //        }
    //        else
    //        {
    //            if (ValidateOperation())
    //            {
    //                return Convert.ToString(num) + code + "(" +
child.GenerateText() + ")";
    //            }
    //            else
    //            {
    //                return Convert.ToString(num);
    //            }
    //        }
    //    }
    //}
    //class Model
    //{
    //    //questions to be asked may involve differentiating the given
model/expression
    //    List<Term> terms = new List<Term>();
    //    byte type = 0;
    //    Polynomial p;
    //    public void Clear()
    //    {
    //        terms.Clear();
    //        type = 0;
    //    }
    //    public string GenerateText()
    //    {
    //        string text = "";
    //        if (type == 1)
    //        {
    //            // y = a + b * ( c ^ [k*x] )
    //            text = terms[0].GenerateNum() + "+" + terms[1].GenerateNum() +
"*e^(" + terms[3].GenerateNum() + "*x)";
    //        }
    //        else
    //        {
    //            if(type == 2)
    //            {
    //                // y = [ax^3 + bx^2 + cx + d] + e^(kx)
    //                text = "e^" + terms[1].GenerateText() + "x + " +
p.GenerateString();
    //            }
    //            else
    //            {
    //                //type == 3
    //                // y = [ax^3 + bx^2 + cx + d] + ke^(kx)
    //                //terms[0] = e, terms[1] = k
    //                text = terms[1].GenerateText() + "e^(" +
terms[1].GenerateText() + "x) + " + p.GenerateString();
    //            }
    //        }
    //        return text;
    //    }
    //    public double SolveModel(double x)
    //    {
    //        if(type == 1)
    //        {
    //            // y = a + b * ( c ^ [k*x] )
    //            //terms 0 = a, terms 1 = b, terms 2 = e, terms 3 = k
    //            return terms[0].GenerateNum() + terms[1].GenerateNum() *
Math.Pow(2.718, terms[3].GenerateNum() * x);
    //        }
    //        else if(type == 2)
    //        {
    //            // y = [ax^3 + bx^2 + cx + d] + e^(kx)
    //            //terms[0] = e, terms[1] = k
    //            return p.SolvePolynomial(x) + Math.Pow(2.718,
terms[1].GenerateNum() * x);
    //        }
    //        else if(type == 3)
    //        {
    //            // y = [ax^3 + bx^2 + cx + d] + ke^(kx)
    //            //terms[0] = e, terms[1] = k
    //            return p.SolvePolynomial(x) + Math.Pow(2.718 *
terms[1].GenerateNum(), terms[1].GenerateNum() * x);
    //        }
    //        else
    //        {
    //            return 0;
    //        }
    //    }
    //    public void CreateModelWithCubicAndE()
    //    {
    //        Clear();
    //        Random rnd = new Random();
    //        p = new Polynomial();
    //        p.SetTerms(new List<int>() { 1, rnd.Next(1, 4), rnd.Next(5, 13),
rnd.Next(4, 9) });
    //        // y = [ax^3 + bx^2 + cx + d] + e^(kx)
    //        //terms.Add(new Term(p, "", false));
    //        //term below = k
    //        terms.Add(new Term(Math.E, "", false));
    //        terms.Add(new Term(rnd.Next(1, 4), "", false));
    //        type = 2;
    //    }
    //    public void CreateModelWithE(int[] bounds)
    //    {
    //        // y = a + b * ( c ^ [k*x] )
    //        //has a size of 4 --> a, b, c and k; x is a variable to be solved.
    //        Clear();
    //        Random rnd = new Random();
    //        terms.Add(new Term(rnd.Next(bounds[0], bounds[1]), "", false));
    //        terms.Add(new Term(rnd.Next(bounds[0], bounds[1]), "", false));
    //        terms.Add(new Term(Math.E, "", false));
    //        terms.Add(new Term(rnd.Next(2, 4), "", false));
    //        type = 1;
    //    }
    //    public Model Differentiate()
    //    {
    //        Model m = new Model();
    //        if(type == 1)
    //        {
    //            m.type = 1;
    //            //y = 0 + [b*k]*e ^ [kx]
    //            m.terms.Add(new Term(0, "", false));
    //            m.terms.Add(new Term(terms[1].GenerateNum() *
terms[3].GenerateNum(), "", false));
    //            m.terms.Add(new Term(Math.E, "", false));
    //            m.terms.Add(new Term(terms[3].GenerateNum(), "", false));
    //        }
    //        else if(type == 2)
    //        {
    //            // y = [ax^3 + bx^2 + cx + d] + e^(kx)
    //            // y = 0x^3 + ax^2 + bx + ke^kx
    //            m.type = 3;
    //            m.p = p.Differentiate();
    //            Term t = new Term(terms[0], "e", true);
    //            t.AddChild(new Term(1, "", false));
    //            m.terms.Clear();
    //            m.terms.Add(t);
    //            m.terms.Add(new Term(terms[0], "", false));
    //        }
    //        return m;
    //    }
    //    //public string Differentiate()
    //    //{
    //    //    string text = "";
    //    //    if(type == 1)
    //    //    {
    //    //        text = Convert.ToString(terms[1].GenerateNum() *
terms[2].GenerateNum());
    //    //        if (terms[2].GenerateNum() == Math.E)
    //    //        {
    //    //            text += "e";
    //    //        }
    //    //        else
    //    //        {
    //    //            text += "Ln(" + terms[2] + ") * " + terms[2];
    //    //        }
    //    //        text += "^(" + terms[3] + "x)";
    //    //    }
    //    //    else if(type == 2)
    //    //    {
    //    //        text = terms[1].GenerateNum() * terms[2].GenerateNum() + "e^"
+ terms[2].GenerateText() + "x + " + p.Differentiate().GenerateString();
    //    //    }
    //    //    return text;
    //    //}
    //    public KeyValuePair<double, string> Differentiate(double x)
    //    {
    //        string val = "";
    //        double key;
    //        if (type == 1)
    //        {
    //            // y = a + b*c ^ [k*x] )
    //            val = Convert.ToString(terms[1].GenerateNum() *
terms[2].GenerateNum());
    //            if (terms[2].GenerateNum() == Math.E)
    //            {
    //                val += "e";
    //            }
    //            else
    //            {
    //                val += "Ln(" + terms[2] + ") * " + terms[2];
    //            }
    //            val += "^(" + terms[3] + "x)";
    //            //string to be returned
    //        }
    //        if (x == 999)
    //        {

    //            //don't need to return a number for the solved equation
    //            //assign 999 to the key to show it has no x value given and thus
cannot be differentiated definitely
    //            key = 999;
    //        }
    //        else
    //        {
    //            //x has been given - find derivative AND solve it
    //            key = Convert.ToDouble(terms[1].GenerateNum() *
terms[2].GenerateNum()) *
Math.Pow(Math.Log(Convert.ToDouble(terms[2].GenerateNum())), x *
Convert.ToDouble(terms[3].GenerateNum()));
    //        }
    //        return new KeyValuePair<double, string>(key, val);
    //    }
    //}
    class Binomial : Polynomial
    {
        int exponent = 1;
        public int GetExponent()
        {
            return exponent;
        }
        public int DetermineCoefficientOfSpecificTerm(int powerOfX)
        {
            if (terms.Count != 2)
            {
                return 0;
            }
            else
            {
                return NCR(exponent, powerOfX) * 
Convert.ToInt32(Math.Pow(terms[1], exponent - powerOfX) * Math.Pow(terms[0], 
powerOfX));
            }
        }
        public string ShowFactorisedBinomial()
        {
            return "(" + terms[1] + "x + " + terms[0] + ") ^ " + exponent;
        }
        public override void SetTerms(List<int> values)
        {
            if(values.Count == 2)
            {
                terms = values;
            }
        }
        public override void AddTerm(int x)
        {
            if (terms.Count < 2)
            {
                terms.Add(x);
            }
        }
        public void SetExponent(int exp)
        {
            exponent = exp;
        }
        public Polynomial ReturnExpansionAsPolynomial()
        {
            Polynomial expansion = new Polynomial();
            Stack<int> stack = new Stack<int>();
            //the use of the stack allows me to keep the same algorithm for 
generating the coefficients and then just pop from the stack to get them in the 
right order for the list of a polynomial
            //the algorithm finds coefficients of terms with descending powers of 
x however adding these straight to the list for polynomials won't work based on 
how my polynomial class is parsed
            //the LIFO structure for stacks means the order of coefficients can be 
added to the list reversed which is perfect
            int a = terms[1];
            int b = terms[0];
            for (int i = 0; i < exponent + 1; i++)
            {
                stack.Push(NCR(exponent, i) * Convert.ToInt32(Math.Pow(a, exponent 
- i) * Math.Pow(b, i)));
            }
            foreach(int j in stack)
            {
                expansion.AddTerm(stack.Pop());
            }
            return expansion;
        }
        public List<string> ReturnExpansionAsStringList()
        {
            List<string> expansion = new List<string>();
            int a = terms[1];
            int b = terms[0];
            for (int i = 0; i < exponent + 1; i++)
            {
                string xTerm;

                if (i == exponent - 1)
                {
                    xTerm = "x";
                }
                else
                {
                    if (i == exponent)
                    {
                        xTerm = "";
                    }
                    else
                    {
                        xTerm = "x^" + Convert.ToString(exponent - i);
                    }
                }
                expansion.Add(Convert.ToString(NCR(exponent, i) * Math.Pow(a,
exponent - i) * Math.Pow(b, i)) + xTerm);
            }
            return expansion;
        }
        public string ReturnExpansionAsString()
        {
            string text = "";
            int a = terms[1];
            int b = terms[0];
            for (int i = 0; i < exponent + 1; i++)
            {
                string xTerm;

                if (i == exponent - 1)
                {
                    xTerm = "x";
                }
                else
                {
                    if (i == exponent)
                    {
                        xTerm = "";
                    }
                    else
                    {
                        xTerm = "x^" + Convert.ToString(exponent - i);
                    }
                }
                text += Convert.ToString(NCR(exponent, i) * Math.Pow(a, exponent - 
i) * Math.Pow(b, i)) + xTerm + "\n";
            }
            return text;
        }
    }
    class Polynomial : MathsMethods
    {
        bool factorisedQuadratic = false;
        string[] factorsOfQuadratic = { "", "" };
        bool logApplied = false;
        //bool hiddenPolynomial = false;
        //Term hide;
        //public void HidePolynomial(Term operationToX)
        //{
        //    factorisedQuadratic = false;
        //    bool? validOperator = null;
        //    Term t = operationToX;
        //    while (validOperator == null)
        //    {
```

```
        //        if (t.isJustX == true)
        //        {
        //            validOperator = true;
        //        }
        //        else
        //        {
        //            try
        //            {
        //                t = t.GetChild();
        //            }
        //            catch
        //            {
        //                //t has no child nor are any of its parents equal to x
        //                validOperator = false;
        //            }


        //        }
        //    }
        //    if (validOperator == true)
        //    {
        //        hide = operationToX;
        //    }
        //}
        public double FindDefiniteIntegral(double lower, double upper)
        {
            return TrapeziumRule(lower, upper, 1000000);
        }
        public double TrapeziumRule(double lowerBound, double upperBound, int 
strips)
        {
            double height = (upperBound - lowerBound) / strips;
            double integral = 0;
            for(double i = lowerBound + height; i < upperBound; i = Sum(i, 
height))
            {
                integral += SolvePolynomial(i);
            }
            integral *= 2;
            integral += (SolvePolynomial(lowerBound) +
SolvePolynomial(upperBound));
            integral *= (height / 2);
            return integral;
        }
        protected List<int> terms = new List<int>();
        public void ApplyLog()
        {
            logApplied = true;
        }
        public void RemoveLog()
        {
            logApplied = false;
        }
        public virtual void SetTerms(List<int> values)
        {
            factorisedQuadratic = false;
            terms = values;
        }
        public virtual void AddTerm(int x)
        {
            factorisedQuadratic = false;
            terms.Add(x);
        }
        public List<int> GetTerms()
        {
            return terms;
        }
        public void MultiplyPolynomialByInt(int j)
        {
            factorisedQuadratic = false;
```

```
    for (int i = 0; i < terms.Count; i++)
    {
        terms[i] *= j;
    }
}
public Polynomial MultiplyPolynomial(int j)
{
    Polynomial p = new Polynomial();
    p.SetTerms(terms);
    factorisedQuadratic = false;
    for (int i = 0; i < terms.Count; i++)
    {
        terms[i] *= j;
    }
    return p;
}
public Polynomial SumPolynomial(Polynomial p)
{
    Polynomial returnP = new Polynomial();
    int loop = p.GetTerms().Count;
    if (loop > terms.Count)
    {
        loop = terms.Count;
    }
    for(int i = 0; i < loop; i++)
    {
        returnP.AddTerm(p.GetTerms()[i] + terms[i]);
    }
    return returnP;
}
public int GetDiscriminantFromQuadratic()
{
    if (terms.Count == 3 && terms[terms.Count-1] != 0)
    {
        return terms[1] * terms[1] - (4 * terms[2] * terms[0]);
    }
    else
    {
        return 99999999;
    }
}
public void CreateQuadraticWithOneRoot()
{
    factorisedQuadratic = false;
    //discriminant = 0
    //b^2  -  4ac  = 0
    //b^2 = 4ac
    Random rnd = new Random();
    int b = 2 * rnd.Next(1, 5);
    int a = 0;
    int c = 0;
    while(b*b != 4 * a * c)
    {
        a = rnd.Next(2, 10);
        c = rnd.Next(2, 10);
    }
    SetTerms(new List<int>() { c, b, a });
}
public double FindCorrectRoot()
{
    double root = 9999;
    double counter = -10;
    while(root == 9999)
    {
        root = NewtonsMethod(counter);
        counter += 0.1;
    }
    return root;
```

```
        }
        public void GenerateFactorisableQuadratic()
        {
            //finds a quadratic that can be factorised as (ax + b)(cx + d)
            //expanded quadratic should be (ac)x^2 + (bc + ad)x + (bd)
            Random rnd = new Random();
            int a = rnd.Next(1, 3);
            int b = rnd.Next(0, 10);
            int c = rnd.Next(1, 3);
            int d = rnd.Next(0, 10);
            SetTerms(new List<int>() { b * d, b * c + a * d, a * c });
            factorsOfQuadratic[0] = "(" + a + "x + " + b + ")(" + c + "x + " + d +
")";
            factorsOfQuadratic[1] = "(" + c + "x + " + d + ")(" + a + "x + " + b +
")";
            factorisedQuadratic = true;
        }
        public string[] ShowFactorsOfQuadratic()
        {
            if(factorisedQuadratic == true)
            {
                return factorsOfQuadratic;
            }
            else
            {
                return new string[] { "", "" };
            }
        }
        public void GenerateRandomPolynomial(int highestPower, int lowerBound, int
higherBound)
        {
            factorisedQuadratic = false;
            Random rnd = new Random();
            terms.Clear();
            for(int i = 0; i < highestPower+1; i++)
            {
                terms.Add(rnd.Next(lowerBound, higherBound));
            }
        }
        public string GenerateString()
        {
            string x = "x";
            if(logApplied == true)
            {
                x = "[log(x)]";
            }
            string text = "";
            for (int i = terms.Count - 1; i > -1; i--)
            {
                if (terms[i] != 0)
                {
                    if (i != terms.Count - 1)
                    {
                        if (terms[i] > 0)
                        {
                            text += "+";
                        }
                    }
                    text += terms[i];
                    if (i < 2)
                    {
                        if (i == 1)
                        {
                            text += x;
                        }
                    }
                    else
                    {
                        text += x + "^" + i;
                    }
                }


            }
            return text;
        }
        public double[] FindClosestStationaryPoint(double xApproximation)
        {
            if (CheckIfStationaryPointsExist())
            {
                //stationary pooints exist for the polynomial
                double[] point = new double[] { 0, 0 };
                point[0] = Differentiate().NewtonsMethod(xApproximation);
                if (point[0] == 9999)
                {
                    point[0] = Differentiate().FindCorrectRoot();
                }
                point[1] = SolvePolynomial(point[0]);
                return point;
            }
            else
            {
                return new double[] { 9999, 9999 };
            }
        }
        public int CheckStatusOfPoint(double[] stationaryPoint)
        {
            double x;
            x =
Differentiate().Differentiate().SolvePolynomial(stationaryPoint[0]);
            if (x == 0)
            {
                return 0;
                //not a turning point --> gradient = 0
            }
            else
            {
                if (x > 0)
                {
                    return 1;
                    //is a minimum turning point --> gradient > 0 --> code returns
1
                }
                else
                {
                    return -1;
                    //is a maximum turning point --> gradient < 0 --> code returns
-1
                }
            }

        }
        public bool CheckIfStationaryPointsExist()
        {
            int termsOfDerivative = Differentiate().GetTerms().Count;
            if (termsOfDerivative % 2 == 1)
            {
                //polynomial's derivative's highest power is even -> if this has
no real roots then there are no stationary points
                if(termsOfDerivative == 3)
                {
                    //quadratic - check if at least one of its roots has only real
components
                    if(Differentiate().SolveQuadratic()[0].imaginary == 0 ||
Differentiate().SolveQuadratic()[1].imaginary == 0)
                    {
                        //quadratic has a real solution -> polynomial has a
stationary point
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                }
                else
                {
                    //the derivative is a quartic or similar -> apply Newton's
Method to try and find a root
                    if(Differentiate().FindCorrectRoot() != 9999)
                    {
                        //correct root has been found -> stationary points exist
                        return true;
                    }
                    else
                    {
                        //no correct root could be identified so no stationary
points can be found
                        return false;
                    }
                }
            }
            else
            {
                //derivative of the polynomial must cross the x axis due to having
an odd power for its highest power, meaning the polynomial has a stationary point
                return true;
            }
        }
        public double SolvePolynomial(double x)
        {
            //finds the solution to the polynomial when x is a particular value
            double solution = 0;
            for (int i = 0; i < terms.Count; i++)
            {
                solution += terms[i] * Math.Pow(x, i);
            }
            return solution;
        }
        public int newtonCounter = 0;
        public double NewtonsMethod(double x)
        {
            newtonCounter++;
            if (Math.Round(SolvePolynomial(x), 4) == 0)
            {
                newtonCounter = 0;
                return x;
            }
            else
            {
                if(newtonCounter < 30)
                {
                    return NewtonsMethod(x - SolvePolynomial(x) /
Differentiate().SolvePolynomial(x));
                }
                else
                {
                    newtonCounter = 0;
                    return 9999;
                }
            }
        }
        public Polynomial Differentiate()
        {
            Polynomial derivative = new Polynomial();
            for (int i = 0; i < terms.Count; i++)
            {
                if (i == 0)
                {
                }
                else
                {
                    if (terms[i] != 0)
                    {
                        derivative.AddTerm(i * terms[i]);
                    }
                    else
                    {
                        derivative.AddTerm(0);
                    }
                }
            }
            return derivative;
        }
        public List<string> FindRootsOfPolynomialAsStrings()
        {
            List<string> strings = new List<string>();
            List<Complex> roots = FindRootsOfPolynomial();
            foreach(Complex root in roots)
            {
                strings.Add(root.GenerateString());
            }
            return strings;
        }
        public List<Complex> FindRootsOfPolynomial()
        {
            List<Complex> solutions = new List<Complex>();
            if (terms.Count == 3 && terms[2] != 0)
            {
                Complex[] quadRoots = new Complex[] { new Complex(0,0), new
Complex(0,0) };
                quadRoots = SolveQuadratic();
                foreach(Complex root in quadRoots)
                {
                    solutions.Add(root);
                }
            }
            else
            {
                if(terms.Count > 3)
                {
                    //randomise approximation start-points for Newton's Method
                    //Run the algorithm a few times to try and find several roots
                    for (int i = 0; i < 10; i++)
                    {
                        double approximation = i - 5;
                        Complex x = new Complex(NewtonsMethod(approximation), 0);
                        bool alreadyFound = false;
                        foreach (Complex solution in solutions)
                        {
                            x.real = Math.Round(x.real, 5);
                            solution.real = Math.Round(solution.real, 5);
                            if (x.real == solution.real)
                            {
                                alreadyFound = true;
                            }
                        }
                        if (!alreadyFound)
                        {
                            solutions.Add(x);
                        }
                    }
                }
            }
            return solutions;
        }
```

```
        public Complex[] SolveQuadratic()
        {
            Complex[] solutions = new Complex[] { new Complex(0, 0), new
Complex(0, 0) };
            if(terms.Count == 3 && terms[2] != 0)
            {
                double a = terms[2];
                double b = terms[1];
                double c = terms[0];
                double root;
                if (!((b * b) - (4 * a * c) < 0))
                {
                    root = Math.Sqrt((b * b) - (4 * a * c));
                    solutions[0].real = (-b + root) / (2 * a);
                    solutions[1].real = (-b - root) / (2 * a);
                }
                else
                {
                    root = Math.Sqrt(Math.Abs((b * b) - (4 * a * c)));
                    solutions[0].real = (-b) / 2;
                    solutions[0].imaginary = root / 2;
                    solutions[1].real = (-b) / 2;
                    solutions[1].imaginary = (-root) / 2;
                }
            }
            return solutions;
        }
    }
    public class Complex
    {
        public Complex(double rComponent, double iComponent) { real = rComponent;
imaginary = iComponent; }
        public double real; public double imaginary;
        public double ConvertToDouble()
        {
            if(imaginary == 0){ return real; }

            else return 9999999999999999;

        }

        public void Add(Complex num) { real += num.real; imaginary +=
num.imaginary; }

        public void Sub(Complex num) { real -= num.real; imaginary -=
num.imaginary; }

        public void MultiplyByComplex(Complex complex) { real = (real *
complex.real) + (imaginary * complex.imaginary);

            imaginary = (real * complex.imaginary) + (imaginary * complex.real);

        }

        public void DivideByComplex(Complex complex) {  real = (real /
complex.real) + (imaginary / complex.imaginary);

            imaginary = (real / complex.imaginary) + (imaginary / complex.real);

        }

        public void MultiplyByReal(double multiplicand)

        {

            real *= multiplicand;

            imaginary *= multiplicand;

        }

        public void DivideByReal(double divisor)

        {

            real /= divisor;

            imaginary /= divisor;

        }

        public string GenerateString()

        {

            if(imaginary > 0)

            {

                //2 + 3i or 2 + 0i

                return real + " + " + imaginary + "i";

            }

            else if(imaginary == 0)

            {

                return Convert.ToString(real);
            }
            else return real + " - " + Math.Abs(imaginary) + "i";
        }
    }
}
```

Binary tree class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using HtmlAgilityPack;
using System.Windows.Forms;

namespace PolynomialQuiz
{
    class BinaryTree
    {
    }
    class Node
    {
        public Node()
        {

        }
        public Node(int val)
        {
            data = val;
        }
        private int data = -9999999;
        private Node left;
        private Node right;
        //each node has data as well as two child nodes: the one directly to its 
left and the one directly to its right.

        //the node class contains methods for getting and setting these 
attributes, which are utilised by the tree class.

        public int GetData()

        {

            return data;

        }

        public Node GetLeftNode()

        {

            return left;

        }

        public Node GetRightNode()

        {

            return right;

        }

        public void SetRightNode(Node node)

        {

            right = node;

        }

        public void SetLeftNode(Node node)

        {

            left = node;

        }

        public void SetData(int val)

        {

            data = val;

        }

    }

    class Tree

    {

        public string ShowNodes(List<int> nodes)

        {

            string text = "{ ";

            foreach(int i in nodes)
            {
                text += i + ", ";
            }
            text += "}";
            return text;
        }
        public List<int> GenerateRandomTree(Node root, int numberOfNodesToInsert,
int lowerBound, int upperBound)
        {
            List<int> nodes = new List<int>();
            Random rnd = new Random();
            if(numberOfNodesToInsert > 0)
            {
                for (int i = 0; i < numberOfNodesToInsert; i++)
                {
                    int num = rnd.Next(lowerBound, upperBound);
                    if(root.GetData() == -9999999)
                    {
                        root.SetData(num);
                    }
                    else
                    {
                        root = InsertNode(root, num);
                    }

                    nodes.Add(num);
                }
            }
            return nodes;
        }
        public void PreOrderTraversal(Node root, List<int> nums)
        {
            nums.Add(root.GetData());
            if (root.GetLeftNode() != null)
```

```
    {
        PreOrderTraversal(root.GetLeftNode(), nums);
    }
    if (root.GetRightNode() != null)
    {
        PreOrderTraversal(root.GetRightNode(), nums);
    }
}
public void InOrderTraversal(Node root, List<int> nums)
{

    if (root.GetLeftNode() != null)
    {
        PreOrderTraversal(root.GetLeftNode(), nums);
    }
    nums.Add(root.GetData());
    if (root.GetRightNode() != null)
    {
        PreOrderTraversal(root.GetRightNode(), nums);
    }
}
public void PostOrderTraversal(Node root, List<int> nums)
{

    if (root.GetLeftNode() != null)
    {
        PreOrderTraversal(root.GetLeftNode(), nums);
    }

    if (root.GetRightNode() != null)
    {
        PreOrderTraversal(root.GetRightNode(), nums);
    }
    nums.Add(root.GetData());
```

```
        }
        public void PreOrderTraversal(Node root, Control areaToWrite)
        {
            areaToWrite.Text += Convert.ToString(root.GetData()) +
Environment.NewLine;
            if (root.GetLeftNode() != null)
            {
                PreOrderTraversal(root.GetLeftNode(), areaToWrite);
            }
            if (root.GetRightNode() != null)
            {
                PreOrderTraversal(root.GetRightNode(), areaToWrite);
            }
        }
        public void InOrderTraversal(Node root, Control areaToWrite)
        {
            if (root.GetLeftNode() != null)
            {
                InOrderTraversal(root.GetLeftNode(), areaToWrite);
            }
            areaToWrite.Text += Convert.ToString(root.GetData()) +
Environment.NewLine;
            if (root.GetRightNode() != null)
            {
                InOrderTraversal(root.GetRightNode(), areaToWrite);
            }
        }
        public void PostOrderTraversal(Node root, Control areaToWrite)
        {
            if (root.GetLeftNode() != null)
            {
                PostOrderTraversal(root.GetLeftNode(), areaToWrite);
            }
            if (root.GetRightNode() != null)
            {
                PostOrderTraversal(root.GetRightNode(), areaToWrite);

            }

            areaToWrite.Text += Convert.ToString(root.GetData()) +
Environment.NewLine;

        }

        public Node InsertNode(Node root, int v)

        {

            /*

            * The InsertNode algorithm is recursive and keeps cycling through
nodes in the tree until it finds one with the left or right child of it empty.

             * At this point, the node is assigned to be the located node's child,
using setters and getters to access node attributes and modify them.

             */

            if (root == null)

            {

                /*

                 * The constructor for node doesn't need any parameters as these
are assigned when new nodes are made and added to the tree.

                 * The left and right attributes need to be left null in order for
the algorithm to easily locate available positions in the tree.

                */

                root = new Node();

                root.SetData(v);

            }

            else if (v < root.GetData())

            {

                root.SetLeftNode(InsertNode(root.GetLeftNode(), v));

            }

            else

            {

                root.SetRightNode(InsertNode(root.GetRightNode(), v));

            }

            return root;

        }

    }

    class Date
```

```
{
    private int year;
    private int month;
    private int day;
    private string USdate;
    private string UKdate;
    private bool leapYear;
    private bool validDateComplete = false;
    public Date()
    {

    }
    public Date(int day, int month, int year)
    {
        SetDate(day, month, year);
    }
    public bool GetValidity()
    {
        return validDateComplete;
    }
    public int GetYear()
    {
        return year;
    }
    public int GetMonth()
    {
        return month;
    }
    public int GetDay()
    {
        return day;
    }
    public string GetUSdate()
    {
            return USdate;
        }
        public string GetUKdate()
        {
            return UKdate;
        }
        public Date GenerateRandomDate()
        {
            //generates a random date to be used to create questions
            Random rnd = new Random();
            Date date = new Date();
            while (!date.GetValidity())
            {
                date.SetDate(rnd.Next(1, 32), rnd.Next(1, 13), rnd.Next(1900, 
2021));
            }
            return date;
        }
        public Date SetDate(int inputDay, int inputMonth, int inputYear)
        {
            int dayLimit = 0;
            /*
             * Calculate if the year is a leap year based on the following rules:
             * Every year that is exactly divisible by four is a leap year,
             * except for years that are exactly divisible by 100, but these 
centurial years are leap years if they are exactly divisible by 400.
             * For example, the years 1700, 1800, and 1900 are not leap years, but 
the years 1600 and 2000 are.
            */
            if((inputYear % 400 == 0) || (inputYear % 100 != 0 && inputYear % 4 == 
0))
            {
                leapYear = true;
            }
            else
            {
                leapYear = false;
            }
            //Figure out whether the days entered is a valid date based on the
month and whether it is a leap year or not
            if (inputMonth < 13 && inputMonth > 0)
            {
                switch (inputMonth)
                {
                    case 1:
                        dayLimit = 31;
                        break;
                    case 2:
                        if (leapYear)
                        {
                            dayLimit = 29;
                        }
                        else
                        {
                            dayLimit = 28;
                        }
                        break;
                    case 3:
                        dayLimit = 31;
                        break;
                    case 4:
                        dayLimit = 30;
                        break;
                    case 5:
                        dayLimit = 31;
                        break;
                    case 6:
                        dayLimit = 30;
                        break;
                    case 7:
                        dayLimit = 31;
                        break;
                    case 8:
                        dayLimit = 31;
                        break;
                    case 9:
                        dayLimit = 30;
                        break;
                    case 10:
                        dayLimit = 31;
                        break;
                    case 11:
                        dayLimit = 30;
                        break;
                    case 12:
                        dayLimit = 31;
                        break;
                }
                //Check that the day entered is valid for the month
                if (inputDay > 0 && inputDay <= dayLimit)
                {
                    day = inputDay;
                    month = inputMonth;
                    year = inputYear;
                    USdate = Convert.ToString(month) + "/" + Convert.ToString(day)
+ "/" + Convert.ToString(year);
                    UKdate = Convert.ToString(day) + "/" + Convert.ToString(month)
+ "/" + Convert.ToString(year);
                    validDateComplete = true;
                }
                else
                {
                    validDateComplete = false;
                }
```

```
            }
            else
            {
                validDateComplete = false;
            }
            if (validDateComplete)
            {
                return this;
            }
            else
            {
                return null;
            }
        }
        public bool CompareDates(Date d2)
        {
            //determine if a date is earlier than another date
            bool earlier = false;
            if (GetYear() < d2.GetYear())
            {
                earlier = true;
            }
            else
            {
                if (GetYear() == d2.GetYear())
                {
                    if (GetMonth() < d2.GetMonth())
                    {
                        earlier = true;
                    }
                    else
                    {
                        //find out if the month is equal and if not, go down to
the day
```

```
                        if (GetMonth() == d2.GetMonth())
                        {
                            if (GetDay() < d2.GetDay())
                            {
                                earlier = true;
                            }
                        }
                    }
                }
            }
            return earlier;
        }
    }
    class PersonNode
    {
        public PersonNode()   {  }
        public PersonNode(string name, Date date) => person = new
KeyValuePair<string, Date>(name, date);
        private KeyValuePair<string, Date> person = new KeyValuePair<string,
Date>("", new Date().SetDate(1, 1, 1));
        //key is person's name, value is their date of birth
        private PersonNode left;
        private PersonNode right;
        //each node has data as well as two child nodes: the one directly to its
left and the one directly to its right.
        //the node class contains methods for getting and setting these
attributes, which are utilised by the tree class.
        public KeyValuePair<string, Date> GetData() => person;
        public KeyValuePair<string, Date> GetKVP() => person;
        public string GetName() => person.Key;
        public string GetDate() => person.Value.GetUKdate();
        public string GenerateString() => GetName() + ": " + GetDate();
        public PersonNode GetLeftNode() => left;
        public PersonNode GetRightNode() => right;
        public void SetRightNode(PersonNode node) => right = node;
        public void SetLeftNode(PersonNode node) => left = node;

        public void SetData(string name, Date birth) => new PersonNode(name,
birth);

    }

    class DictionaryTree

    {

        public PersonNode root = new PersonNode();

        Dictionary<string, Date> nodes = new Dictionary<string, Date>();

        public Dictionary<string, Date> GetNodesInOrderOfInsertionIntoTree()

        {

            return nodes;

        }

        public void GenerateQuizReadyTree()

        {

            if (treeTypeSet == false)

            {


            }

            else

            {

                //GenerateQuizReadyTree(int traversalType)

                HtmlWeb web = new HtmlAgilityPack.HtmlWeb();

                HtmlAgilityPack.HtmlDocument doc =
web.Load("https://www.verywellfamily.com/top-1000-baby-boy-names-2757618");
//website with boys' names rated

                string name = "";

                Random rndName = new Random();

                for (int i = 0; i < 6; i++)

                {

                    foreach (var item in
doc.DocumentNode.SelectNodes("/html/body/main/article/div[2]/ol/li[" +
rndName.Next(1, 1001) + "]"))

                    {

                        //webscraping is used to get 10 names from a list of 1000
names in a site to assign to people to insert into the tree

                        name = item.InnerText + " ";
                    }

                    Date date = new Date().GenerateRandomDate();

                    MessageBox.Show("Person " + Convert.ToString(i+1) + " added to
the binary tree is " + name + ", born on " + date.GetUKdate());

                    nodes.Add(name, date);

                    if (root.GetDate() == new Date(1,1,1).GetUKdate())

                    {

                        root.SetData(name, date);

                    }

                    else

                    {

                        root = InsertNode(root, name, date);

                    }

                }

            }

        }

        private int alphabeticalOrAge;

        private bool treeTypeSet = false;

        //alphabeticalOrAge basically determines whether nodes should be added to
the tree based on a person's name or their age

        //Once the method that will be used for inserting nodes is determined, it
cannot be changed, due to the boolean treeTypeSet

        public void SetTreeType(int val)

        {

            if (!treeTypeSet)

            {

                alphabeticalOrAge = val;

                treeTypeSet = true;

            }

            //if treeTypeSet isn't true then the type of tree to build can be set

            //if val is 0, the tree is built based on name; if it is 1, it is
built based on age

        }

        public Stack<KeyValuePair<string, Date>> GetStackFromTraversal(byte b,
PersonNode root)

        {
            List<KeyValuePair<string, Date>> people = new
List<KeyValuePair<string, Date>>();

            if (b == 0)

            {

                //pre order

                PreOrderTraversal(root, people);

            }

            else

            {

                if (b == 1)

                {

                    //in order

                    InOrderTraversal(root, people);

                }

                else

                {

                    //post order

                    PostOrderTraversal(root, people);

                }

            }

            Stack<KeyValuePair<string, Date>> stack = new
Stack<KeyValuePair<string, Date>>();

            foreach (KeyValuePair<string, Date> kvp in people)

            {

                stack.Push(kvp);

            }

            return stack;

        }

        public Queue<KeyValuePair<string, Date>> GetQueueFromTraversal(byte b,
PersonNode root)

        {

            List<KeyValuePair<string, Date>> people = new
List<KeyValuePair<string, Date>>();

            if (b == 0)

            {
                //pre order
                PreOrderTraversal(root, people);
            }
            else
            {
                if (b == 1)
                {
                    //in order
                    InOrderTraversal(root, people);
                }
                else
                {
                    //post order
                    PostOrderTraversal(root, people);
                }
            }
            Queue<KeyValuePair<string, Date>> q = new Queue<KeyValuePair<string,
Date>>();
            foreach(var item in people)
            {
                q.Enqueue(item);
            }
            return q;
        }
        public void PreOrderTraversal(PersonNode root, List<KeyValuePair<string,
Date>> people)
        {
            //format the dictionary
            //foreach (KeyValuePair<string, Date> kvp in root.GetData())
            //{
            //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";
            //}
            people.Add(root.GetKVP());
            if (root.GetLeftNode() != null)
            {
```

```
            PostOrderTraversal(root.GetLeftNode(), people);
        }

        if (root.GetRightNode() != null)
        {
            PostOrderTraversal(root.GetRightNode(), people);
        }
    }
    //public List<PersonNode> PreOrderTraversal(PersonNode root)
    //{
    //    List<PersonNode> output = new List<PersonNode>();
    //    //format the dictionary
    //    //foreach (KeyValuePair<string, Date> kvp in root.GetData())
    //    //{
    //    //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";
    //    //}
    //    //Console.WriteLine(output);
    //    output.Add(root);
    //    if (root.GetLeftNode() != null)
    //    {
    //        PreOrderTraversal(root.GetLeftNode());
    //    }
    //    if (root.GetRightNode() != null)
    //    {
    //        PreOrderTraversal(root.GetRightNode());
    //    }
    //    return output;
    //}
    //public List<PersonNode> InOrderTraversal(PersonNode root)
    //{
    //    List<PersonNode> output = new List<PersonNode>();
    //    //format the dictionary
    //    //foreach (KeyValuePair<string, Date> kvp in root.GetData())
    //    //{
        //    //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";
        //    //}
        //    if (root.GetLeftNode() != null)
        //    {
        //        InOrderTraversal(root.GetLeftNode());
        //    }
        //    output.Add(root);
        //    if (root.GetRightNode() != null)
        //    {
        //        InOrderTraversal(root.GetRightNode());
        //    }
        //    return output;
        //}
        public void InOrderTraversal(PersonNode root, List<KeyValuePair<string,
Date>> people)
        {
            //format the dictionary
            //foreach (KeyValuePair<string, Date> kvp in root.GetData())
            //{
            //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";
            //}
            if (root.GetLeftNode() != null)
            {
                PostOrderTraversal(root.GetLeftNode(), people);
            }
            people.Add(root.GetKVP());
            if (root.GetRightNode() != null)
            {
                PostOrderTraversal(root.GetRightNode(), people);
            }
        }
        public void PostOrderTraversal(PersonNode root, List<KeyValuePair<string,
Date>> people)
        {
            //format the dictionary
            //foreach (KeyValuePair<string, Date> kvp in root.GetData())

            //{

            //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " } ";

            //}

            if (root.GetLeftNode() != null)

            {

                PostOrderTraversal(root.GetLeftNode(), people);

            }

            if (root.GetRightNode() != null)

            {

                PostOrderTraversal(root.GetRightNode(), people);

            }

            people.Add(root.GetKVP());

        }

        public PersonNode InsertNode(PersonNode leaf, string name, Date date)

        {

            /*

            * The InsertNode algorithm is recursive and keeps cycling through
nodes in the tree until it finds one with the left or right child of it empty.

             * At this point, the node is assigned to be the located node's child,
using setters and getters to access node attributes and modify them.

             */

            if (treeTypeSet)

            {

                if (leaf == null)

                {

                    /*

                     * The constructor for node doesn't need any parameters as
these are assigned when new nodes are made and added to the tree.

                     * The left and right attributes need to be left null in order
for the algorithm to easily locate available positions in the tree.

                    */

                    leaf = new PersonNode(name, date);

                    //string output = "";

                    //foreach (KeyValuePair<string, Date> kvp in root.GetData())
                    //{
                    //    output = kvp.Key + " { " + kvp.Value.GetUKdate() + " }
";
                    //}
                }
                else
                {
                    //leaf.SetData(name, date);
                    if (alphabeticalOrAge == 0)
                    {
                        #region alphabetical
                        //build tree based on name --> not done, probably won't
use
                        string nameOfPerson = leaf.GetName();
                        byte[] asciiOfInsertionNode =
Encoding.ASCII.GetBytes(name);
                        byte[] asciiOfCurrentNode =
Encoding.ASCII.GetBytes(nameOfPerson);
                        if (asciiOfCurrentNode.Length >
asciiOfInsertionNode.Length)
                        {
                            //do a for loop which is capped at the shorter name's
length
                            foreach (char character in asciiOfInsertionNode)
                            {
                                //if(character >)
                            }
                        }
                        else
                        {


                        }
                        #endregion
                    }
                    else
                    {
                        #region numeric
                        //build tree based on age
                        //get the date in the dictionary
                        Date dateOfRoot = new Date();
                        dateOfRoot = leaf.GetData().Value;
                        if (date.CompareDates(dateOfRoot))
                        {
                            //the person to be inserted is older, so moves left
                            leaf.SetLeftNode(InsertNode(leaf.GetLeftNode(), name,
date));
                        }
                        else
                        {
                            //the person to be inserted is younger, so moves right
                            leaf.SetRightNode(InsertNode(leaf.GetRightNode(),
name, date));
                        }
                        #endregion
                    }

                }
                return leaf;
            }
            else
            {
                return null;
            }
        }
    }
}
```

Form2 aka Computing quiz

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using HtmlAgilityPack;

namespace PolynomialQuiz
{

    public partial class Form2 : Form
    {
        User user;
        DictionaryTree tree = new DictionaryTree();
        Tree numericTree = new Tree();
        //PersonNode root;
        TrueFalse Q1 = new TrueFalse();
        TrueFalse Q6 = new TrueFalse();
        TextBoxQuestion Q2 = new TextBoxQuestion();
        TextBoxQuestion Q3 = new TextBoxQuestion();
        TextBoxQuestion Q4 = new TextBoxQuestion();
        TextBoxQuestion Q5 = new TextBoxQuestion();
        TextBoxQuestion Q7 = new TextBoxQuestion();
        List<CompQuestion> questions = new List<CompQuestion>();
        List<KeyValuePair<string, Date>> people = new List<KeyValuePair<string,
Date>>();
        List<KeyValuePair<string, Date>> traversal = new List<KeyValuePair<string,
Date>>();
        List<int> numTreeNodes;
        List<CheckBox> markScheme;
        List<string> answers = new List<string>() { "", "", "", "", "", "", "" };
        public Form2(User player)
        {
            //the data structures quiz has no difficulty variation
            InitializeComponent();
            markScheme = new List<CheckBox>(){ cb1, cb2, cb3, cb4, cb5, cb6, cb7
};
            user = player;
        }

        private void Form2_Load(object sender, EventArgs e)
        {

            questions = new List<CompQuestion>() { Q1, Q2, Q3, Q4, Q5, Q6, Q7 };
            Random rnd = new Random();
            tree.SetTreeType(1);
            //tree is instantiated based on age
            tree.GenerateQuizReadyTree();
            foreach (KeyValuePair<string, Date> kvp in
tree.GetNodesInOrderOfInsertionIntoTree())
            {
                txtNodes.Text += kvp.Key + " " + kvp.Value.GetUKdate() +
Environment.NewLine;
                //people.Add(p);
                //shows the nodes in the randomly generated tree in order of
insertion
                people.Add(kvp);
            }
            Node node = new Node();
            numTreeNodes = numericTree.GenerateRandomTree(node, 10, 1, 30);

            //q1 = name of the first person returned from a pre-order traversal
            answers[0] = people[rnd.Next(0, people.Count - 1)].Key;
            Q1.SetUpQuestion(new Control[] { rbQ1True, rbQ1False }, cb1, lblQ1,
"True or false: the name of the first person returned from a pre-order traversal
is " + answers[0]);
            answers[0] = "true";

            //q2 = name of the last person returned from a post-order traversal
            Q2.SetUpQuestion(new TextBox[] {txtQ2}, cb2, lblQ2, "Type the birthday
of the last person to be returned from a post-order traversal in the format
dd/mm/yyyy unless the day or month only has one digit in which case d/mm/yyyy |
dd/m/yyyy | d/m/yyyy");

            tree.PostOrderTraversal(tree.root, traversal);

            answers[1] = traversal[traversal.Count - 1].Value.GetUKdate();


            //q3 = name of the second oldest person

            Q3.SetUpQuestion(new TextBox[] { txtQ3 }, cb3, lblQ3, "Who is the
second person returned from an inorder traversal?");

            tree.InOrderTraversal(tree.root, traversal);

            answers[2] = traversal[1].Key;


            //q4 = name of the youngest person

            Q4.SetUpQuestion(new TextBox[] { txtQ4 }, cb4, lblQ4, "What's the name
of the last person returned from an inorder traversal?");

            answers[3] = tree.GetStackFromTraversal(1, tree.root).Pop().Key;


            //q5 = find dot product of two randomly generated vectors

            Vector U = new Vector(0, 0).GenerateRandomVector(2, 10);

            Vector V = new Vector(0, 0).GenerateRandomVector(1, 5);

            Q5.SetUpQuestion(new TextBox[] { txtQ5 }, cb5, lblQ5, "What is the dot
product of " + U.GenerateString() + " and " + V.GenerateString() + " ?");

            answers[4] = U.DotProduct(V.GetXY()).ToString();


            //q6 = true or false about a numeric binary tree

            Node root = new Node();

            numTreeNodes = numericTree.GenerateRandomTree(root, 10, -20, 20);

            List<int> nodes = new List<int>();

            numericTree.PreOrderTraversal(root, nodes);

            lblQ6.Text = numericTree.ShowNodes(nodes);

            Q6.SetUpQuestion(new RadioButton[] { rbQ6True, rbQ6False }, cb6,
lblQ6, lblQ6.Text + " True or false: in the tree with the nodes shown to the left
inserted into it, " + numTreeNodes[4] + " will be the third number returned from a
pre-order traversal.");

            if(nodes[2] == numTreeNodes[4])

            {
                answers[5] = "true";
            }
            else
            {
                answers[5] = "false";
            }

            //q7 - angle of vectors to the nearest degree
            U.GenerateRandomVector(3, 8);
            V.GenerateRandomVector(1, 6);
            Q7.SetUpQuestion(new TextBox[] { txtQ7 }, cb7, lblQ7, "What is the
angle formed between the following vectors? " + U.GenerateString() + " and " +
V.GenerateString());
            answers[6] = Math.Round(U.CalcVecAng(V.GetXY())).ToString();
        }

        private void btnSubmit_Click(object sender, EventArgs e)
        {
            //validate that all questions have been answered
            //interface is used to check if each question has been answered
through the method QuestionAnswered() which varies depending on the question type
            bool canBeMarked = true;
            foreach(CompQuestion q in questions)
            {
                if (!q.QuestionAnswered())
                {
                    canBeMarked = false;
                }
            }
            if(canBeMarked == true)
            {
                int score = 0;
                for (int i = 0; i < questions.Count; i++)
                {
                    if (questions[i].MarkQuestion(answers[i]))
                    {
                        //correct
                        questions[i].GetCheckBox().Checked = true;
                        score++;
                    }
                    else
                    {
                        //incorrect
                        questions[i].GetCheckBox().Checked = false;
                    }
                }
                score *= (100 / 7);
                MessageBox.Show("Well done! You got " + Convert.ToString(score) +
"%");
                user.SaveQuizScore(new Quiz(user.GetUserID(), "C", score));
            }
            else
            {
                MessageBox.Show("You haven't answered all the questions!");
            }
        }

        private void rbQ6False_CheckedChanged(object sender, EventArgs e)
        {

        }
    }
    public class CompQuestion
    {
        public bool MarkQuestion(string answer)
        {
            if(answer == Convert.ToString(GetAnswer()))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        Label lblQuestion;
        CheckBox cb;
        public void SetUpQuestion(Control[] inputs, CheckBox cb, Label lbl, string
textForLabel)
        {
            SetLabel(lbl);
            SetTextOfLabel(textForLabel);
            SetCheckBox(cb);
            AssignInputs(inputs);
        }
        public virtual void AssignInputs(Control[] inputs)
        {

        }
        public void SetLabel(Label lbl) => lblQuestion = lbl;
        public void SetCheckBox(CheckBox checkbox) => cb = checkbox;
        public CheckBox GetCheckBox() => cb;
        public void SetTextOfLabel(string text) => lblQuestion.Text = text;
        public virtual Object GetAnswer()
        {
            return "";
        }
        public virtual bool QuestionAnswered()
        {
            if (GetAnswer() == "")
            {
                return false;
            }
            else
            {
                return true;
            }
        }
    }
    public class Vector{
        public Vector(double x, double y) {
            XY.Clear();
            XY.Add(x);
            XY.Add(y);  }
        Random rnd = new Random();
        List<double> XY = new List<double>() { };
        public void SetXY(double x, double y) {
            XY.Clear();
            XY.Add(x);
            XY.Add(y);  }
        public string GenerateString(){ return "{ " + XY[0] + ", " + XY[1] + " }";
        }
        public Vector GenerateRandomVector(){
            SetXY(rnd.NextDouble(), rnd.NextDouble());
            return this;  }
        public Vector GenerateRandomVector(int lowerBound, int upperBound) {
            SetXY(rnd.Next(lowerBound, upperBound), rnd.Next(lowerBound,
upperBound));
            return this; }
        public List<double> GetXY()   { return XY;  }
        public double GetX()  {
            try {    return XY[0]; }
            catch {  return 0;  } }
        public double GetY()  {
            try
            { return XY[1]; }
            catch
            { return 0; }  }
        public double CalcVecAng(List<double> vectorV)
        {
            //Multiply the vectors
            double UxV = DotProduct(vectorV);
            double magU = Math.Sqrt(DotProduct(XY));
            double magV = Math.Sqrt(DotProduct(vectorV));
            double angle = Math.Acos(UxV / (magU * magV) * (Math.PI / 180));
            return angle;
        }
        public double DotProduct(List<double> vectorV)  {
            double dotProduct = 0;
            for (int i = 0; i < XY.Count; i++)
            {
                dotProduct += XY[i] * vectorV[i];
            }
            return dotProduct;  } }
    public class TextBoxQuestion : CompQuestion, IQuestionAnswered
    {
        TextBox txt;
        public override void AssignInputs(Control[] inputs)
        {
            if(inputs.Length == 1)
            {
                if(inputs[0] is TextBox)
                {
                    SetTextBox((TextBox)inputs[0]);
                }
            }
        }
        public void SetTextBox(TextBox textbox)
        {
            txt = textbox;
        }
        public string GetText()
        {
            return txt.Text;
        }
        public override Object GetAnswer()
        {
            if(GetText() == "")
            {
                return "";
            }
            return GetText();
        }
    }
    public static class TextBoxExtension
    {
        //allows textboxes to have an extended method
        public static bool QuestionAnswered(this TextBox txt)
        {
            if (txt.Text != "")
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
    class TrueFalse : CompQuestion, IQuestionAnswered
    {
        RadioButton rbTrue;
        RadioButton rbFalse;
        public override void AssignInputs(Control[] inputs)
        {
```

```
    if (inputs.Length == 2)

    {

        if (inputs[0] is RadioButton && inputs[1] is RadioButton)

        {

            SetButtons((RadioButton)inputs[0], (RadioButton)inputs[1]);

        }

    }

}

public void SetButtons(RadioButton rb1, RadioButton rb2)

{

    rbTrue = rb1;

    rbFalse = rb2;

}

public override Object GetAnswer()

{

    if (QuestionAnswered())

    {

        if (rbTrue.Checked)

        {

            return "true";

        }

        else

        {

            return "false";

        }

    }

    else

    {

        return "";

    }

}

public override bool QuestionAnswered()

{

    if(!rbTrue.Checked && !rbFalse.Checked)
```

```
            {
                return false;
            }
            else
            {
                return true;
            }
        }
    }
    ////class Matrix
    //{
    //    //not completed
    //    private List<List<double>> table = new List<List<double>>();
    //    //each list represents a new row
    //    //the size of a single list represents the number of columns
    //    private int[] type = { 0, 0 };
    //    //type[0] is the number of rows of the matrix and type[1] is the number
of columns
    //    public List<List<double>> GetTable()
    //    {
    //        return table;
    //    }
    //    public int[] GetType()
    //    {
    //        return type;
    //    }
    //    public Matrix MultiplyByMatrix(Matrix m)
    //    {
    //        //this function assumes the matrix being called is the first
multiplicand and that the parameter of the function is the second,
    //        //e.g. if this was 2x3 and the parameter was 3x2, the result would
be a 2x3 x 3x2 --> returns a 2x2 matrix
    //        //in matrix multiplication, the number of rows of the first matrix
must equal the number of columns of the second matrix
    //        if (type[0] == m.GetType()[1])
    //        {
    //            //matrix multiplication can occur due to the rule mentioned
above
    //            Matrix result = new Matrix();
    //            //result will have the same number of rows as the first matrix
and the same number of columns as the second matrix
    //            result.DefineBounds(type[0], m.GetType()[1]);
    //            //now we need to multiply the matrices

    //            for (int i = 0; i < type[0]; i++)
    //            {
    //                List<double> row = new List<double>();
    //                for (int j = 0; j < m.GetType()[1]; j++)
    //                {
    //                    row.Add(table[i][j] * m.GetTable()[i][j]);
    //                }
    //                result.InsertRow(row);
    //            }
    //            return result;
    //        }
    //        else
    //        {
    //            //rows of first matrix != columns of second matrix -->
multiplication can't occur
    //            return null;
    //        }
    //    }
    //    public void ScalarOperation(int scalar, char operation)
    //    {
    //        char[] validOperators = { '+', '-', '*', '/', '%' };
    //        bool validInput = false;
    //        foreach (char op in validOperators)
    //        {
    //            if (operation == op)
    //            {
```

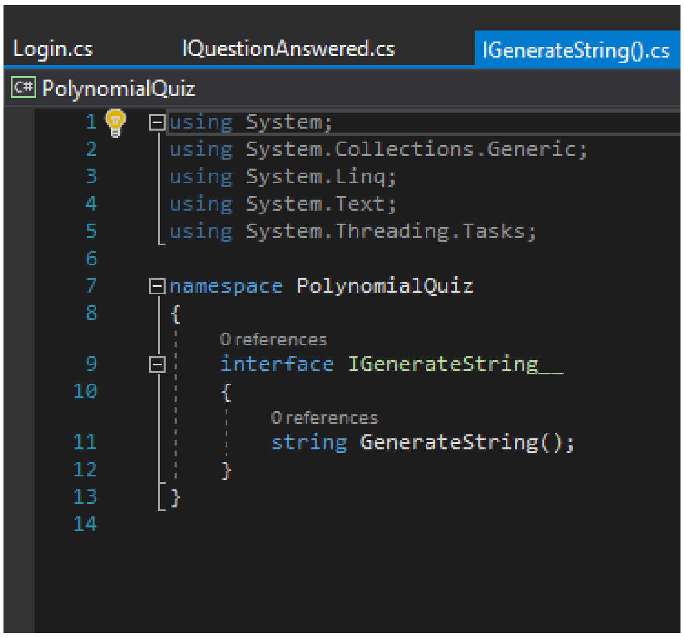
```
//                  validInput = true;
//              }
//          }
//          if (validInput == true)
//          {
//              foreach (List<double> row in table)
//              {
//                  for (int i = 0; i < row.Count; i++)
//                  {
//                      switch (operation)
//                      {
//                          case '+':
//                              row[i] += scalar;
//                              break;
//                          case '-':
//                              row[i] -= scalar;
//                              break;
//                          case '*':
//                              row[i] *= scalar;
//                              break;
//                          case '/':
//                              row[i] /= scalar;
//                              break;
//                          case '%':
//                              row[i] %= scalar;
//                              break;
//                      }
//                  }
//              }
//          }
//      }
//      public void Debug()
//      {
//          Console.WriteLine(type[0] + "x" + type[1] + "|" + table.Count);
```

```
    //    }
    //    public void DefineBounds(int rows, int columns)
    //    {
    //        table.Clear();
    //        type[0] = rows;
    //        type[1] = columns;
    //    }
    //    public void InsertRow(List<double> row)
    //    {
    //        if (table.Count < type[0])
    //        {
    //            if (row.Count == type[1])
    //            {
    //                table.Add(row);
    //            }
    //        }
    //    }
    //    public string PrintMatrix()
    //    {
    //        string text = "";
    //        foreach (List<double> row in table)
    //        {
    //            foreach (int num in row)
    //            {
    //                text += num.ToString() + ", ";
    //            }
    //            text += "\n";
    //        }
    //        return text;
    //    }
    //}
}
```

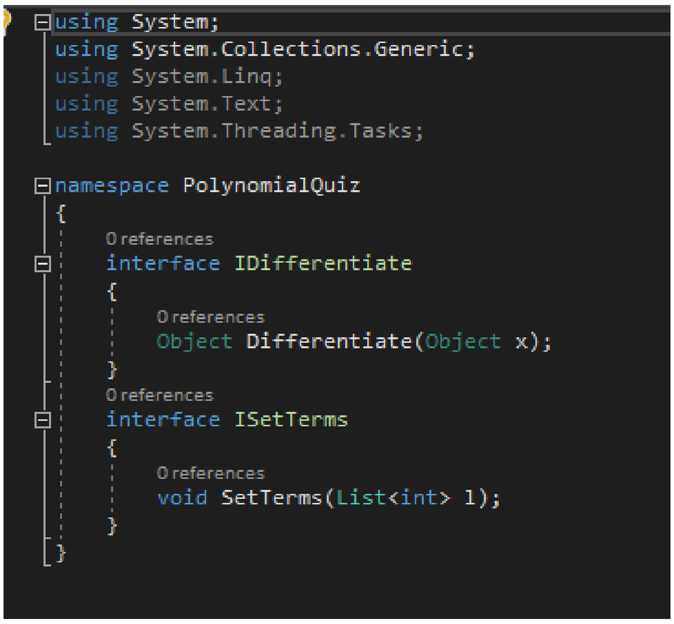
Form3 aka Tree Builder

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PolynomialQuiz
{
    public partial class Form3 : Form
    {
        Tree tree = new Tree();
        Node root = new Node();
        public Form3()
        {
            InitializeComponent();
        }
        private void btnExit_Click(object sender, EventArgs e)
        {
            this.Close();
        }
        private void btnAddNodeToTree_Click(object sender, EventArgs e)
        {
            int numToAdd;
            try
            {
                numToAdd = Convert.ToInt32(txtEntry.Text);
            }
            catch
            {
                numToAdd = -999999999;
            }
            if(numToAdd != -999999999)
            {
                if(root.GetData() == -9999999)
                {
                    root.SetData(numToAdd);
                }
                else
                {
                    root = tree.InsertNode(root, numToAdd);
                }
                txtNodesInOrderOfInsertion.Text += numToAdd + Environment.NewLine;
                txtEntry.Text = "";
            }
        }

        private void btnClearTree_Click(object sender, EventArgs e)
        {
            root = new Node();
            txtNodesInOrderOfInsertion.Text = "";
            txtNodesFromTraversal.Text = "";
        }

        private void btnPreOrder_Click(object sender, EventArgs e)
        {
            if(root.GetData() != -9999999)
            {
                txtNodesFromTraversal.Text = "";
                tree.PreOrderTraversal(root, txtNodesFromTraversal);
            }
        }

        private void btnInOrder_Click(object sender, EventArgs e)
        {
            if (root.GetData() != -9999999)
            {
                txtNodesFromTraversal.Text = "";
                tree.InOrderTraversal(root, txtNodesFromTraversal);
            }
        }

        private void btnPostOrder_Click(object sender, EventArgs e)
        {
            if (root.GetData() != -9999999)
            {
                txtNodesFromTraversal.Text = "";
                tree.PostOrderTraversal(root, txtNodesFromTraversal);
            }
        }
    }
}
```

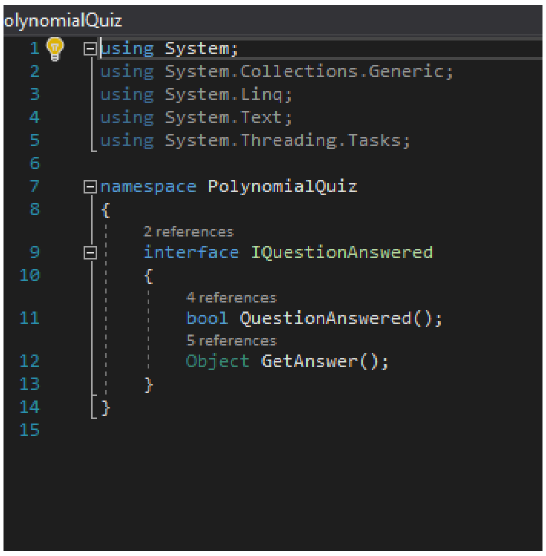
Interfaces



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PolynomialQuiz
{
    interface IGenerateString__
    {
        string GenerateString();
    }
}
```



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PolynomialQuiz
{
    0 references
    interface IDifferentiate
    {
        0 references
        Object Differentiate(Object x);
    }
    0 references
    interface ISetTerms
    {
        0 references
        void SetTerms(List<int> l);
    }
}
```



olynomialQuiz

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PolynomialQuiz
{
    2 references
    interface IQuestionAnswered
    {
        4 references
        bool QuestionAnswered();
        5 references
        Object GetAnswer();
    }
}
```

Login

```
using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Data.OleDb;
```

```
namespace PolynomialQuiz
{
    public partial class Login : Form
    {
        OleDbConnection conn = new
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source = Project.mdb");
        public Login()
        {
            InitializeComponent();
        }
        private void btnCreateNewUser_Click(object sender, EventArgs e){
            if (txtPassword.Text.Length >= 8 && txtUsername.Text != "")
            {
                //check that the chosen username doesn't already exist
                User newUser = new User(txtUsername.Text, txtPassword.Text);
                string sql = "Select * from tblUser where Username = @Username";
                OleDbConnection conn = newUser.GetConn();
                conn.Open();
                OleDbCommand checkIfUserExists = new OleDbCommand(sql, conn);
                checkIfUserExists.Parameters.AddWithValue("@Username",
newUser.GetUsername());
                OleDbDataReader qryReader = checkIfUserExists.ExecuteReader();
                bool canUseThisName = true;
                while (qryReader.Read())
                {
                    try
                    {
                        qryReader.GetInt32(0);
                        canUseThisName = false;
                        //tries to get the first user name value with the given
value
                        //if it can get it, it exists so a user can't be created
with this username
                    }
                    catch
                    {
                        canUseThisName = true;
                        //username not found in the database - it can be used
                    }
                }
                conn.Close();
                if (canUseThisName == true)
                {
                    //check that the password is at least 8 characters long
                    if (newUser.GetPassword().Length >= 8)
                    {
                        //password is valid
                        conn.Open();
                        sql = "Select MAX (UserID) from tblUser";
                        int uid = Convert.ToInt32(new OleDbCommand(sql,
conn).ExecuteScalar()) + 1;
                        sql = "INSERT INTO tblUser VALUES (@UserID, @Username,
@Password)";
                        OleDbCommand createUser = new OleDbCommand(sql, conn);
                        createUser.Parameters.AddWithValue("@UserID", uid);
                        createUser.Parameters.AddWithValue("@Username",
newUser.GetUsername());
                        createUser.Parameters.AddWithValue("@Password",
newUser.GetPassword());
                        createUser.ExecuteNonQuery();
                        conn.Close();
                        MessageBox.Show("New user successfully created!");
                        LogIn();
                    }
                    else
                    {
                        lblStatus.Text = "Passwords for new users must have at
least 8 characters.";
                    }
                }
                else
                {
                   lblStatus.Text = "That username already exists! Pick another to
create a new account.";
                }
            } }
        private void LogIn()
        {
            if (txtUsername.Text != "" && txtPassword.Text != "")
            {
                User u = new User();
                u.LoadUser(txtUsername.Text);
                if (u.ComparePassword(txtPassword.Text))
                {
                    //login successful
                    new Menu(u).Show();
                }
                else
                {
                    //login unsuccessful
                    lblStatus.Text = "Incorrect credentials!";
                }
                conn.Close();
            }
        }
        private void btnLogin_Click(object sender, EventArgs e)
        {
            LogIn();
        }
    }
    public class Quiz
    {
        private int QuizID;
        private int UserID;
        private string QuizType = "";

        private int Score;

        public string DisplayAsString(User user)

        {

            string quiztype = "";

            switch (QuizType)

            {

                case "C":

                    quiztype = "Computing";

                    break;

                case "ME":

                    quiztype = "Maths (easy)";

                    break;

                case "MH":

                    quiztype = "Maths (hard)";

                    break;

            }

            return "Quiz ID : " + QuizID + " | User : " + user.GetUsername() + " |
" + "Quiz Type : "

                + quiztype + " | Score as percentage : " + Score;

        }

        public Quiz(){}

        public Quiz(int uid, string qt, int s)

        {

            UserID = uid;

            QuizType = qt;

            Score = s;

        }

        public Quiz(int qid, int uid, string qt, int s)

        {

            SetData(qid, uid, qt, s);

        }

        public int GetUserID()

        {
```

```
        return UserID;
    }
    public int GetQuizID()
    {
        return QuizID;
    }
    public int GetScore()
    {
        return Score;
    }
    public string GetType()
    {
        if (QuizType == "C" || QuizType == "ME" || QuizType == "MH")
        {
            //computing quiz = C
            //easy maths quiz = ME
            //hard maths quiz = MH
            return QuizType;
        }
        else
        {
            return "";
        }
    }
    public void SetData(int qID, int uID, string qT, int sc)
    {
        QuizID = qID;
        UserID = uID;
        QuizType = qT;
        Score = sc;
    }

}
public class User
```

```
    {
        private int UserID;
        private string Username;
        private string Password;
        private Dictionary<int, Quiz> QuizScores = new Dictionary<int, Quiz>();
        bool UserLoaded = false;
        OleDbConnection conn = new
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source = Project.mdb");
        public User()
        {
        }
        public User(string name, string pass)
        {
            Username = name;
            Password = pass;
        }
        public OleDbConnection GetConn()
        {
            return conn;
        }
        public string GetPassword()
        {
            return Password;
        }
        public void SaveQuizScore(Quiz quiz)
        {
            conn.Open();
            string sql = "Select MAX (QuizID) from tblQuiz";
            int qid = Convert.ToInt32(new OleDbCommand(sql, conn).ExecuteScalar())
+ 1;
            sql = "INSERT INTO tblQuiz VALUES (@QuizID, @UserID, @QuizType,
@Score)";
            OleDbCommand storeQuiz = new OleDbCommand(sql, conn);
            storeQuiz.Parameters.AddWithValue("@QuizID", qid);
            storeQuiz.Parameters.AddWithValue("@UserID", quiz.GetUserID());
            storeQuiz.Parameters.AddWithValue("@QuizType", quiz.GetType());
            storeQuiz.Parameters.AddWithValue("@Score", quiz.GetScore());
            storeQuiz.ExecuteNonQuery();
            conn.Close();
        }
        public int GetUserID()
        {
            return UserID;
        }
        public string GetUsername()
        {
            return Username;
        }
        public bool ComparePassword(string attempt)
        {
            if (attempt == Password)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        public void SetData(int UID, string name, string Pass)
        {
            //Dictionary<int, Quiz> quizzes
            UserID = UID;
            Username = name;
            Password = Pass;
            //QuizScores = quizzes;
        }
        public List<Quiz> GetQuizzesByType(string type)
        {
            List<Quiz> QuizzesByType = new List<Quiz>(){ };
            LoadQuizzes();
            if(type == "C" || type == "ME" || type == "MH")
            {
                foreach(KeyValuePair<int, Quiz> kvp in QuizScores)
                {
                    if(kvp.Value.GetType() == type)
                    {
                        QuizzesByType.Add(kvp.Value);
                    }
                }
            }
            return QuizzesByType;
        }
        public Quiz GetQuizByQuizID(int ID)
        {
            LoadQuizzes();
            try
            {
                return QuizScores[ID];
            }
            catch
            {
                return null;
            }
        }
        public List<Quiz> RankQuizzes()
        {
            List<Quiz> quizzes = new List<Quiz>() { };
            conn.Open();
            string sql = "Select * from tblQuiz where UserID = @UserID order by
Score";
            OleDbCommand loadQuizzes = new OleDbCommand(sql, conn);
            loadQuizzes.Parameters.AddWithValue("@UserID", UserID);
            OleDbDataReader qryReader = loadQuizzes.ExecuteReader();
            while (qryReader.Read())
            {
                Quiz quiz = new Quiz();
                quiz.SetData(qryReader.GetInt32(0), qryReader.GetInt32(1),
qryReader.GetString(2), qryReader.GetInt32(3));
                quizzes.Add(quiz);
            }
            conn.Close();
            return quizzes;
        }
        public void LoadQuizzes()
        {
            if (UserLoaded == true)
            {
                conn.Open();
                QuizScores.Clear();
                string sql = "Select * from tblQuiz where UserID = @UserID";
                OleDbCommand loadQuizzes = new OleDbCommand(sql, conn);
                loadQuizzes.Parameters.AddWithValue("@UserID", UserID);
                OleDbDataReader qryReader = loadQuizzes.ExecuteReader();
                while (qryReader.Read())
                {
                    Quiz quiz = new Quiz();
                    quiz.SetData(qryReader.GetInt32(0), qryReader.GetInt32(1),
qryReader.GetString(2), qryReader.GetInt32(3));
                    QuizScores.Add(quiz.GetQuizID(), quiz);
                    //MessageBox.Show(quiz.DisplayAsString(this));
                }
                conn.Close();
            }
        }
        public void LoadUser(string user)
        {
            conn.Open();
            string sql = "Select * from tblUser";
            OleDbCommand loadUser = new OleDbCommand(sql + " where Username = 
@Username", conn);
            loadUser.Parameters.AddWithValue("@Username", user);
            OleDbDataReader qryReader = loadUser.ExecuteReader();
            while (qryReader.Read())
            {
                SetData(qryReader.GetInt32(0), qryReader.GetString(1), 
qryReader.GetString(2));
            }
            UserLoaded = true;
            conn.Close();
        }
    }
}
```

Maths methods

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PolynomialQuiz
{
    abstract class MathsMethods
    {
        protected double Sum(double a, double b)
        {
            return a + b;
        }
        protected int NCR(int N, int R)
        {
            return Factorial(N) / (Factorial(R) * Factorial(N - R));
        }
```

```
protected double Sec(double num)
{
    return 1 / Math.Cos(num);
}
protected double Csc(double num)
{
    return 1 / Math.Sin(num);
}
protected double Cot(double num)
{
    return 1 / Math.Tan(num);
}
protected double AngleConversion(double angle, bool type)
{
    if(type == true)
    {
        //convert from radians to degrees
        return angle / (Math.PI / 180);
    }
    else
    {
        //convert from degrees to radians
        return angle * (Math.PI / 180);
    }
}
protected int Factorial(int value)
{
    if (value == 0)
    {
        return 1;
    }
    else
    {
        return value * Factorial(value - 1);
```

```
            }
        }
    }
}
```

Tutorial1 aka tutorial on polynomials

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PolynomialQuiz
{
    public partial class Tutorial1 : Form
    {
        Polynomial p;
        public Tutorial1()
        {
            InitializeComponent();
            p = new Polynomial();
            p.GenerateFactorisableQuadratic();
            lblText1.Text = "Polynomials are sequences with multiple terms of x in
them. Quadratics are a very common type of polynomial, which contain a number of
x^2, such as " + p.GenerateString() + Environment.NewLine + "Polynomials are used
in various places in the real world, including modelling economic markets,
particle motion, determining optimal proportions of materials to use to minimise
cost and maximise efficiency in engineering, creating precise lighting in computer
graphics and various aspects of chemistry." + Environment.NewLine + "Being able to
create polynomials and manipulate them to achieve desired outcomes is very useful,
for example, finding the exact point where a particle stops in relation to time,
or dimensions for a drinks container to minimise costs." + Environment.NewLine +
"To do these, you need to be able to solve polynomials, aka, find points where x
is 0. With quadratics, this is easy as sometimes they can be factorised, like with
" + p.GenerateString() + "where it has two brackets in a form like (ax + b) * (cx
+ d). Other quadratics require the quadratic formula to solve: x = { -b (+ or -)
square root [b^2  - 4*a*c]  } / (2*a) " + Environment.NewLine + "Other polynomial
types don't have simple formulas to use, such as cubics and quartics which do have
formulas but tend to be long winded to find roots with." + Environment.NewLine +
"A straight forward method of finding roots to polynomials where the highest power
is greater than 2 is to use the Newton-Raphson Method." + Environment.NewLine +
"The following link gives an in-depth explanation of how it works -->
https://youtu.be/-RdOwhmqP5s?t=271" + Environment.NewLine + " but it can be
effectively summarised as" + Environment.NewLine + "x(n+1) = x(n) - P(x) / P'(x) -
-> you start with a guess of what the root may be; from here you, put that into
the polynomial; next, you have to find the derivative of the polynomial (the
derivative means the rate of change; it is explained further on what a derivative
is);" + Environment.NewLine + "you then plug in your guess for what x could be
into the derivative like you did for the polynomial; next, you divide the
polynomial by its derivative and then subtract this number from your guess for x;
you keep iterating this process and eventually your approximation for the root is
refined." + Environment.NewLine + "Calculating derivatives is important as they
show the rate at which something changes, for example, a model for a meteor's
velocity can be differentiated with respect to time to find its acceleration at a
given time." + Environment.NewLine + "Derivatives can also be used to optimise
modelling, for example, finding where the rate at which money is spent on
materials for a car to be made is at its minimum by finding where the rate of
change is 0 for a formula that describes the car's costs and relation to
materials." + Environment.NewLine + "The formula for finding a polynomials
derivative is to look at each term individually: the derivative for a term that is
a * x^n  is a * n * x^(n-1). For example, in " + p.GenerateString() + ", the
derivative would be " + p.Differentiate().GenerateString() + Environment.NewLine +
"   Terms where x is to the power of 1, i.e. ax, are just a as n is 1 and a * 1 =
a; for terms where x is to the power of 0 (so just a number like 12), the
derivative is 0 as 12 * 0 = 0;";

        }


        private void btnExitTutorial_Click(object sender, EventArgs e)

        {

            this.Close();

        }

    }

}
```

PolynomialQuiz - actually the form which allows you to interactively learn about polynomials by solving them, finding derivatives, etc.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PolynomialQuiz
{
    public partial class PolynomialQuiz : Form
    {
        Polynomial p;
        Binomial b;
        TextBox[] binomialInformation;
        public PolynomialQuiz()
        {
            InitializeComponent();
            p = new Polynomial();
            binomialInformation = new TextBox[] { txtBinomialA, txtBinomialB,
txtBinomialC };
            lblTerms.Location = new Point(40, 157);
        }
        private void btnFindRoots_Click(object sender, EventArgs e)
        {
            lbSolutions.DataSource = p.FindRootsOfPolynomialAsStrings();
            lblTerms.Text = "";
            p = new Polynomial();
        }
        private void btnAddTerm_Click(object sender, EventArgs e)
        {
            int coefficient;
            try
            {
                coefficient = Convert.ToInt32(txtTermEntry.Text);
            }
            catch
            {
                coefficient = 999999991;
            }
            if(coefficient != 999999991)
            {
                p.AddTerm(coefficient);
                lblTerms.Text = p.GenerateString();
                txtTermEntry.Text = "";
            }
        }
        private void btnTutorial_Click(object sender, EventArgs e)
        {
            new Tutorial1().Show();
        }
        private void btnFindDerivative_Click(object sender, EventArgs e)
        {
            lbSolutions.DataSource = new List<string>() {
p.Differentiate().GenerateString() };
            lblTerms.Text = "";
            p = new Polynomial();
        }
        private List<int> GetABN()
        {
            List<int> abn = new List<int>();
            bool expansionPossible = true;
            foreach (TextBox t in binomialInformation)
            {
                if (t.Text != "")
                {
```

```
            try
            {
                int i = Convert.ToInt32(t.Text);
                if (i > 0)
                {
                    abn.Add(i);
                }
                else
                {
                    expansionPossible = false;
                }
            }
            catch
            {
                expansionPossible = false;
            }
        }
        else
        {
            expansionPossible = false;
        }
    }
    if(expansionPossible == false)
    {
        return new List<int>(){ };
    }
    else
    {
        return abn;
    }
}
private void btnExpand_Click(object sender, EventArgs e)
{
    if (GetABN().Count != 0)
```

```
            {
                b = new Binomial();
                b.SetTerms(new List<int>() { GetABN()[0], GetABN()[1] });
                b.SetExponent(GetABN()[2]);
                //lbSolutions.DataSource = new List<string>() {
b.ReturnExpansionAsString() };
                lbSolutions.DataSource = b.ReturnExpansionAsStringList();
                b = new Binomial();
            }
        }

        private void btnIntegrate_Click(object sender, EventArgs e)
        {
            if(GetABN().Count != 0)
            {
                lbSolutions.DataSource = new List<string>() {
Convert.ToString(p.TrapeziumRule(GetABN()[0], GetABN()[1], GetABN()[2])) };
            }
        }
    }
}
```

Menu

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PolynomialQuiz
{
```

```
public partial class Menu : Form
{
    User user;
    public Menu(User player)
    {
        InitializeComponent();
        user = player;
    }

    private void btnCreateQuiz_Click(object sender, EventArgs e)
    {
        if(!rbDataStructures.Checked && !rbPureMaths.Checked)
        {
            MessageBox.Show("You haven't selected a quiz type to load!");
        }
        else
        {
            if (rbPureMaths.Checked)
            {
                //maths quiz gets generated
                new Form1(tbDifficulty.Value, user).Show();
            }
            else
            {
                //computer science data structures quiz gets generated
                new Form2(user).Show();
            }
        }

    }

    private void rbDataStructures_CheckedChanged(object sender, EventArgs e)
    {
        if (rbDataStructures.Checked)
        {
            tbDifficulty.Enabled = false;
        }
        else
        {
            tbDifficulty.Enabled = true;
        }
    }

    private void btnPolynomialCalculator_Click(object sender, EventArgs e)
    {
        new PolynomialQuiz().Show();
    }

    private void btnBuildTrees_Click(object sender, EventArgs e)
    {
        new Form3().Show();
    }

    private void btnViewPersonalBests_Click(object sender, EventArgs e)
    {
        List<string> PBs = new List<string>() { };
        foreach(Quiz q in user.RankQuizzes())
        {
            PBs.Add(q.DisplayAsString(user));
        }
        lbLeaderboard.DataSource = PBs;
    }

    private void btnGetQuizById_Click(object sender, EventArgs e)
    {
        try
        {
            Quiz q = user.GetQuizByQuizID(Convert.ToInt32(txtQuizInfo.Text));
```

```
                if(q != null)

                {

                    if (user.GetUserID() != q.GetUserID())

                    {

                        lbLeaderboard.DataSource = new string[] { "The quiz 
specified isn't one you have performed so cannot be viewed!"};

                    }

                    else

                    {

                        lbLeaderboard.DataSource = new string[] { 
user.GetQuizByQuizID(Convert.ToInt32(txtQuizInfo.Text)).DisplayAsString(user) };

                    }

                }

            }

            catch

            {

                lbLeaderboard.DataSource = new string[] { "You have to enter a 
valid numerical quiz ID!" };

            }

        }


        private void btnGetQuizByType_Click(object sender, EventArgs e)

        {

            lbLeaderboard.Text = "";

            List<string> quizzes = new List<string>(){ };

            foreach(Quiz q in user.GetQuizzesByType(txtQuizInfo.Text))

            {

                quizzes.Add(q.DisplayAsString(user));

            }

            lbLeaderboard.DataSource = quizzes;

        }

    }

}
```

Testing

Loading quizzes from the database

https://youtu.be/K_UfxARbc6w

Computing quiz

https://www.youtube.com/watch?v=AaRWtTxBPps

Polynomial Calculator

https://www.youtube.com/watch?v=FIcQWK8azXg

Easy Maths Quiz

https://www.youtube.com/watch?v=RYwSK82H2-Y

Hard Maths Quiz

https://www.youtube.com/watch?v=GjSUlrMXx7Y

Binary Tree Builder

https://www.youtube.com/watch?v=-ydPmaLnaoc

Login and Signup

https://www.youtube.com/watch?v=dmpqZa8UsuY

NOTE: These links may have changed or been removed