**Development**

Contents

[1. Libraries used – Abstraction 2](#_Toc35457003)

[2. Object-Oriented Programming (OOP) – Abstraction 6](#_Toc35457004)

[3. Encapsulation – Abstraction 9](#_Toc35457005)

[4. File Access 12](#_Toc35457006)

[5. Validation 14](#_Toc35457007)

[6. Error handling 16](#_Toc35457008)

[7. Static polymorphism 16](#_Toc35457009)

[8. Dynamic polymorphism 16](#_Toc35457010)

[9. Graphs 18](#_Toc35457011)

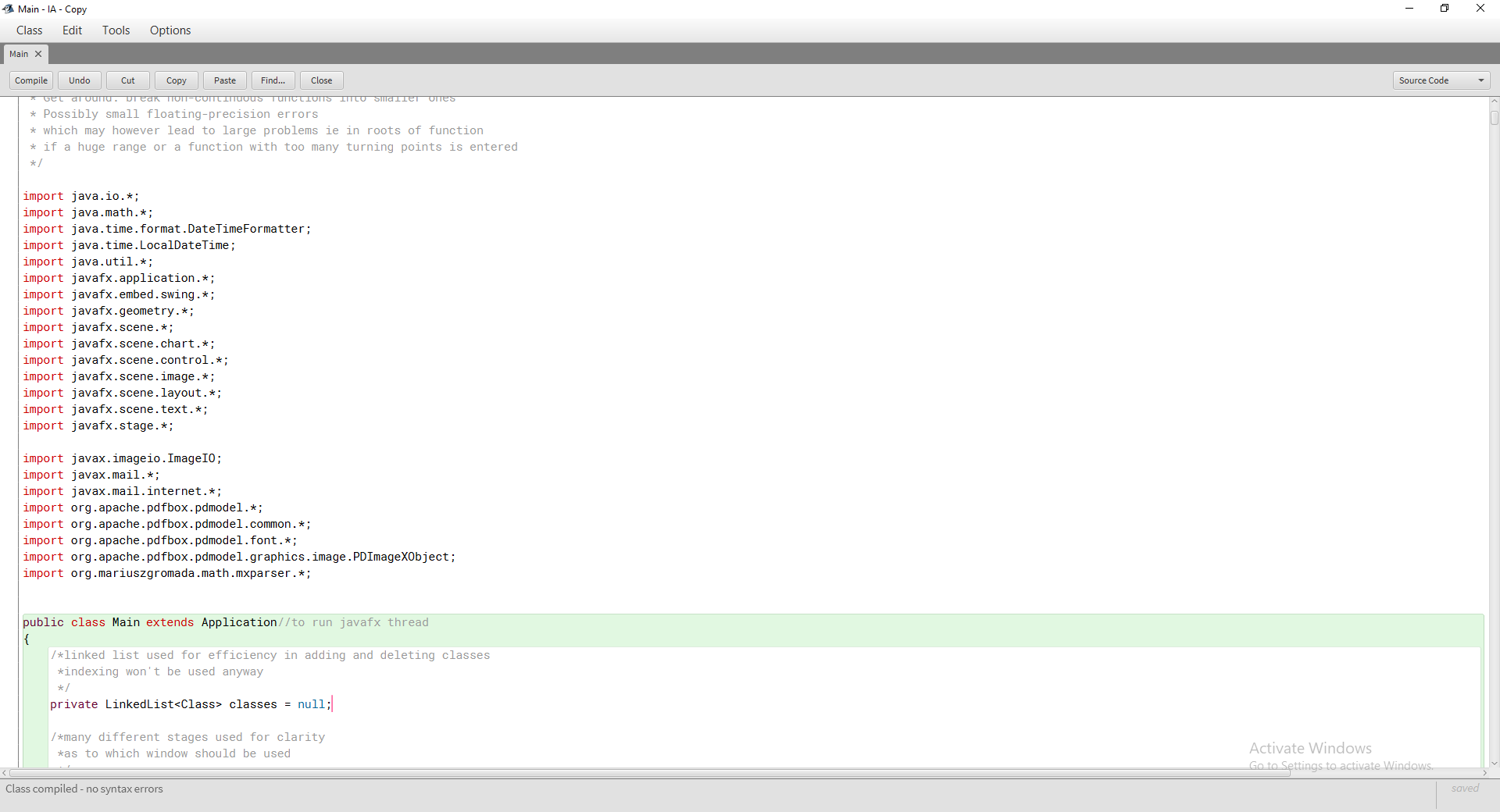
[10. Implementation of mathematical functions 20](#_Toc35457012)

[11. Inline events 29](#_Toc35457013)

[References 30](#_Toc35457014)

# Libraries used – Abstraction

Imported libraries play a major role in the program. They offer the chance to complete complicated tasks with simple commands, such as simple statements, function calls and variable definitions. They make coding easier and less time-consuming, since pre-existing code does not need to be written from scratch, as well as less error-prone and easier to debug, as the lines of code included in libraries have been tested by millions of users and achieve their aim beyond doubt. Finally, when using pre-compiled code, we do not need to know **how** each function works; abstraction exists.



Alphabetically sorted external libraries linked

Used to read and write from file

Used to set precision of doubles

Used to get current time in specific format

Used for LinkedList and System.properties

Used to create user-friendly dynamic GUI

Alphabetically sorted imported libraries included in JRE

Used to create image file from javafx graph

Used to send email with attachment

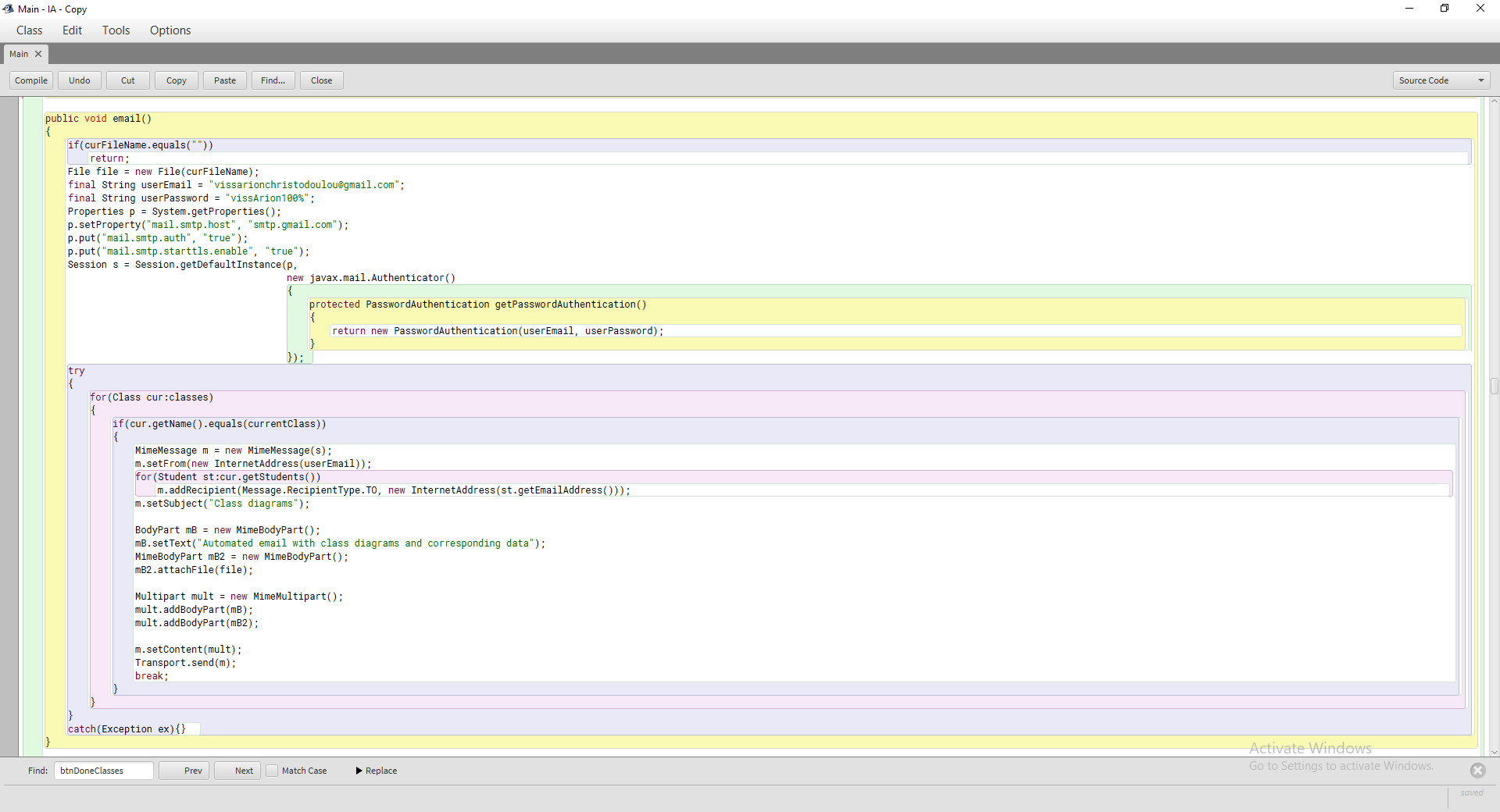
Used to manage pdf documents

Used to manage user-defined mathematical functions

**Figure 1.1: Libraries imported in main Class**

It is worth noting how the external libraries are used. javax.mail[[1]](#endnote-1) and Apache PDFBox[[2]](#endnote-2) are presented below, while mXparser[[3]](#endnote-3) (org.mariuszgromada) will be presented throughout the document.

**Figure 1.2: Using javax.mail to send emailiv**



If no file has been created terminate function

Load file

Start new session, log in

curFileName is a String member-variable

Current class is a string member variable

Linear search to find active class

Add every student as recipient

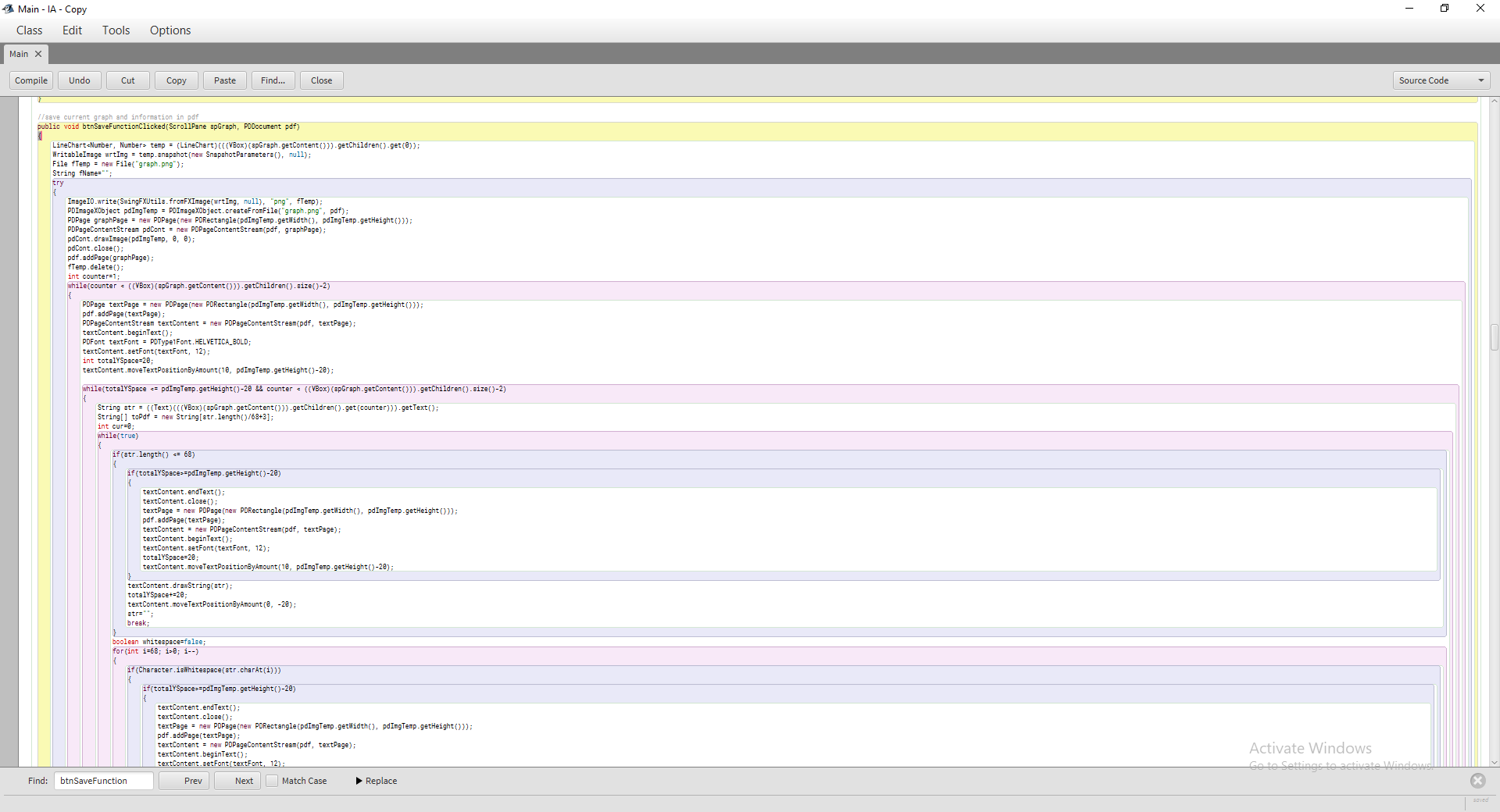
Set subject

Set text

Attach file

Send email

//send pdf created to students of currentClass



If string fits in one line just print it

Nested while to fill each line with text

Nested while to fill each page with text

Create new page and prepare for writing

Loop through all text elements

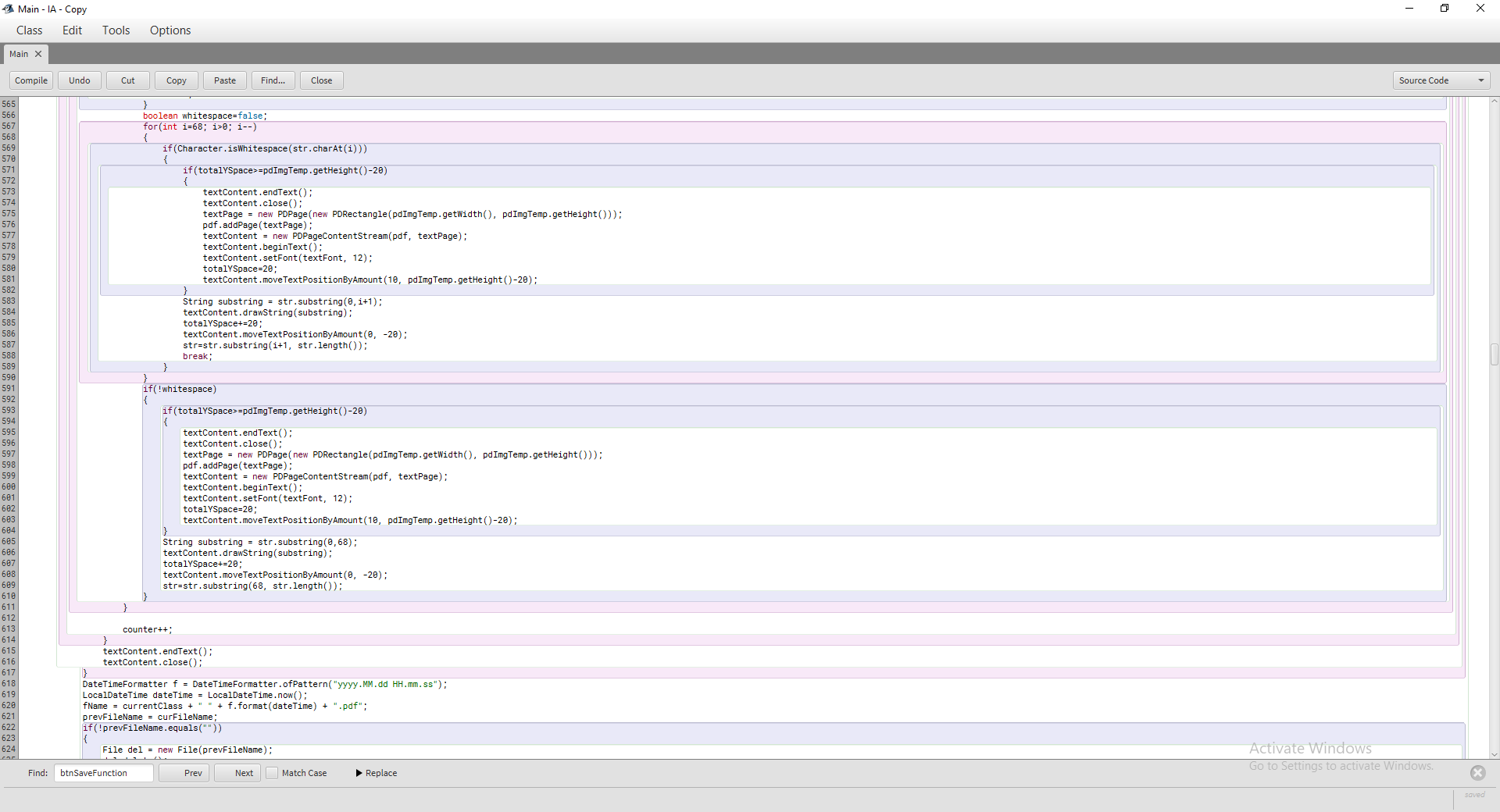
Delete image file

Add image to new pdf page

Get the graph

Create image file from graph

**Figure 1.3a: Using Apache PDFBox to add to pdf**



Easy to identify

Versions from previous sessions not overwritten

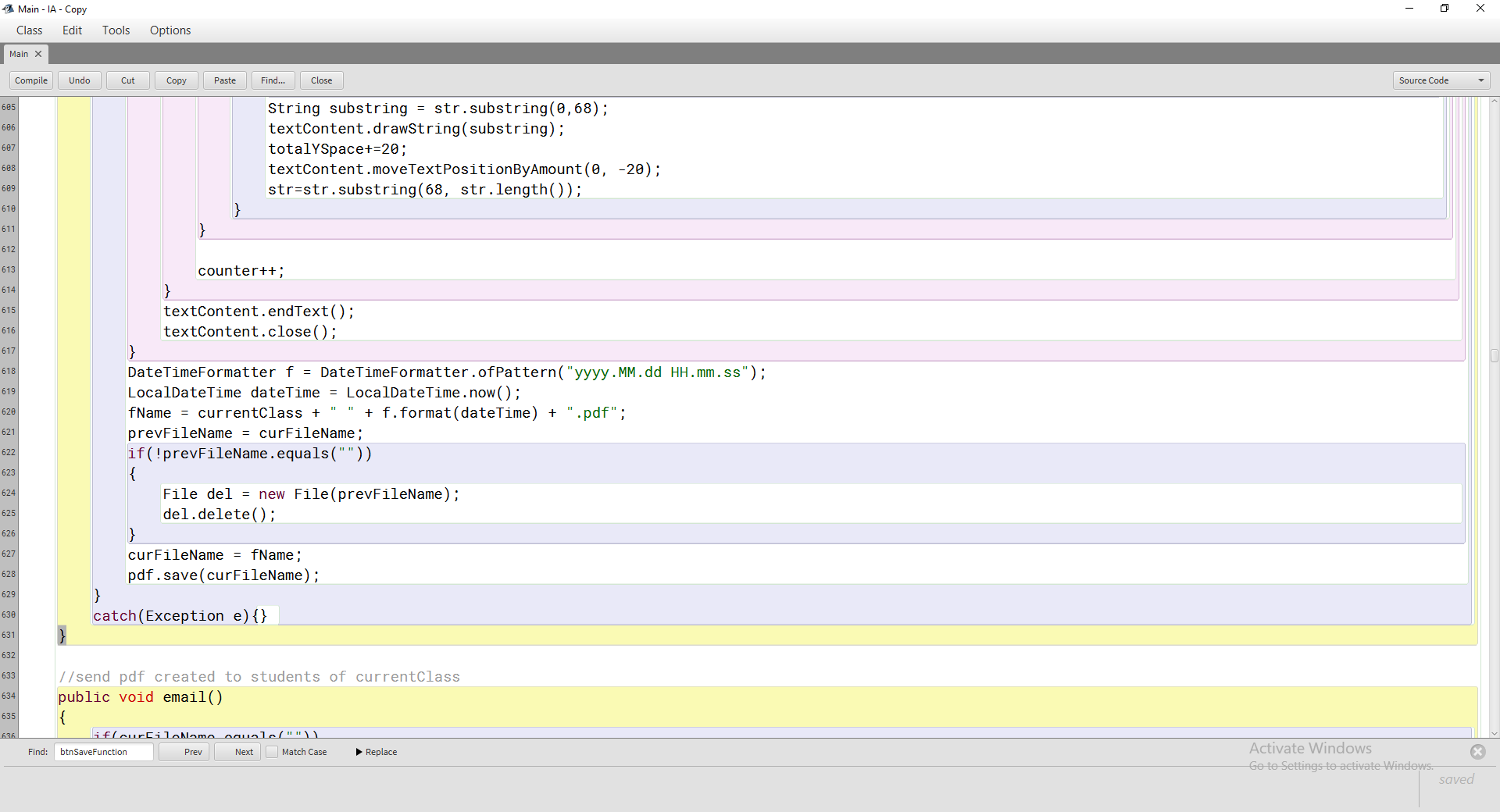
Class, date and time in file name

Break line when whitespace found

If there is no whitespace break line after 68th character

Nested for loop to find last whitespace before 68th character

**Figure 1.3b: Using Apache PDFBox to add to pdf (Continued)**



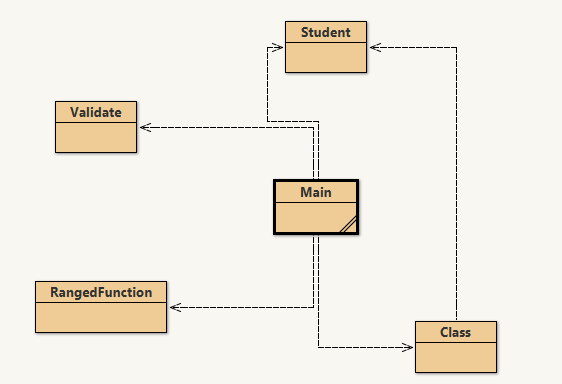
Delete previous version from the same session, since it was modified

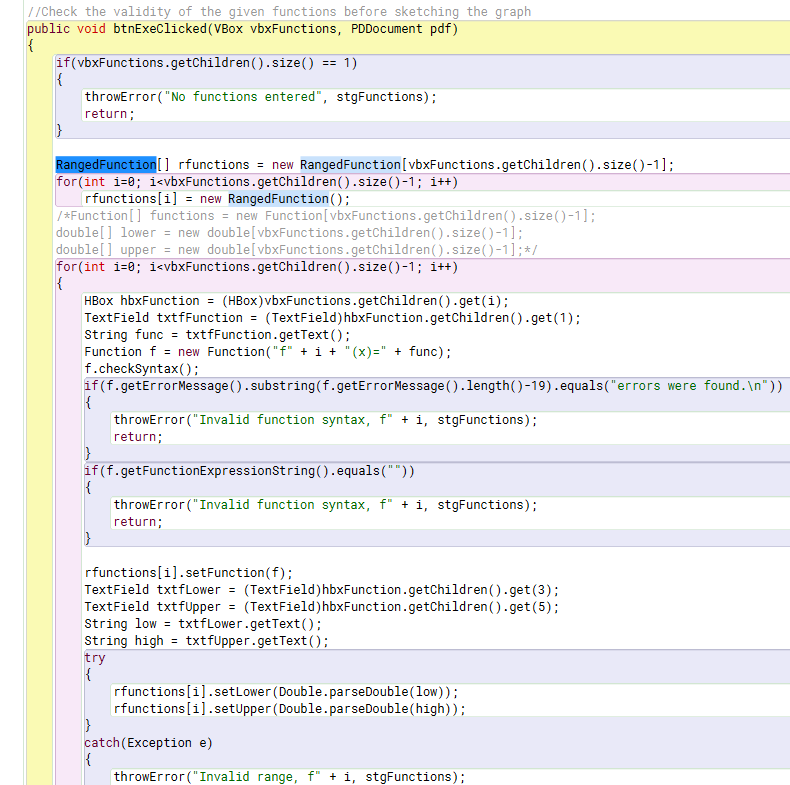
**Figure 1.3c: Using Apache PDFBox to add to pdf (Continued)**

# Object-Oriented Programming (OOP) – Abstraction

OOP, evident from the class dependencies depicted in Figure 2.1, is yet another example of abstraction. Each class does not need to know how the other classes work; it can just access their public fields. Moreover, objects represent real world entities and through them, a more precise syntax can be acquired of the form subject.method(object) instead of method(subject, object) (in this context object refers to the syntactical position in an English sentence). A characteristic example of the use of classes is the use of the RangedFunction class in the code in Figure 2.2. Instead of using parallel arrays for the functions and their upper and lower bounds, I preferred to use objects of type RangedFunction that can be passed with less code from one method to another as arguments, while simultaneously the intrinsic relationship between the function and its boundaries is depicted.

**Figure 2.1: Class Connections**





Check if **numbers** were entered as range by user

Check if function entered by user is valid

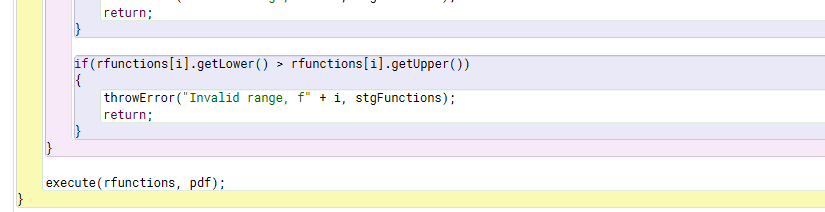
Instead of three arrays, function, lower and upper

**Figure 2.2a: Use of RangedFunction objects instead of parallel arrays and validating user input**

One argument needs to be passed instead of three

Check if range entered by user is valid

**Figure 2.2b: Use of RangedFunction objects instead of parallel arrays and validating user input (Continued)**



# Encapsulation – Abstraction

The use of encapsulation also serves as an example of abstraction. All member variables in all classes other than main are private and have corresponding accessor and mutator methods, as proved in Figures 3.1-3.3. This ensures that when a member variable is changed, it will be done through methods defined within the class and will, as a result, not be modified by mistake.

Both constructors with different parameter lists

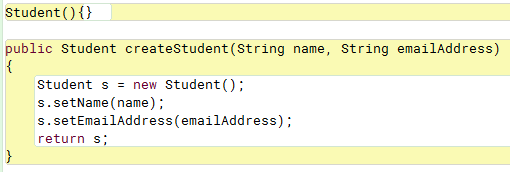
Much clearer than the alternative in Figure 3.1b

Public accessors and mutators

Private member variables



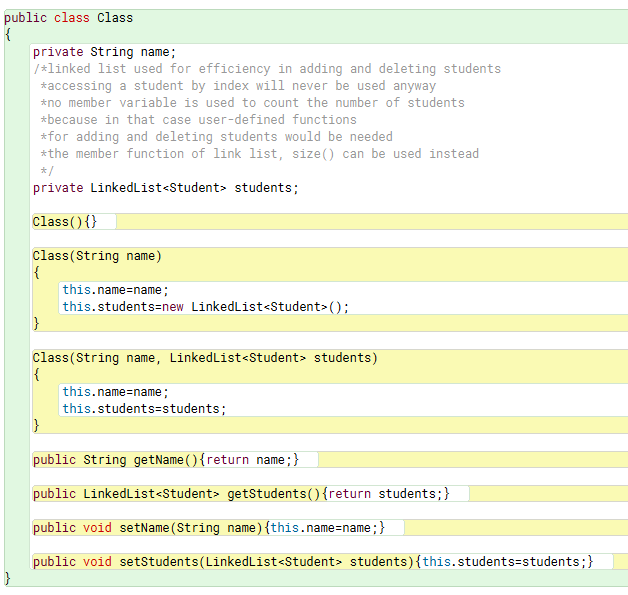
**Figure 3.1a: Encapsulation and static polymorphism in class Student**



**Figure 3.1b: Student without polymorphism**

Less obvious name

Not so clear code

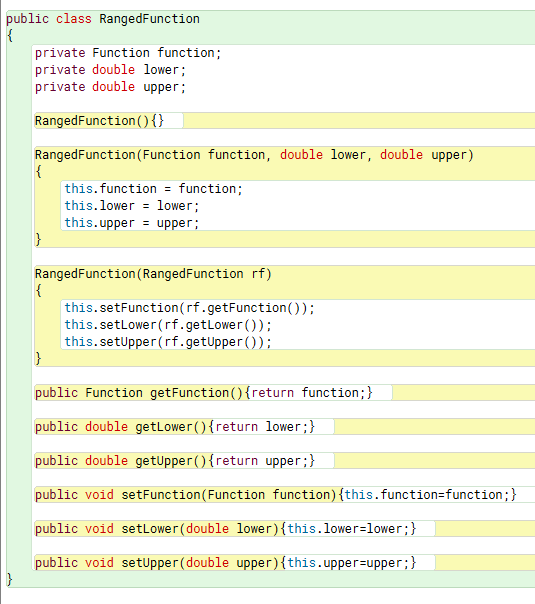


Multiple constructors with different parameter lists

Public accessors and mutators

Private member variables

**Figure 3.2: Encapsulation and static polymorphism in class Class**



Public accessors and mutators

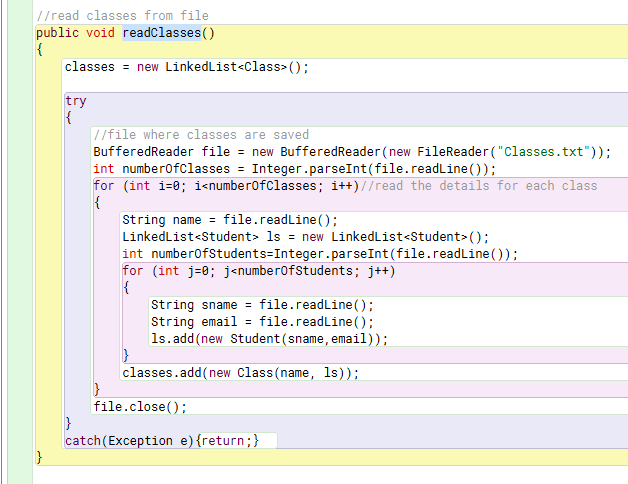
Multiple constructors with different parameter lists

Private member variables

**Figure 3.3: Encapsulation and static polymorphism in class RangedFunction**

# File Access

Except for working with pdfs, the program also interacts with a text file. It reads all information about Mr. Christos’ classes and students when it starts running, and saves any changes when they take place. This provides a way to save information from one session to another, which is programmatically easy and efficient.



Close file

Add the class in LinkedList classes

for loop to read details for each student

Read number of students

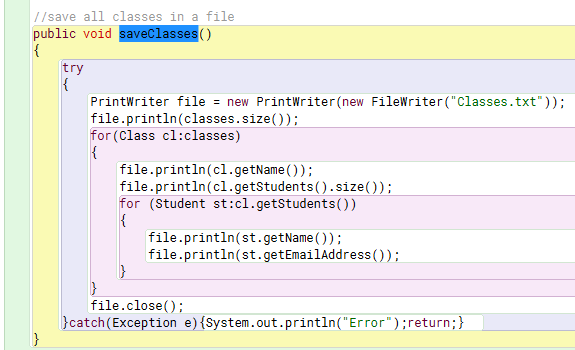
for loop to read details for each class

Read number of classes

Open file

Try-catch block: an error thrown will not crash the program

**Figure 4.1: Reading classes from text file and try-catch blocks**



Close file

Try-catch block: an error thrown will not crash the program

Enhanced for loop to read details for each student

Open file

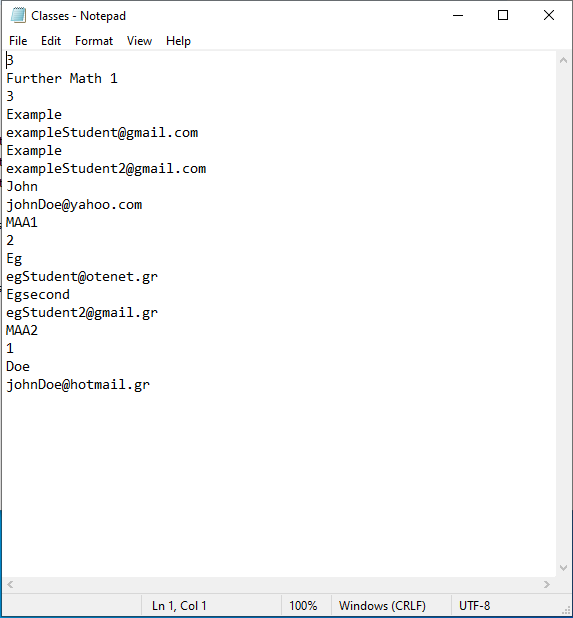
Print number of classes

Enhanced for loop to print details for each class

Print number of students

**Figure 4.2: Saving classes to text file and try-catch blocks**

**Figure 4.3: Example text file**



Name of first class

Number of classes

Number of students in first class

Students of first class

# Validation

Validation methods ensure that the user does not enter invalid data. It is essential to prevent the program from crashing and to avoid bugs. The Validate class, as shown in Figure 5.1, provides two methods for ensuring that when adding a new student, the name and email address are valid. As was noted in Figure 2.2, validation also takes place when the user inputs functions and ranges. Figure 5.3 outlines the behavior of the program when the user inputs invalid data in any field.

The rest should be lowercase

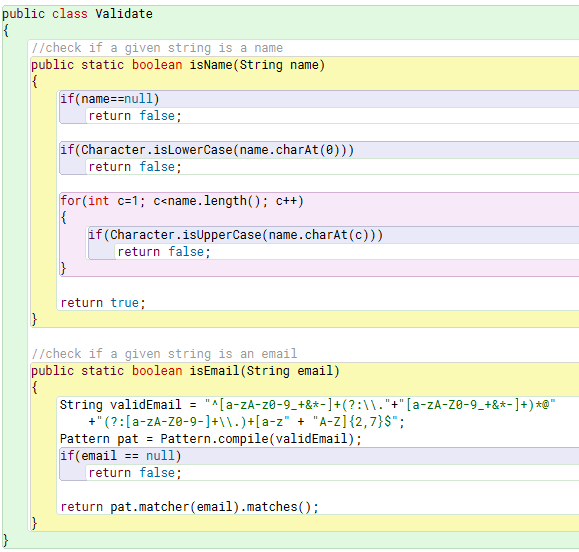
First letter should be uppercase

Method adoptedv

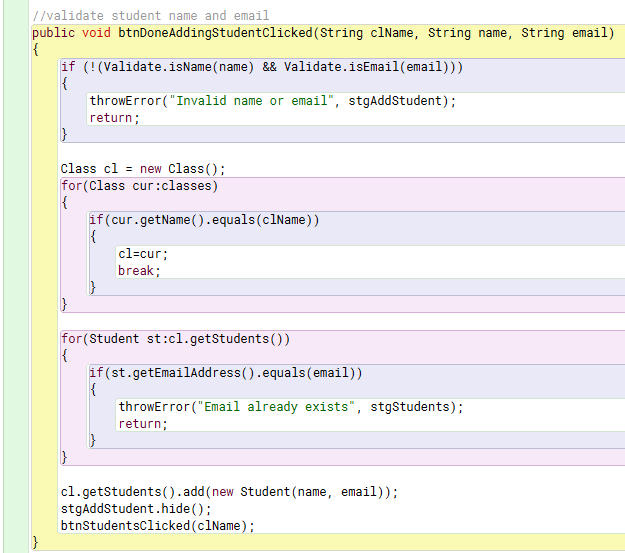
Regular expression

Check if the given string satisfies it

**Figure 5.1: The Validate class**



**Figure 5.2: The validate class in use**



Add student in class

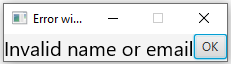
Enhanced for loop: uniqueness of email

Enhanced for loop: linear search

**Figure 5.3a: Entering invalid input**



**Figure 5.3b: The program’s response to invalid input**



# Error handling

Error handling is also an essential part of this program, keeping it from crashing. It takes the form of try-catch blocks, and it may be a part of validating user input, as in Figure 2.2, or of ensuring that the program does not face unexpected errors, as explained in Figures 4.1 and 4.2.

# Static polymorphism

Static polymorphism, achieved through function overloading, namely having multiple functions with the same names but different parameter lists, helps in creating programs where the method names are indeed representative of what the method does without being too descriptive. Moreover, they reduce code redundancy. These two points are explained in Figures 3.1-3.3.

# Dynamic polymorphism

Dynamic polymorphism is what makes the javafx library usable in the first place, for it allows, by extending the Application class and overwriting its start method, to run the code on a JavaFX Application thread.



Event handlers 🡪dynamic GUI

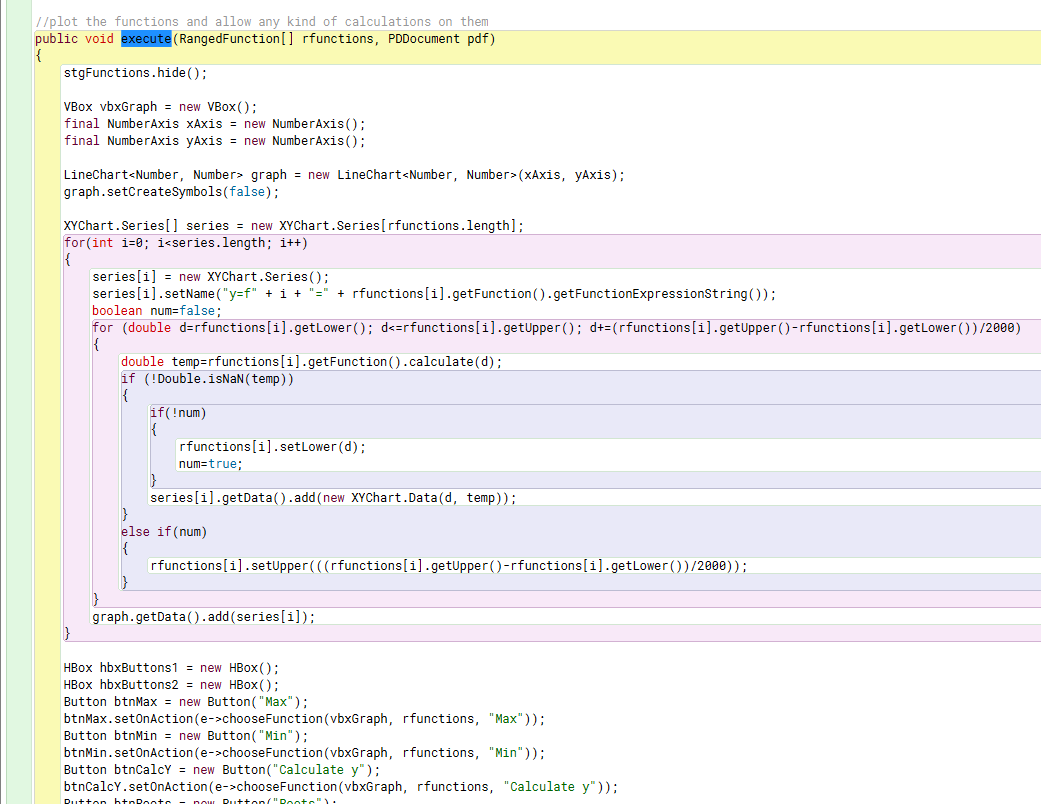
Image copiedvi

Override method start defined in superclass Application

**Figure 8.1: Overriding method start**

# Graphs

One of the main purposes of the application created was to draw graphs. This is achieved by calculating the y-value for thousands of x-values within the function’s domain, adding them as points in a set of axes and connecting these points with straight lines. The great number of points in a limited amount of pixels will give the graphs of the curves their real shape, meaning that the graph will look like a connection of straight lines, although essentially it is nothing more than that, as can be derived from Figure 9.1. The resulting graph and any data calculated on it can be seen in Figure 10.6.



Upper limit changed if function not defined for large values

Add point

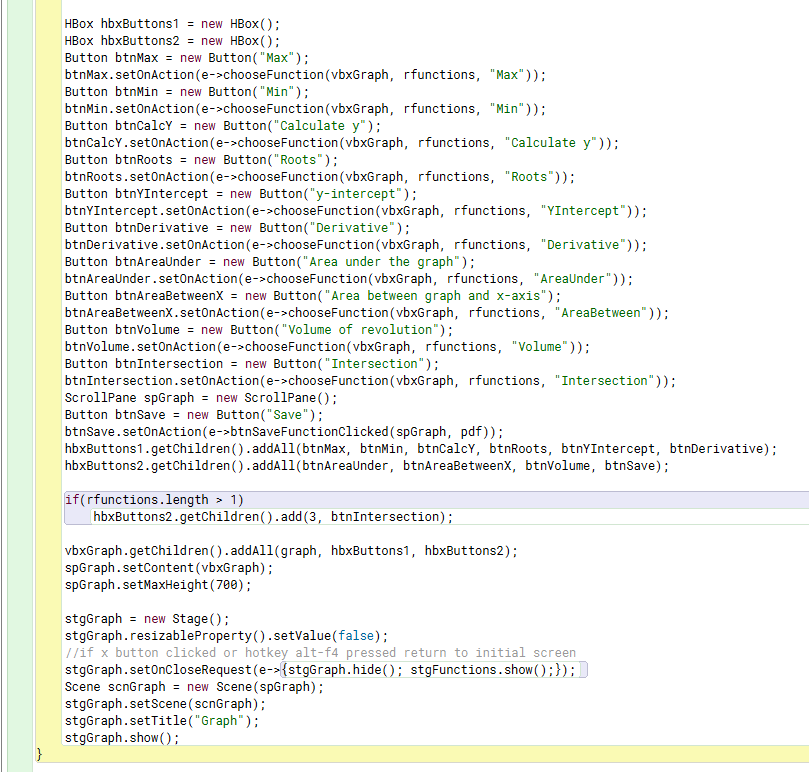
Lower limit changed if function not defined for small values

Nested for loop to calculate the coordinates of 2000 successive points

for loop to add new curve for each function

Create graph with axis

**Figure 9.1a: Drawing the graph**



Add intersection button only if more than one functions were entered

Create buttons with event handlers

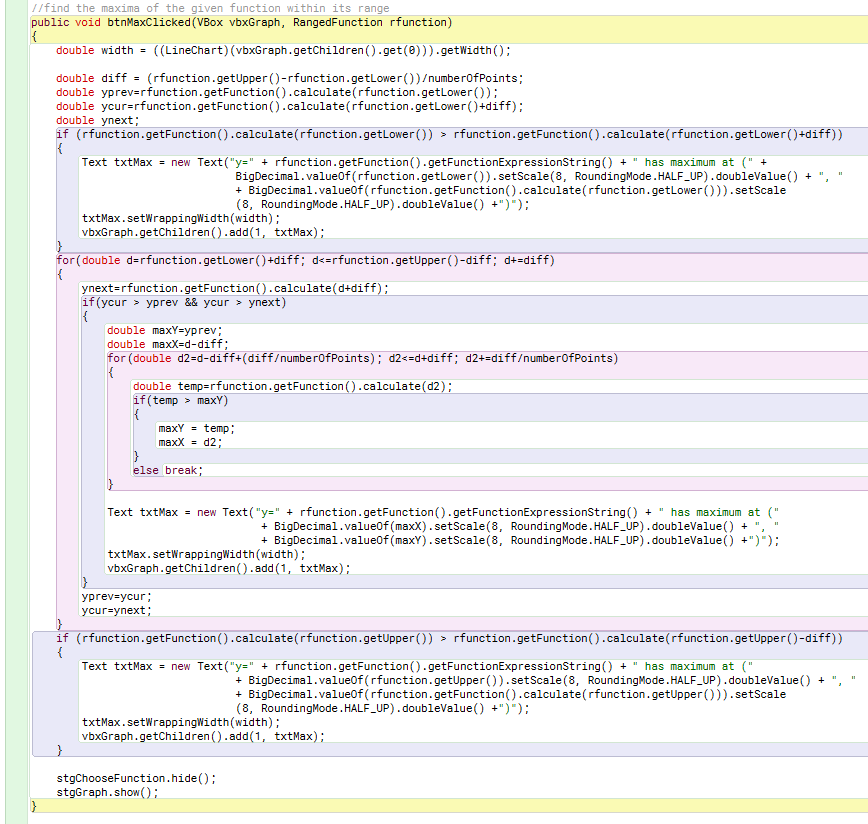
**Figure 9.1b: Drawing the graph (Continued)**

# Implementation of mathematical functions

The other crucial feature of the program was to be able to calculate certain values given a function. These have been implemented in a very brute-force way, presented in figures 10.1-10.5, which may lead to some error when an extremely large domain is given for x. Of course, I am referring to extreme cases, such as entering the function x2 in the range (-10e5, 10e5), which are very rarely useful in real life, and even less so in a mathematics class. I opted for these algorithms because they terminate after a predetermined number of iterations, much faster than what it would need for a more sophisticated, recursive or iterative approach to converge. This, in combination with the limited precision that Mr. Christos requires, convinced me that the overhead of having to implement a more complicated solution was not worth it for this aspect of the program.

Maxima

**Figure 10.1: Finding local maxima**



Set precision to eight decimals

Lower limit is a local max if greater than next value

If larger than both previous and next value

for loop: iterate over finite number of points

nested for loop: find exact coordinates of turning point

Upper limit is a local max if greater than previous value

Minima

The code is similar.

y at x0

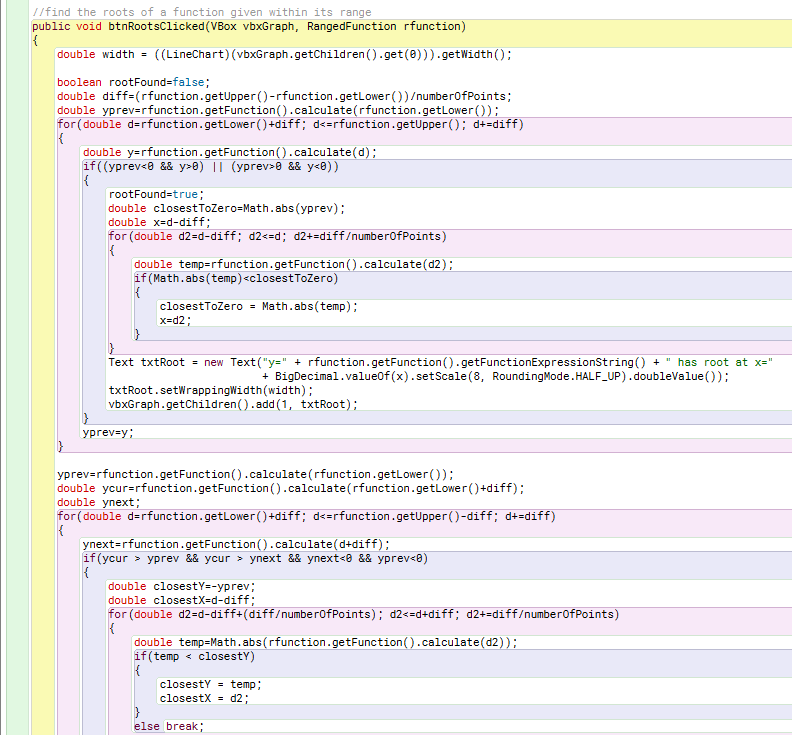
Calculating y given an x-value x0 is as easy as calling f(x0) after checking that

lower bound < x0 < upper bound

y-intercept

Simply call f(0) if lower bound < 0 < upper bound

Roots



nested for loop: find closest value to 0

For better accuracy check this case which might find missed roots

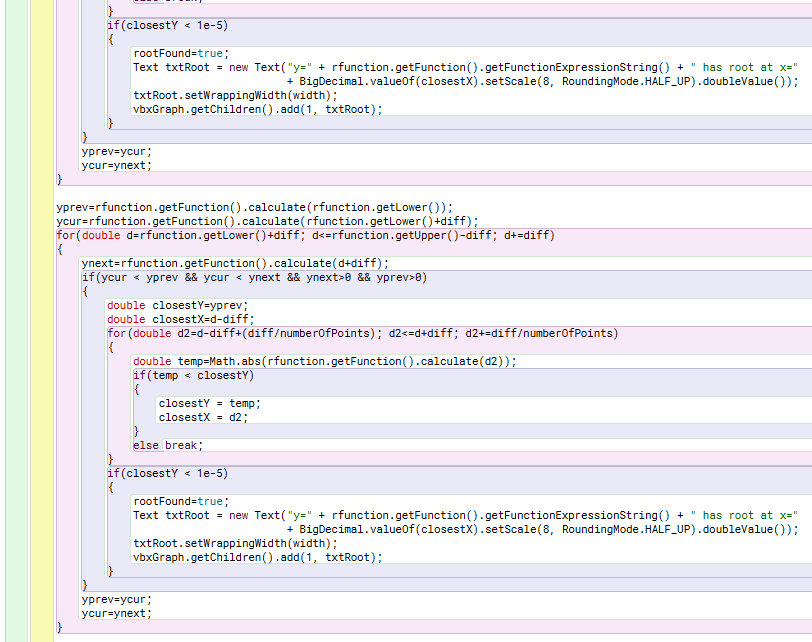
nested for loop: find exact coordinates of root

By Rolle’s theorem, if this is true then a root exists

for loop: iterate over finite number of points

**Figure 10.2a: Finding roots**

If close enough to 0, consider the point a root



If close enough to 0, consider the point a root

nested for loop: find closest value to 0

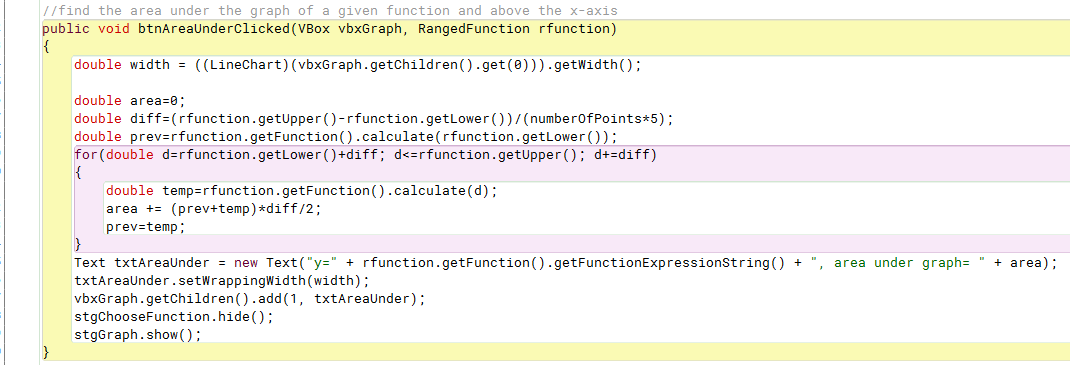
For better accuracy check this case which might find missed roots

**Figure 10.2b: Finding roots (Continued)**

Derivative at x0

This is the rate of change at x0. We can just see how f changes very close to x0. The answer will thus be , where h is a small number.

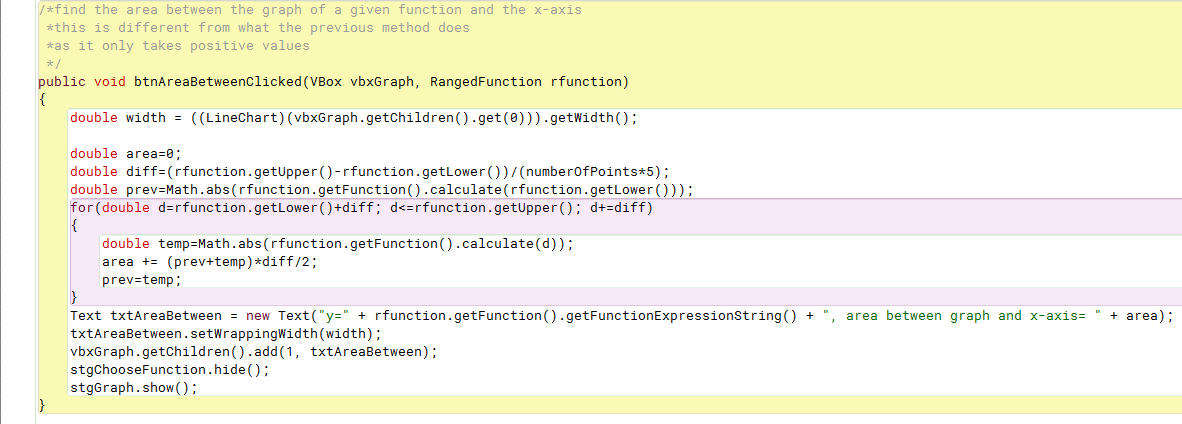
Area under the graph



Use trapezium method in f(x) for area under graph

**Figure 10.3: Finding Area Under Graph**

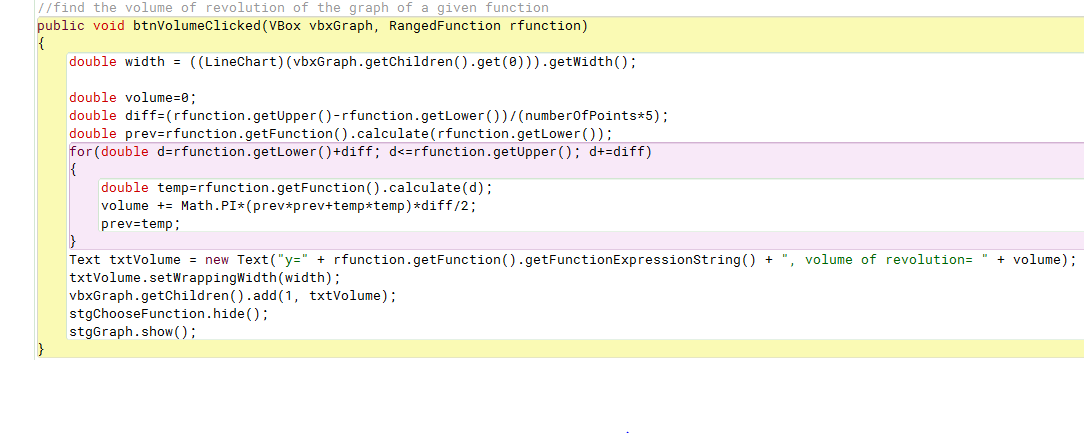
Area between the graph and the x-axis



Use trapezium method in |f(x)| for area between graph and x-axis

**Figure 10.4: Finding Area Between Graph and x-axis**

Volume of revolution



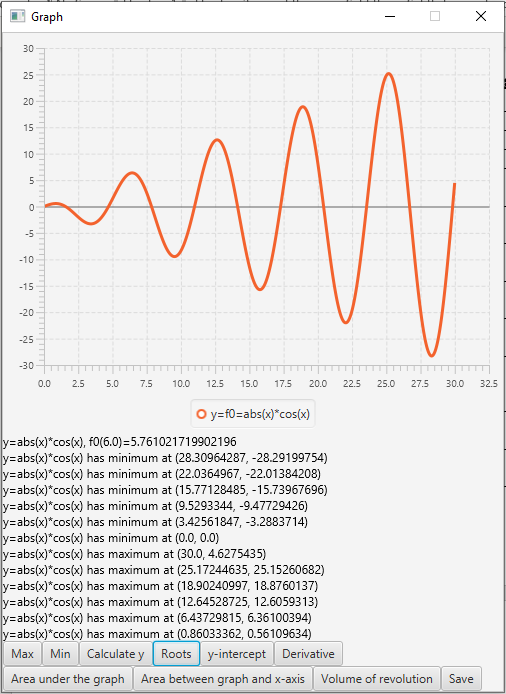
Use trapezium method in for volume of revolution

**Figure 10.5: Finding Volume of Revolution**

Point of intersection of two functions

Assuming two functions f(x) and g(x), their points of intersection are the same as the roots of f(x)-g(x).

**Figure 10.5a: Plotted function and data calculated**



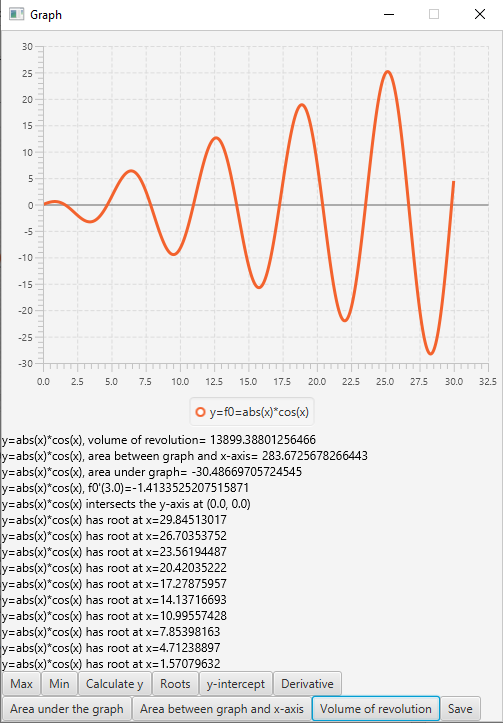
Plot function

Value of y at given value of x

Local minima

Local maxima

**Figure 10.5b: Plotted function and data calculated (Continued)**



Volume of revolution

Area between graph and x-axis

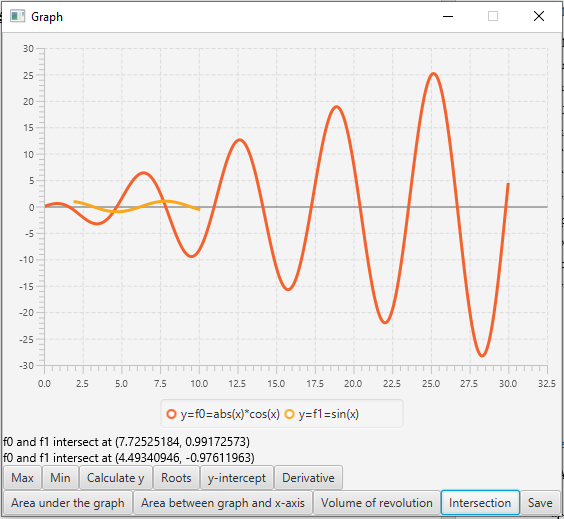
Area under graph

Numeric derivative

y-intercept

Roots

**Figure 10.5c: Plotted function and data calculated (Continued)**



Identify functions

Plot multiple functions

Intersection of functions

# Inline events

Inline events, meaning blocks of code instead of method calls when an event-handler is called, can have a lot of advantages. Even though, under some circumstances they can do more harm than good to one’s program, when used judicially, for very short methods that are not called many times, they can make both the code itself clearer and more readable and its execution slightly faster. The first is true because inline events reduce the vertical whitespace of the code, and also, when reading code with a lot of methods, one has to see where each one of them is defined and keep track of where they were called, which is avoided by using inline events. As for faster execution, similarly to how a programmer might have difficulty reading a program with many methods, a compiler will have to stop and start execution from different points several times, while simultaneously the free space in the heap will be reduced, slowing down the execution, which can again be avoided by using inline events. Therefore, inline events have, to some degree, been exploited for the purposes of my program.

**Figure 11.1: An example of an inline event**



Short and elegant inline event

Word Count: 1126

# References

“Apache PDFBox® - A Java PDF Library.” Apache PDFBox | A Java PDF Library,

pdfbox.apache.org/.

“Check If Email Address Valid or Not in Java.” GeeksforGeeks, 4 Feb. 2018,

www.geeksforgeeks.org/check-email-address-valid-not-java/.

“Complex Mathematics Images.” Shutterstock,

www.shutterstock.com/search/complex+mathematics.

Dimitriou, Kostas, and Markos Hatzitaskos. Core Computer Science: for the IB Diploma Program (International Baccalaureate). Express Publishing, 2018.

Dimitriou, Kostas, and Markos Hatzitaskos. Advanced Computer Science: for the IB Diploma Program (International Baccalaureate) High Level Computer Science. Express Publishing, 2018.

“Example of Sending Email in Java - Javatpoint.” Www.javatpoint.com,

www.javatpoint.com/example-of-sending-email-using-java-mail-api.

Java Platform Standard Edition 8 Documentation, docs.oracle.com/javase/8/docs/.

JavaMail, javaee.github.io/javamail/.

“MXparser – Math Expressions Parser for JAVA Android C# .NET/MONO/Xamarin –

Mathematical Formula Parser / Evaluator Library.” MXparser – Math Expressions Parser

for JAVA Android C# .NET/MONO/Xamarin – Mathematical Formula Parser / Evaluator

Library, mathparser.org/.

1. https://javaee.github.io/javamail/ [↑](#endnote-ref-1)
2. https://pdfbox.apache.org/ [↑](#endnote-ref-2)
3. http://mathparser.org/

   iv Adopted code from https://www.javatpoint.com/example-of-sending-email-using-java-mail-api

   v https://www.geeksforgeeks.org/check-email-address-valid-not-java/

   vi https://www.shutterstock.com/search/complex+mathematics [↑](#endnote-ref-3)