USER/DEVELOPER MANUAL

for

Code Visualization

Version 1.0

Prepared by : Curtice Gough
Joshua Hartzfeld
Catherine DiResta

Submitted to : Dr. Philip Chan
Instructor

April 15, 2024

Contents

1 Introduction

1.1 Overview

Code Visualization is a project that aims to simplify the debugging process via the use of visual diagrams. Mainstream debuggers such as GNU Debugger (GDB) and Windows Debugger (WinDBG), as well as built-in debuggers shipped with popular IDEs are undeniably useful. However, the learning curve required to become comfortable with their everyday use can often be quite steep. Code Visualization provides detailed visual representations of in-memory data to more clearly convey information at each program state.

1.2 Core Features

The key features of Code Visualization include:

- Code editor
- Data visualization
- Custom classes support
- Forward and backward execution

Each of these features are described in further detail in chapters 3 and 4.

1.3 User Demographics

Code Visualization is a proof-of-concept aimed to assist beginner-level programmers and students in understanding various Data Structures concepts as well as debugging simple coding tasks. As such, advanced features such as stack frame selection are secondary tasks that are not yet implemented.

2 Installation

2.1 System Requirements

2.1.1 Operating System

Code Visualization has only been tested on Linux systems running the X Window System (X11). Because of the directory traversal method by which Code Visualization communicates with the Traceprinter backend, the software does not currently run on Windows systems.

2.1.2 Python Libraries

The majority of Code Visualization's frontend was written in Python 3. Install dependencies using the following command:

```
pip install PyQt6 \
    json \
    subprocess \
    sys
```

2.1.3 Java JDK

The Traceprinter backend for Code Visualization requires a very specific version of the Java Developer Kit (JDK). This version of Java is no longer publicly available, and as such is shipped built-in with Code Visualization.

```
java version "1.8.0_20"
Java(TM) SE Runtime Environment (build 1.8.0_20-b26)
Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode)
```

2.2 Compiling from Source

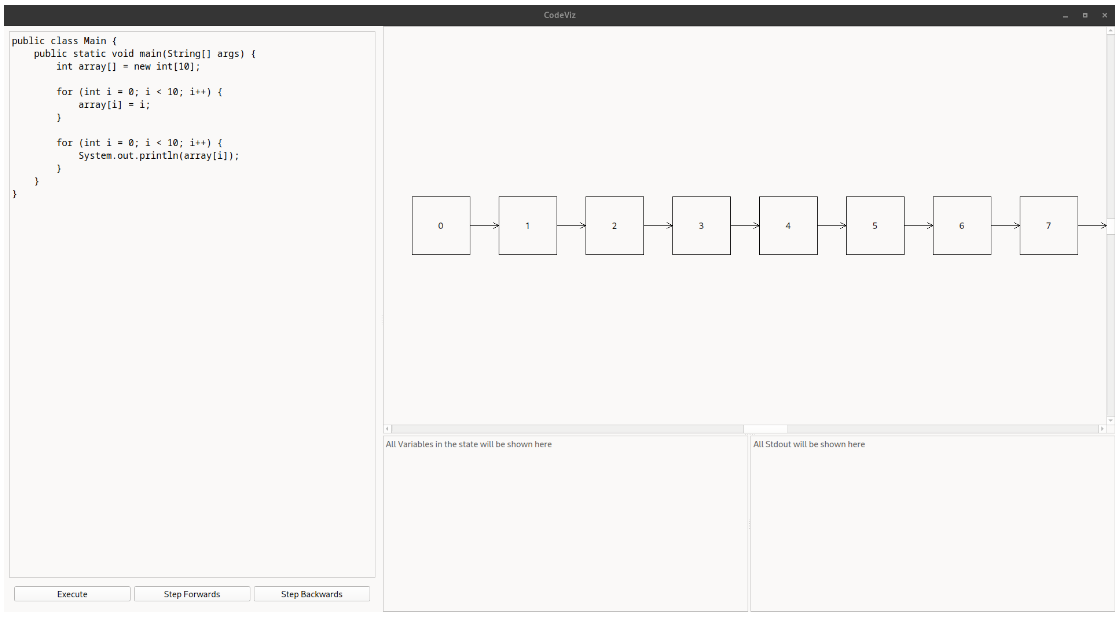
Each of Traceprinter's Java source files must be compiled before use. Use the included Makefile to simplify the process.

```
cd traceprinter_backend/cp/
make
```

3 Getting Started

Code Visualization is started by running GUI.py

```
python3 GUI.py
```



```
public class Main {
    public static void main(String[] args) {
        int array[] = new int[10];

        for (int i = 0; i < 10; i++) {
            array[i] = i;
        }

        for (int i = 0; i < 10; i++) {
            System.out.println(array[i]);
        }
    }
}
```

All Variables in the state will be shown here

All Stdout will be shown here

3.1 Code Editor

The leftmost pane of the main window is the Code Editor pane. Here, any valid Java 8 code can be written and executed for visualization. The user must provide a full and complete Java program. Code snippets are not enough. If it does not compile on the user's system, it will not compile in Code Visualization. An example of a valid program is shown below:

```
public class Main {
    public static void main(String[] args) {
        int array[] = new int[10];

        for (int i = 0; i < 10; i++) {
            array[i] = i;
        }

        for (int i = 0; i < 10; i++) {
            System.out.println(array[i]);
        }
    }
}
```

Click ”Execute” to begin tracing. Step through the program states using the ”Step Forwards” and ”Step Backwards” buttons.

3.2 Data Structures View

During tracing, the Data Structures View will show visual diagrams of supported data structures at each program state. When said diagrams are especially large, the user is able to click-and-drag to navigate the visualization area. Zooming is not currently supported by Code Visualization.

3.3 STDOUT

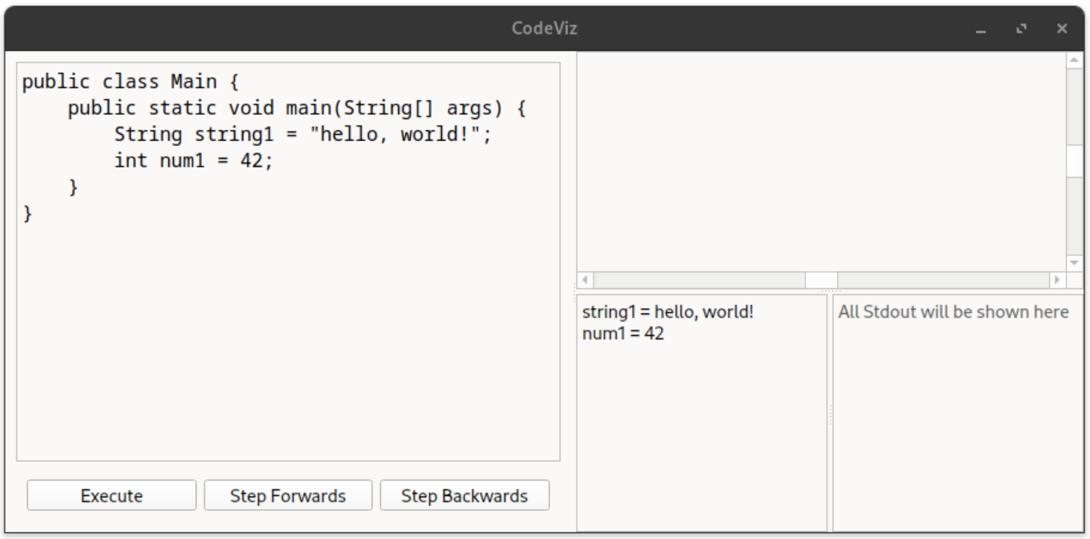
During execution of user-provided code, any text to be printed to the console (STDOUT) is shown in a pane underneath the Data Structures View. STDOUT will reflect the contents of STDOUT at each program state, regardless of order of execution. Stepping backwards from a print statement will remove that line from STDOUT.

4 Supported Data Structures

Code Visualization currently supports visualization of three data types, with room for expansion (see section 5.2).

4.1 Primitives

All variables containing primitive data types are listed underneath the Data Structures View pane.



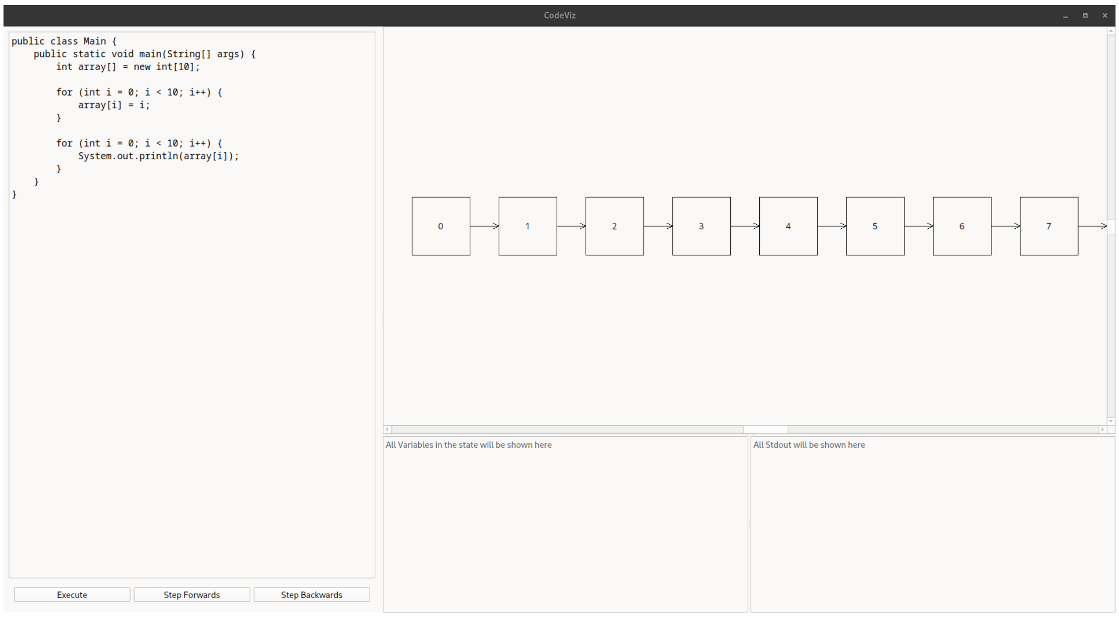
```
public class Main {
    public static void main(String[] args) {
        String string1 = "hello, world!";
        int num1 = 42;
    }
}
```

```
string1 = hello, world!
num1 = 42
```

All Stdout will be shown here

4.2 Arrays

Arrays are presented with a generic 'LIST' diagram in the Data Structures View. The contents of each 'LIST' item may be any supported 'primitive' data type.



```
public class Main {
    public static void main(String[] args) {
        int array[] = new int[10];

        for (int i = 0; i < 10; i++) {
            array[i] = i;
        }

        for (int i = 0; i < 10; i++) {
            System.out.println(array[i]);
        }
    }
}
```

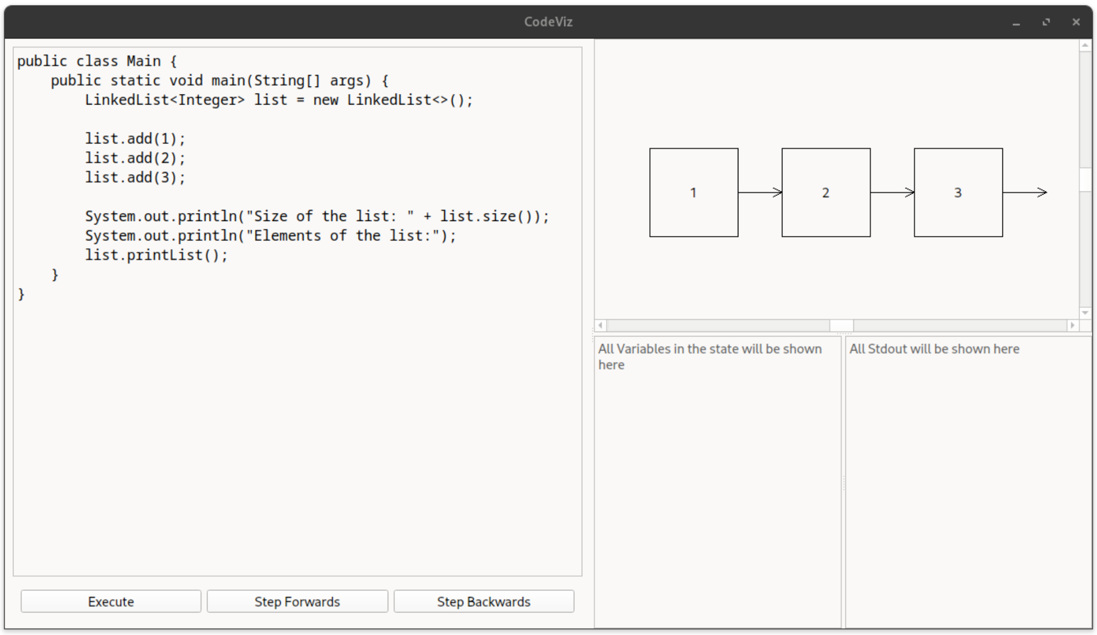
All Variables in the state will be shown here

All Stdout will be shown here

4.3 Linked Lists

Linked Lists are presented with a generic 'LIST' diagram in the Data Structures View. Thew contents of each 'LIST' item may be any supported 'primitive' data type. Code Visualization currently only supports the built-in LinkedList class included under:

```
traceprinter_backend/cp/codeviz/
```



```
public class Main {
    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();

        list.add(1);
        list.add(2);
        list.add(3);

        System.out.println("Size of the list: " + list.size());
        System.out.println("Elements of the list:");
        list.printList();
    }
}
```

All Variables in the state will be shown here

All Stdout will be shown here

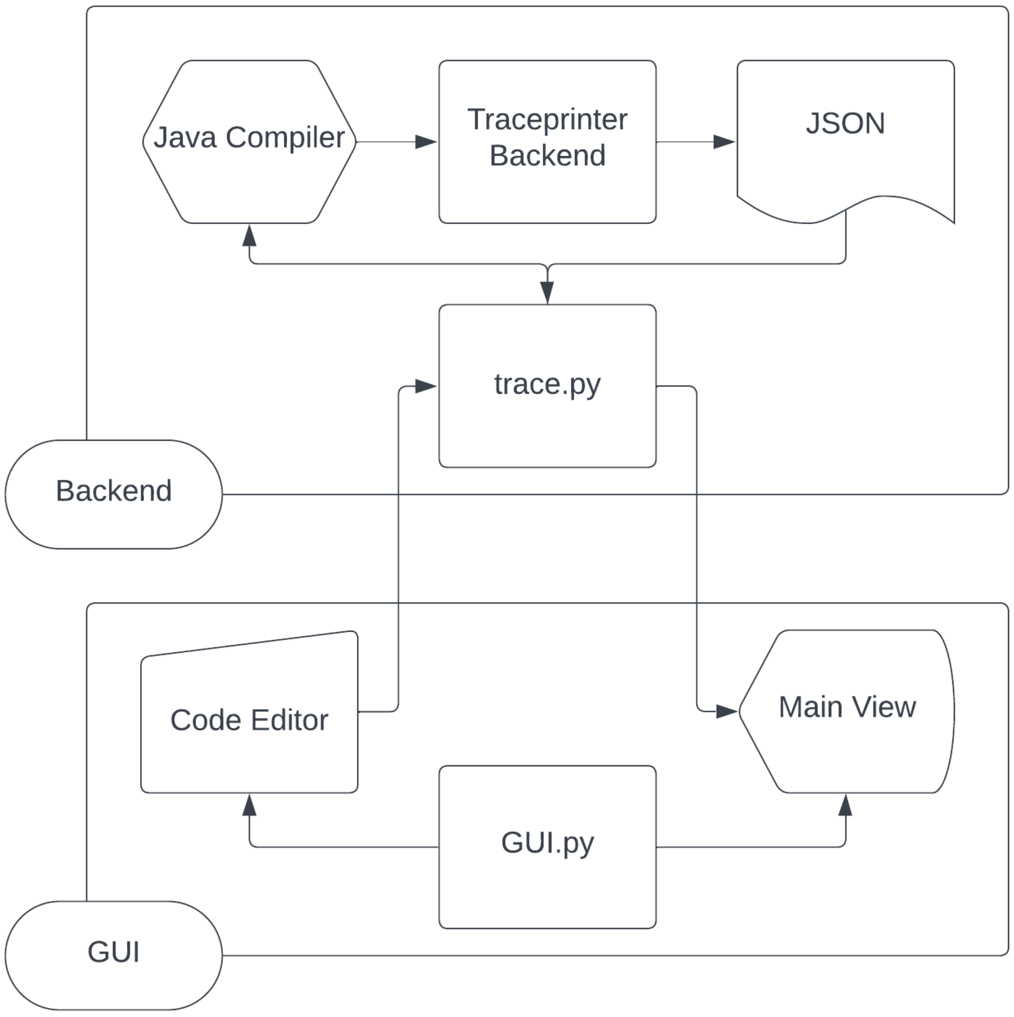
4.4 Custom Data Structures

All .java source files placed in the codeviz directory are automatically added to the classpath and compiled alongside user-provided code. Custom data structures may be added by placing the source files in this directory. For example, Linked Lists (see 4.3) are a custom data structure placed in the codeviz directory.

```java
public class LinkedList<T> {

    // Node class representing each element in the LinkedList
    private static class Node<T> {
        T data;
        Node<T> next;

        public Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    private Node<T> head; // Head of the LinkedList
    private int size;     // Size of the LinkedList

    // Constructor to initialize an empty LinkedList
    public LinkedList() {
        head = null;
        size = 0;
    }

    // Method to add a new element to the end of the LinkedList
    public void add(T data) {
        Node<T> newNode = new Node<>(data);
        if (head == null) {
            head = newNode;
        } else {
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
        size++;
    }

    // Method to get the size of the LinkedList
    public int size() {
        return size;
    }

    // Method to check if the LinkedList is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Method to print the elements of the LinkedList
    public void printList() {
        Node<T> current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
}
```

5 Further Development

5.1 System Architecture



5.2 Parsing Traceprinter Output

All JSON parsing is done in trace.py.

5.2.1 JSON Structure

The output of Traceprinter is a JSON document containing a list of ”states” representing the contents of memory in the JVM at each step of execution. A single ”state” object may look something like this:

```json
{
  "stdout": "",
  "event": "step_line",
  "line": 3,
  "stack_to_render": [
    {
      "func_name": "main:3",
      "encoded_locals": {},
      "ordered_varnames": [],
      "parent_frame_id_list": [],
      "is_highlighted": true,
      "is_zombie": false,
      "is_parent": false,
      "unique_hash": "2",
      "frame_id": 2
    }
  ],
  "globals": {},
  "ordered_globals": [],
  "func_name": "main",
  "heap": {}
}
```

Stack Variables

Local stack variables at each state will appear under ”encoded_locals” in each entry of ”stack_to_render”. ”stack_to_render” is a list of all currently volatile stack frames, with the 0th entry being the current frame. Below is an example of a state where multiple local stack variables are present.

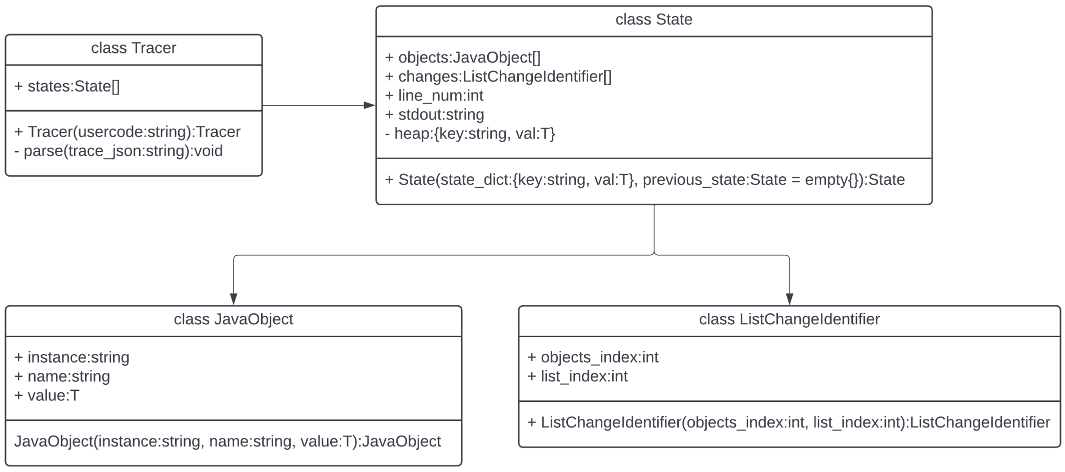
```json
{
  "stdout": "",
  "event": "step_line",
  "line": 5,
  "stack_to_render": [
    {
      "func_name": "main:5",
      "encoded_locals": {
        "x": 10,
        "hello": "hello, world!"
      },
      "ordered_varnames": [
        "x",
        "hello"
      ],
      "parent_frame_id_list": [],
      "is_highlighted": true,
      "is_zombie": false,
      "is_parent": false,
      "unique_hash": "7",
      "frame_id": 7
    }
  ],
  "globals": {},
  "ordered_globals": [],
  "func_name": "main",
  "heap": {}
}
```

Heap References

Global variables and class instances will be placed on the heap. Each heap reference has an ID that can be referenced in the ”heap” dictionary object in the JSON. Below is an example of a LinkedList in memory.

```
{
  "stdout": "Size of the list: 3\nElements of the list:\n1 2 3 \n",
  "event": "return",
  "line": 12,
  "stack_to_render": [
    {
      "func_name": "main:12",
      "encoded_locals": {
        "list": [
          "REF",
          476
        ],
        "__return__": [
          "VOID"
        ]
      },
      "ordered_varnames": [
        "list",
        "__return__"
      ],
      "parent_frame_id_list": [],
      "is_highlighted": true,
      "is_zombie": false,
      "is_parent": false,
      "unique_hash": "474",
      "frame_id": 474
    }
  ],
  "globals": {},
  "ordered_globals": [],
  "func_name": "main",
  "heap": {
    "476": [
      "INSTANCE",
      "LinkedList",
      [
        "head",
        [
          "REF",
          479
        ]
      ],
      [
        "size",
        3
      ]
    ],
    "479": [
      "INSTANCE",
      "Node",
      [
        "data",
        1
      ],
      [
        "next",
        [
          "REF",
          481
        ]
      ]
    ],
    "477": 1,
    "481": [
      "INSTANCE",
      "Node",
      [
        "data",
        2
      ],
      [
        "next",
        [
          "REF",
          483
        ]
      ]
    ],
    "480": 2,
    "483": [
      "INSTANCE",
      "Node",
      [
        "data",
        3
      ],
      [
        "next",
        null
      ]
    ],
    "482": 3
  }
}
```

5.2.2 trace.py UML


class Tracer
+ states:State[]
+ Tracer(usercode:string):Tracer
- parse(trace_json:string):void
class State
+ objects:JavaObject[]
+ changes:ListChangeIdentifier[]
+ line_num:int
+ stdout:string
- heap:{key:string, val:T}
+ State(state_dict:{key:string, val:T}, previous_state:State = empty{}):State
class JavaObject
+ instance:string
+ name:string
+ value:T
JavaObject(instance:string, name:string, value:T):JavaObject
class ListChangeIdentifier
+ objects_index:int
+ list_index:int
+ ListChangeIdentifier(objects_index:int, list_index:int):ListChangeIdentifier

5.3 GUI Development

We have developed the Main GUI program in a modular class-based format. The 2 main classes are the MainWindow and the MyGraphicsView. In the hierarchy of importance, Main should always call upon the other classes, i.e., MainWindow constructs the MyGraphicsView class and so on. The __init__ of MainWindow will also be the location of all baseline fundamental changes to the GUI. For example, adding any new windows or buttons will happen in MainWindow.__init__. Any changes relating to the Graphics View/visualization must happen in MyGraphicsView. In both classes, sub-methods are called upon when needed and linked to certain actions like button presses.

5.3.1 PyQt6 Widgets

Code Edit Widget

This widget is part of the foundation's main GUI window, which is called "code_edit." Its main function is to take valid Java code and send it to the Traceprinter back end when the Execute button is pressed.

```
self.code_edit = QPlainTextEdit()
code_font = QFont("Monospace", 12)
self.code_edit.setFont(code_font)
self.code_edit.setTabStopDistance(40)
```

Graphics Widget

The graphical widget is defined alongside code_edit; however, it constructs the MyGraphicsView class so that we can keep the code modular. This class handles all the visualization and drawing needed when called upon by the main window class.

```
class MyGraphicsView(QGraphicsView):
    def __init__(self):
        super().__init__()
        self.setDragMode(QGraphicsView.DragMode.ScrollHandDrag)
        self.setMouseTracking(True)
        self.last_mouse_pos = None
        self.setInteractive(True)
        self.scene = QGraphicsScene()
        self.setScene(self.scene)
        self.arrow = QPixmap("arrow.png")

        self.scene.setSceneRect(-10000, -10000, 20000, 20000)

    def draw_square(self, x, y, width, height):
        self.scene.addRect(x, y, width, height)

    def draw_array(self, y, width, height, items, distance,text):
        self.scene.clear()
        for i in range(items):
            x = i * (width + distance)
            self.scene.addRect(x, y, width, height)
            text_item = QGraphicsTextItem(str(text[i]))
            text_item.setPos(x+width/2 - text_item.boundingRect().width()
    /2,y+height/2 - text_item.boundingRect().height()/2)
            self.scene.addItem(text_item)
            self.draw_arrow(x+width,y+50,x+distance+width,y+50)

    def draw_arrow(self,start_x,start_y,end_x,end_y):
        self.scene.addLine(start_x,start_y,end_x,end_y)
        self.scene.addLine(end_x-10,end_y-5,end_x,end_y)
        self.scene.addLine(end_x-10,end_y+5,end_x,end_y)
```

Buttons

The main window setup defines the buttons and assigns their own functions. Execute will send code to Traceprinter, and receive a list of all the states generated by Traceprinter. Step forward and step backward with iterate through the list of states returned by Traceprinter and reflect the changes to the graphics view

Button Setup

```
code_execute_button = QPushButton("Execute")
step_forward_button = QPushButton("Step Forwards")
step_backwards_button = QPushButton("Step Backwards")
code_pane_buttons = QWidget()
code_pane_buttons_layout = QHBoxLayout(code_pane_buttons)
code_pane_buttons_layout.addWidget(code_execute_button)
code_execute_button.released.connect(self.code_execute)
code_pane_buttons_layout.addWidget(step_forward_button)
step_forward_button.released.connect(self.step)
code_pane_buttons_layout.addWidget(step_backwards_button)
step_backwards_button.released.connect(self.step_backwards)
```

Execute

```
def code_execute(self):
    usercode = self.code_edit.toPlainText()
    tracee = trace.Tracer(usercode)
    self.state_index = 0
    self.trace_report = tracee
```

Step Forward/backward

```
def step(self):
    if self.state_index + 1 < len(self.trace_report.states):
        self.state_index += 1
        print(f"State index -> {self.state_index}")
        self.show_state()

def step_backwards(self):
    if self.state_index == 0:
        return
    self.state_index -= 1
    print(f"State index -> {self.state_index}")
    self.show_state()
```

Variable and Stdout Widgets

These are the two windows below the Graphics widget. These are read-only QTextEdit widgets that we update every time a new state is shown. This is done inside the show_-state function defined below. In this method i also left in debug print statements that go to console to help with development debugging.

```python
def show_state(self):
    #print(type(trace.states[29].objects[0].name))
    #print(trace.states[29].stdout)
    #print(trace.states[29].objects[1].name)
    self.primitive_output.clear()
    self.view.scene.clear()
    for each in self.trace_report.states[self.state_index].objects:
        print(each.name)
        if each.instance == "primitive":
            self.primitive_output.append(f"{each.name} = {each.value}")
        elif each.instance == "LIST":
            print(f"LIST = {each.name}, {each.value}")
            print(each.value)
            print(len(each.value))
            self.view.draw_array(0,100,100,len(each.value),50,each.value)
    self.stdout_widget.clear()
    self.stdout_widget.append(self.trace_report.states[self.state_index].
   stdout)
    print("End of Debug")
```