

Practical 2

Software Design & Practices

Due date (Homework section): Wednesday 22 May at 11:59pm

In this practical session you will become familiar with the Git distributed version control system, and with testing. You will also use a different ecosystem of tools: the Java ecosystem.

Setup

1. Sign up for GitHub at <https://github.com> .
2. Sign up for ChatGPT at <https://chat.openai.com> .
3. If you aren't using a Hamilton Labs computer,
 - a. Install the Java SE Development Kit 18.0.1.1 from [this link](#).
 - b. Install Java by going to <https://java.com>, clicking on "Download Java", and following the instructions to install it. If it asks you to uninstall the 18.0.1.1 version, say no!

Warm-up

Part A: Testing a different ecosystem

We'll begin by installing the Java ecosystem. We'll start with a task runner, which we can then use to download and install other tools as necessary.

1. Download the Gradle "binary-only" software from [this link](#). Gradle is a task runner for the Java ecosystem; you can read about it [here](#). It comes as a zipfile.
-

-
2. Open the Windows Terminal and create a directory in the root filesystem by typing:
mkdir c:\Gradle
 3. Go into the zipfile that you downloaded, and extract the **gradle-8.0.2** folder into the **c:\Gradle** folder. Note that you are copying the whole **gradle-8.0.2** folder, and *not* just copying its contents.
 4. Right-click on the Windows menu button on the taskbar, and click on **System**.
 5. On the right side of the window that pops up, you will see an item named **Advanced system settings**. Click on it.
 6. Near the bottom of the “System Properties” window that appears, you will see a button entitled **Environment Variables...**. Click on it.
 7. Click on the “Path” variable, then click on the **Edit...** button.
 8. Double-click on a blank line and type: **c:\Gradle\gradle-8.0.2\bin**. Press Enter.
 9. Click on **OK** buttons until the “System Properties” window has disappeared.
 10. Close the terminal window. Then open a new Terminal window. (You need to do this so that the change you’ve made to the settings will be recognized by the Terminal).
 11. Type in **gradle -v**. If you see some diagnostic information that says “Gradle 8.0.2” at the top, then you’ve done well. Otherwise, you might have missed a step.

Now we’ll create a folder to work in, and start a new Java project.

12. In Windows Terminal, change to your Desktop folder: **cd Desktop**
 - a. It doesn’t actually matter where you are, as long as you know where you are working and can continue working from there if you want to come back to work on it later.
13. Create a folder called “sdp-warmup” and change into it: **mkdir sdp-warmup** and then **cd sdp-warmup**
14. Create a new project by typing: **gradle init --type java-application**
 - a. You can just press the **Enter** key to accept the defaults.

If you have Visual Studio Code installed, you can type **code .** from the Terminal to see the files in an IDE. Gradle has created a project with some basic automation already, and provided a script called **gradlew** for you to use with the project:

- To build the project, type **./gradlew build**
- To run the project, type **./gradlew run**
- To run the tests, type **./gradlew test**

You might notice that *the first time you run gradlew*, it *downloads* a copy of Gradle! From now on, the local copy of Gradle, which is stored in the **.gradle** folder, will be used. The benefit of this approach is that anyone who runs **gradlew**, whether they have Gradle installed on their system or not, will be able to use Gradle to perform tasks with your code—because the first time that **gradlew** is run, it gets Gradle from online. It doesn't need to be pre-installed on the system, so another developer who works on your code will never need to install it manually, like you did. In fact, if you want to, you can delete the **c:\Gradle** folder now, and remove the environment variable that we added, and you'll see that the **./gradlew** commands noted above will continue to work.

Let's create a git repository for our project, and practice using version control. It may help you to have your notes open and/or the slides, so that you can relate what you're doing to the DAG operations that you know about.

15. Type **git init** to initialize a local git repository.

16. Now type **git status** to understand what Git understands about the source code.

You should see a bunch of "untracked" files, with no commits yet, on the "master" branch.

17. You'll note that Gradle has created a **.gitignore** file for us—take a look and see what is inside it! Git is so "standard" in the Java ecosystem, and many other language ecosystems, that Gradle just assumed that we would be using Git for version control.

-
18. Now look at the **build.gradle** file. This is like the **package.json** file that you've seen, but the syntax is quite different. Can you see where the dependencies are? Can you see how it knows which method to call when you start your program? Can you see where the test runner is configured?
19. Let's check all the files into git. We'll begin by adding them: type **git add ***
20. Now type **git status** and see what it says. Notice that there are several files that have been added to the staging area. (However, *also* notice that nothing in the **.gradle** folder and nothing in the **app/build** folder have been added—they are *ignored* because of what is written in the **.gitignore** file!).
21. What are the files in the staging area? Let's take a look.
- The source code files are **App.java** and **AppTest.java**.
 - The task runner configuration files are **build.gradle** and **settings.gradle**.
 - Scripts and Java files that Gradle recommends adding to source control (see [here](#)) are **gradle-wrapper.jar**, **gradle-wrapper.properties**, **gradlew**, and **gradlew.bat**. Together, they form a "bootstrap" system that downloads Gradle if it's not already available in the **.gradle** folder.
 - Git-specific files are **.gitignore**, which you already know about, and **.gitattributes**, which tells Git to store some of Gradle's files with line-endings that different operating systems might expect.
22. Let's commit this set of changes. Type: **git commit -m "Initial commit"** and see the difference with **git status**.

So far, so good. Let's do some development now.

23. Open up **App.java** and add this method:

```
public Object[] flatten(Object[][] nested) {  
    List<Object> result = new ArrayList<Object>();  
    for (Object[] innerList : nested) {  
        for (Object item : innerList) {  
            result.add(item);  
        }  
    }  
}
```

```
}
Object[] out = new Object[result.size()];
result.toArray(out);
return out;
}
```

24. You will also need to **import java.util.*;** at the top of **App.java**. And while we're here, modify the greeting so that it says "Hello SDP Prac 2!" instead.

25. Let's add a test, too. Open up **AppTest.java** and add this method:

```
@Test void flattenUnpacksANestedArray() {
    // Arrange
    App classBeingTested = new App();
    Object[][] data = { {"hello", "world"}, {"how", "are", "you"}, {}, {"now?"} };
    Object[] expected = { "hello", "world", "how", "are", "you", "now?" };
    // Act
    Object[] result = classBeingTested.flatten(data);
    // Assert
    assertEquals(expected, result);
}
```

- a. Let's go through this. Our test is organized as ARRANGE→ACT→ASSERT.
- b. First we arrange everything that needs to be created or that will be used during the test. The **classBeingTested** is created; the **data** that we will use as an argument is created; and the **expected** result of the method call is created.
- c. Next we act. We call the method and store the result of that call.
- d. Lastly, we assert—using a function that the JUnit library provides to us—that the actual output of the function is the same as the expected output.

26. Now run the tests (using **./gradlew test**) and verify that no failures occur.

We have a new function, and we've got a test that checks that it works. Let's put this into the DAG.

27. Run **git status** to see what changes have been made. You will see that two files have been modified.

-
28. *How* have the files been modified? Type **git diff** to find out.
- Lines that have a **-** in front of them have been *deleted*.
 - Lines that have a **+** in front of them have been *added*.
 - Notice that when you *change* a line, Git understands this to mean that the previous line has been deleted and a new line has been added in its place.
 - To get back to the command line, press **q**

29. Now let's add these files to the staging area:

git add app/src/main/java/sdp/warmup/App.java

git add app/src/test/java/sdp/warmup/AppTest.java

Note that you could specify *both* files on one command-line. We haven't done that here because Java paths are loooooong and the page width is limited!

30. Type **git status** to check what's in the staging area. You should see both files there.

31. Create a new DAG node with the following command: **git commit -m "Added flatten method and a test for it"**

32. Now run **git log --online --all --graph** and you'll see two commits.

We've now seen how to write and add a test, how to organize a test as Arrange-Act-Assert, how to run tests, how to add and commit files to git, how to ignore files, how to see node-ids and how to ask git about how it sees your files. We're now going to start using branches.

33. Create the file **src/main/java/sdp/warmup/MapFunction.java** which has the following content:

```
package sdp.warmup;

public interface MapFunction {
    Object run(Object x);
}
```

34. Create the file **src/main/java/sdp/warmup/Sum.java** which has the following content:

```
package sdp.warmup;

public class Sum {
    public int first;
    public int second;
    public int result;
    public Sum(int f, int s, int r) {
        first = f;
        second = s;
        result = r;
    }
}
```

35. Add the following function to **App.java**:

```
public Object[] map(Object[] input, MapFunction f) {
    Object[] output = new Object[input.length];
    for (int i = 0; i < input.length; i++) {
        output[i] = f.run(input[i]);
    }
    return output;
}
```

36. Change the **main** function in **App.java** so that it looks like this:

```
public static void main(String[] args) {
    Integer[] numbers = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    App mul = new App();
    Object[] multiplied = mul.map(numbers, i -> mul.map(numbers, j -> new Sum((int)i, (int)j, (int)i * (int)j)));
    for (Object o : multiplied) {
        Object[] p = (Object[])o;
        for (Object q : p) {
            Sum v = (Sum)q;
            System.out.printf("%d x %d = %d\n", v.first, v.second, v.result);
        }
    }
}
```

37. Save the files and make sure that the software builds correctly. Feel free to run it to see what it does!

38. Now we're going to add only *some* of the changes that we did, and we'll leave others out. This is a useful thing because often, you get caught up in creating the software

and forget to commit—so, when you remember, the software might not all be ready.

- a. Add the **MapFunction.java** file to the staging area. Use **git status** to verify that it is in the staging area.
- b. Now, we only want to add the **map** function from the **App.java** file, so let's do that. We will use the command:

git add -p app/src/main/java/sdp/warmup/App.java

👉 the **-p** flag is critical! It tells git that you only want to add *parts* of the file to a staging area.

- i. What happens now is that git asks you whether you wish to “stage” each part of the file¹. You can say “y” for “yes”, “n” for “no”, “s” for “try to split the hunk up a bit more”, “q” for “quit”, and “?” to find out what all the other possible letters mean.
- ii. Use the “y”, “s”, and “n” to stage *only* the **map** function.
- iii. If you make a mistake, don't worry. Just go back to the command line (press “q” if you need to) and unstage those changes by using the command:

git restore --staged file-path

- c. Double-check that only the changes you want to be included are in the staging area. You can do this with the command **git diff --cached**, which tells you which differences from HEAD are in the staging area.
- d. You will notice that when you run **git status**, it tells you that **App.java** is in the staging area *and* in the working tree with uncommitted changes. This is correct, because some changes are in the staging area and some are not.
- e. When you are happy with what you've done, commit with the message "Added a map function".

¹ It calls the changed parts “hunks”.

-
39. Now commit the rest of the changes with the message “Output multiplication tables”. After this commit, there should not be any changes left to commit.

Using arrays in Java can be a painful experience. We probably want to use Lists instead, to be honest. But we have a working program right now; are we sure that it will still be working if we try to replace the arrays with Lists? Let’s go back in the graph and see whether this approach will work out.

40. Find the node-id for the commit with the description “Added a map function”.

Remember that you can do this with **git log --oneline --graph --all**.

41. Now move HEAD to that node: **git switch --detach node-id**

42. Use **git log** again—what do you notice? What does **git status** tell you about HEAD?

43. Create a branch and switch to it: **git switch -c lists-not-arrays**

44. Use **git log** again—what do you notice? And if you type **git status**, which branch does it say that you are on?

45. Now open up **App.java**. Notice that you’ve gone back in time to *before* you made any changes to the **main** function ... and notice, too, that the **Sum.java** file doesn’t exist. These changes *had not been made* when this node of the DAG was committed.

46. For fun, type in **git switch master**—and observe what happens to **App.java**. And look, **Sum.java** is back!

Alright, enough switching. Go back to **lists-not-arrays** (type **git switch lists-not-arrays**) and do some work there.

47. Replace the **flatten** and **map** functions in **App.java** with:

```
public <T> List<T> flatten(List<List<T>>> nested) {
    List<T> result = new ArrayList<T>();
    for (List<T> innerList : nested) {
        for (T item : innerList) {
            result.add(item);
        }
    }
    return result;
}
```

```

}

public List<Object> map(List<Object> input, MapFunction f) {
    List<Object> output = new ArrayList<Object>();
    for (int i = 0; i < input.size(); i++) {
        output.add(f.run(input.get(i)));
    }
    return output;
}

```

48. Save and check that you can build and run successfully. Commit your changes with the message “Switching to lists”.
49. Now take a look at **git log --online --graph --all** and see if you can understand the output. Git has tried to “draw” the graph and nodes using | and * characters... can you see it?
50. Oops—we’ve forgotten about our tests! When we try to run them, they won’t compile. Let’s fix it.
 - a. Near the top of **AppTest.java**, add the line **import java.util.*;**
 - b. Replace the test with this:

```

@Test void flattenUnpacksANestedList() {
    // Arrange
    App classBeingTested = new App();
    List<List<String>> data = Arrays.asList(
        Arrays.asList("hello", "world"),
        Arrays.asList("how", "are", "you"),
        new ArrayList(),
        Arrays.asList("now?")
    );
    List<String> expected = Arrays.asList("hello", "world", "how", "are", "you", "now?");
    // Act
    List<String> result = classBeingTested.flatten(data);
    // Assert
    assertIterableEquals(expected, result);
}

```

- c. Save, run the tests, and commit with the message “Update flatten test”.
51. What does the graph look like now? Run **git log --online --graph --all** to see it.

This “converting arrays to lists” experiment has been successful. Let’s merge it into the master branch.

52. Switch to the master branch.

53. Merge the **lists-not-arrays** branch with the branch that HEAD is pointing to (i.e. **master**): **git merge lists-not-arrays**

54. Take a look at **App.java**: you now have the updated lists code, and the updated **main** function! If the **lists-not-arrays** experiment hadn’t worked out, you could simply switch to **master** and you would have lost nothing.

55. Use **git log** to look at the graph again. Can you see where the newly-merged node is?

56. Replace the **main** function with this one:

```
public static void main(String[] args) {
    List<Object> numbers = Arrays.asList(2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
    App mul = new App();
    List<Object> multiplied = mul.map(numbers, i -> mul.map(numbers, j -> new Sum((int)i, (int)j, (int)i * (int)j)));
    for (Object o : multiplied) {
        List<Object> p = (List<Object>)o;
        for (Object q : p) {
            Sum v = (Sum)q;
            System.out.printf("%d x %d = %d\n", v.first, v.second, v.result);
        }
    }
}
```

57. Save, check that it builds and runs, and then run **git diff**. Notice that although you replaced the whole function, *only the lines that are changed* are highlighted. Stage everything and commit with the message “Updating main for list functions”.

58. We don’t need to keep the **lists-not-arrays** branch around. Delete it with **git branch -d lists-not-arrays**

59. Use **git log** again and notice that all the nodes are still there, but the **lists-not-arrays** branch label isn’t. Deleting a branch just deletes a label that’s attached to a node. Nodes are untouched.

There’s a package that does a very similar thing to the **Sum** class, and if we have that package, then we can get rid of the **Sum** class. Let’s do that.

60. In your **build.gradle** file, just under the line that says
implementation 'com.google.guava:guava:31.0.1-jre'
add this line:

implementation 'org.javatuples'

61. Save and commit your changes with the message “Added JavaTuples dependency”.

62. Now run **./gradlew build**

Whoops, that’s not good. It looks like I’ve given you the wrong instructions. But that’s fine—remember, *we have a time-travel machine!* So let’s use it 😊.

63. Type in **git log --graph --all --oneline** and get the node-id of the last “good” commit (i.e. the one with the description “Updating main for list functions”).

64. Move the HEAD to this node-id: **git switch --detach node-id**

65. Now move the **master** branch to here: **git branch -f master**

66. And now move HEAD to point to **master** again: **git switch master**

67. Take a look at your **build.gradle** file: it’s fixed! Now let’s try a **git log --all --oneline --graph** again. Notice that the node with mistakes has vanished. It’s *still in the graph*, but remember that Git *only* regards a node as important if it’s somewhere along the path of a branch. The node with the mistake is not along the path of a branch any more. Therefore, it’s not shown—and nobody will ever know about our little mistake 😊.

- a. *As long as you have a previous commit to go back to, you never need to fear making a mistake again. You can make as many mistakes as you want, and just jump back in history!*

Really, though, we actually *do* want to use JavaTuples, so this time I’ll give you the correct instructions:

68. In your **build.gradle** file, just under the line that says
implementation 'com.google.guava:guava:31.0.1-jre'

add this line:

implementation 'org.javatuples:javatuples:1.2'

69. Save the file and this time, run **./gradlew build** first. When you're happy that it's working, save and commit² with the message "Added JavaTuples dependency".

70. Near the top of **App.java**, add in **import org.javatuples.*;**

71. Replace the **main** function with:

```
public static void main(String[] args) {
    List<Object> numbers = Arrays.asList(2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
    App mul = new App();
    List<Object> multiplied = mul.map(numbers, i -> mul.map(numbers, j -> new Triplet((int)i, (int)j, (int)i * (int)j)));
    for (Object o : multiplied) {
        List<Object> p = (List<Object>)o;
        for (Object q : p) {
            var v = (Triplet<Integer,Integer,Integer>)q;
            System.out.printf("%d x %d = %d\n", v.getValue0(), v.getValue1(), v.getValue2());
        }
    }
}
```

72. Save and run. Notice that we don't need **Sum** any more—so let's delete it.

73. Type **git rm app/src/main/java/sdp/warmup/Sum.java**

- a. If you type **git status** now, you will see the deletion in the list of changes.
- b. Notice that the file *also* disappears from the working tree.

74. Commit the other changes (using **git add**) with the message "Sum not needed any more".

As a last thing, let's see how far we've come, by extracting the first version of **App.java** from the first commit.

75. Use **git log** to find the node-id of the first commit.

76. Use **git restore --source node-id app/src/main/java/sdp/warmup/App.java** to extract the file and see the total set of changes between the first and last commit.

² I've made you type out filenames explicitly, so far. There's a shortcut. To commit all changes to all tracked files, use **git commit -a -m "some message"**. Of course, if you want to only commit some changes, or you want to add/remove new files, you can't use this shortcut—but for most commits, you can.

-
- a. Note that this file is extracted from the node and copied in-place over your file in the working tree. No DAG modification occurs. From Git's perspective, the result is as if you simply modified the file without interacting with Git at all. Use **git status** and you will see that Git regards it in this way.
77. When you want to go back to the last commit, you can type **git restore app/src/main/java/sdp/warmup/App.java** , which will restore the file to the way it is at the HEAD node.

Part B: Artificial intelligence

In this part, I will create an application using AI. My prompts will be underlined and I will summarise ChatGPT's answers in a block; I might not include all the content because it can sometimes be very verbose. In this section, I encourage you to try the prompts yourself at chat.openai.com . I have used GPT-3.5, which is the same model that you will have access to. *You will not get the same response*, but perhaps you will get something similar!

The key in this part of the warm-up is to pay attention to the *prompting*. I will explain what is behind my thoughts as I prompt, and perhaps that will be helpful to you when you are figuring out your own prompts.

(If you get a message saying that ChatGPT is at "capacity", then you can try all of this later instead.)

78. Good morning, ChatGPT. I would like to create a program that displays a blazing fire. Can you help me with that, please?

Here I start with the *generalities*, and give myself scope to refine as necessary.

Good morning! I'd be happy to help you create a program that displays a blazing fire.

First, it's important to clarify what kind of program you're looking to create. Are you looking to create a graphical program that displays a visual representation of a fire, or are you looking to create a text-based program that simulates a fire using ASCII characters?

If you're looking to create a graphical program, one way to do so is by using a graphics library such as Pygame or Processing. Here's an example of how you could create a basic fire animation using Pygame:

[Python code omitted]

79. Thank you, but I'd like to use Java please. Is that possible?

This is me adding a constraint.

Certainly! Here's an example of how you could create a simple fire animation using Java and the Swing library:

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.util.Random;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class FireAnimation extends JPanel {

    private static final int WIDTH = 640;
    private static final int HEIGHT = 480;

    private static final Color BLACK = Color.BLACK;
    private static final Color RED = new Color(255, 0, 0);
    private static final Color ORANGE = new Color(255, 165, 0);
    private static final Color YELLOW = new Color(255, 255, 0);

    private static final int FIRE_X = WIDTH / 2;
    private static final int FIRE_Y = HEIGHT - 50;

    private static final int FIRE_WIDTH = 80;
    private static final int FIRE_HEIGHT = 120;

    private static final int FLAME_WIDTH = 20;
    private static final int FLAME_HEIGHT = 30;

    private static final int FLAME_SPEED = 5;

    private static final int NUM_FLAMES = 30;
    private static final int[][] flames = new int[NUM_FLAMES][5];

    private static final Random random = new Random();
```

```

public FireAnimation() {
    // Initialize the flames
    for (int i = 0; i < NUM_FLAMES; i++) {
        flames[i][0] = random.nextInt(FIRE_WIDTH) + FIRE_X - FIRE_WIDTH / 2;
        flames[i][1] = random.nextInt(FIRE_HEIGHT) + FIRE_Y - FIRE_HEIGHT;
        flames[i][2] = random.nextInt(FLAME_WIDTH) + FLAME_WIDTH / 2;
        flames[i][3] = random.nextInt(FLAME_HEIGHT) + FLAME_HEIGHT / 2;
        flames[i][4] = random.nextInt(5) + 1;
    }
}

public void update() {
    // Update the flames
    for (int i = 0; i < NUM_FLAMES; i++) {
        flames[i][1] -= FLAME_SPEED;
        flames[i][2] -= 1;
        flames[i][3] -= 1;
        flames[i][4] -= 1;
        if (flames[i][1] < FIRE_Y - FIRE_HEIGHT || flames[i][2] <= 0 ||
flames[i][3] <= 0 || flames[i][4] <= 0) {
            flames[i][0] = random.nextInt(FIRE_WIDTH) + FIRE_X - FIRE_WIDTH / 2;
            flames[i][1] = random.nextInt(FIRE_HEIGHT) + FIRE_Y - FIRE_HEIGHT;
            flames[i][2] = random.nextInt(FLAME_WIDTH) + FLAME_WIDTH / 2;
            flames[i][3] = random.nextInt(FLAME_HEIGHT) + FLAME_HEIGHT / 2;
            flames[i][4] = random.nextInt(5) + 1;
        }
    }
}

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
    // Enable antialiasing
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    // Clear the background
    g2d.setColor(BLACK);
    g2d.fillRect(0, 0, getWidth(), getHeight());
    // Draw the flames
    for (int i = 0; i < NUM

```

80. Note that the code ends in mid-stream! (I didn't notice this at first.)

I'm using gradle as my build tool. Would this still work?

Here I'm asking it whether it can integrate what it's done with the standard tooling.

Yes, you can definitely use Gradle as your build tool for this project. Here's an example of a `build.gradle` file that you could use to build the Java program:

```
plugins {
    id 'java'
}

repositories {
    jcenter()
}

dependencies {
    // Add any required dependencies here
}

jar {
    manifest {
        attributes 'Main-Class': 'FireAnimation'
    }
}

task run(type: JavaExec) {
    classpath sourceSets.main.runtimeClasspath
    mainClassName 'FireAnimation'
}
```

To use this `build.gradle` file, you would need to create a `src/main/java/FireAnimation.java` file with the code for the fire animation. Once you have the `FireAnimation.java` file in place, you can build and run the program with the following commands:

```
./gradlew build
./gradlew run
```

This will build the program and create a runnable JAR file in the `build/libs` directory. The `./gradlew run` command will then run the program from the JAR file.

81. Now we have enough to work with! Create a folder called **fire-animation** , change into it, and create a Java application there by using Gradle:

gradle init --type java-application

82. Create a Git repository in that folder, using **git init** .

83. Now open the folder in VS Code, and look at the build.gradle file (in the **app** folder).

Compare it with what ChatGPT is suggesting. It's quite different! But AI can sometimes get things wrong, so you don't just want to take its word for it.

Fortunately, there's a way to get the best of both worlds. We'll commit to the repository and then use ChatGPT's code. Then if we need to go back at any time, we're easily able to do that.

84. Commit everything with the message "Initial commit".

85. Now replace the contents of the **build.gradle** file with what ChatGPT has suggested. What happens?

86. Yep—the AI hasn't got it quite right. That's fine. Use **git restore** to get you back to where you were, and we'll try to coach ChatGPT onto a better path. Before we do that, though, create the **FireAnimation.java** file in the suggested location and fill in the code.

87. Save it, try to compile it, and an error crops up. The code is incomplete! So let's prompt ChatGPT:

Please continue the Java code?

I apologize for the truncated code in my previous response. Here's the complete code for the Java program that displays a blazing fire:

[exactly the same code, including truncation, as before]

88. Hmm, it seems to be having some trouble here. Let's try a different approach.

Which methods are there in this class? Name them, please.

This Java class **FireAnimation** contains three methods:

1. **update()**: This method is called once per frame and is responsible for updating the position and size of each flame in the animation.
2. **paintComponent(Graphics g)**: This method is called by the Swing framework whenever the panel needs to be redrawn. It is responsible for drawing the background and the flames onto the panel.
3. **FireAnimation()**: This is the constructor for the class, and is responsible for

initializing the flame positions and sizes when the program is first run.

89. I notice that only one of these is missing, so I ask for it.

Write the `paintComponent` method, please.

Certainly! Here's the `paintComponent` method for the `FireAnimation` class:

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
    // Enable antialiasing
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    // Clear the background
    g2d.setColor(BLACK);
    g2d.fillRect(0, 0, getWidth(), getHeight());
    // Draw the flames
    for (int i = 0; i < NUM_FLAMES; i++) {
        int x = flames[i][0];
        int y = flames[i][1];
        int w = flames[i][2];
        int h = flames[i][3];
        int intensity = flames[i][4];
        Color color = null;
        if (intensity == 1) {
            color = YELLOW;
        } else if (intensity == 2) {
            color = ORANGE;
        } else if (intensity == 3) {
            color = RED;
        } else if (intensity == 4) {
            color = BLACK;
        }
        g2d.setColor(color);
        g2d.fillOval(x - w / 2, y - h / 2, w, h);
    }
    // Draw the logs
    g2d.setColor(Color.DARK_GRAY);
    g2d.fillRect(FIRE_X - FIRE_WIDTH / 2, FIRE_Y, FIRE_WIDTH, FIRE_HEIGHT);
}
```

This method is responsible for drawing the background and the flames onto the panel. It first clears the background by filling the panel with black. It then loops through each flame in the animation and draws an ellipse with the appropriate color and size. Finally, it draws the logs at the bottom of the fire with a dark gray color.

90. A `.\gradlew build` now works! But when we run it, nothing happens. So let's ask ChatGPT about that.

Nothing happens when I run this. Explain why?

It's possible that the animation is not running because the `update()` method is not being called. In Swing, you can use a `Timer` to schedule a recurring event that will call the `update()` method and repaint the panel.

Here's how you could modify the `FireAnimation` constructor to add a `Timer`:

```
public FireAnimation() {
    // Initialize the flames
    for (int i = 0; i < NUM_FLAMES; i++) {
        flames[i][0] = random.nextInt(FIRE_WIDTH) + FIRE_X - FIRE_WIDTH / 2;
        flames[i][1] = random.nextInt(FIRE_HEIGHT) + FIRE_Y - FIRE_HEIGHT;
        flames[i][2] = random.nextInt(FLAME_WIDTH) + FLAME_WIDTH / 2;
        flames[i][3] = random.nextInt(FLAME_HEIGHT) + FLAME_HEIGHT / 2;
        flames[i][4] = random.nextInt(5) + 1;
    }
    // Create a Timer to update the animation
    Timer timer = new Timer(1000 / 30, e -> {
        update();
        repaint();
    });
    timer.start();
}
```

This code creates a new `Timer` object that will call the `update()` method and repaint the panel every 1/30th of a second. The `start()` method is then called to start the timer.

With this modification, the animation should now run properly when you run the program.

91. Great, but I don't want to put all that in the constructor; it looks like exactly the wrong place for it, semantically speaking. So I ask it to take a different approach: Instead of what you've done, can you create "main" method code to call `FireAnimation` instead?

Sure! Here's an example `main` method that creates a `FireAnimation` object and

adds it to a **JFrame** window:

```
import javax.swing.JFrame;

public class Main {
    public static void main(String[] args) {
        FireAnimation fireAnimation = new FireAnimation();
        JFrame frame = new JFrame("Fire Animation");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(fireAnimation);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

This code creates a **FireAnimation** object and adds it to a new **JFrame** window with a title of "Fire Animation". It then sets the default close operation, packs the frame to fit the contents, centers the frame on the screen, and makes the frame visible.

You can run this **Main** class to launch the fire animation. Note that you will need to have the **FireAnimation** class defined in the same package or on the classpath.

92. Try replacing the code in **App.java** with this. What do you notice when you compile?

We have errors! Remember, AI isn't great at *integrating* things. We need to help it out.

- a. Change the class name to **App**, because we're in **App.java**.
- b. At the top of the file, put in: **package fire.animation**, just as it was before.
- c. Do the same thing in **FireAnimation.java**.

93. Try to build again, and notice we've still got an error. Oops! Our tests haven't been modified. To fix this, go into the appropriate place and remove the **appHasAGreeting** test.

94. And now we're compiling successfully. But when we **./gradle run**, we just see a blank black window! Perhaps we could try to fix it ourselves at this point, but let's see if the AI can do it for us.

All that I see is a black window, ChatGPT. Could you help me with a fix, please?

If you are seeing a black window, it's possible that the `FireAnimation` object is not being added to the `JFrame` correctly.

One potential issue could be the size of the `FireAnimation` panel. If the panel is not given a preferred size, it will default to a size of 0x0, which could cause it to be invisible in the `JFrame`.

To fix this, you can override the `getPreferredSize()` method in the `FireAnimation` class to return the desired size of the panel. Here's an example of how you could modify the `FireAnimation` class to do this:

```
public class FireAnimation extends JPanel {  
  
    // Other code here...  
  
    @Override  
    public Dimension getPreferredSize() {  
        return new Dimension(WIDTH, HEIGHT);  
    }  
  
    // Other code here...  
  
}
```

This code overrides the `getPreferredSize()` method to return a `Dimension` object with the width and height of the panel. This ensures that the `JFrame` will give the panel the correct size.

If this doesn't solve the issue, there may be another issue with the code. Please let me know if this solution works, or if you need further assistance.

95. Let's try that suggestion and run it again.

And now we have something a bit better! But there's still a problem. Can you see it?

96. Here's a prompt that describes the problem:

Thanks, ChatGPT. I'm now seeing the very first frame of an animation, but it doesn't move. Can you help me with that?

If you're seeing the first frame of the animation but it doesn't move, it's likely that the `update()` method is not being called correctly.

In the Swing framework, you can use a `Timer` to schedule a recurring event that calls the

`update()` method and repaints the panel. You can modify the `FireAnimation` constructor to add a `Timer` like this:

```
public FireAnimation() {
    // Initialize the flames
    for (int i = 0; i < NUM_FLAMES; i++) {
        flames[i][0] = random.nextInt(FIRE_WIDTH) + FIRE_X - FIRE_WIDTH / 2;
        flames[i][1] = random.nextInt(FIRE_HEIGHT) + FIRE_Y - FIRE_HEIGHT;
        flames[i][2] = random.nextInt(FLAME_WIDTH) + FLAME_WIDTH / 2;
        flames[i][3] = random.nextInt(FLAME_HEIGHT) + FLAME_HEIGHT / 2;
        flames[i][4] = random.nextInt(5) + 1;
    }
    // Create a Timer to update the animation
    Timer timer = new Timer(1000 / 30, e -> {
        update();
        repaint();
    });
    timer.start();
}
```

This code creates a new `Timer` object that will call the `update()` method and repaint the panel every 1/30th of a second. The `start()` method is then called to start the timer.

With this modification, the animation should now run properly when you run the program. Let me know if you have any further issues!

97. Let's put it in and see if it builds... and no, it doesn't. There's no **Timer** class defined.

But ChatGPT has already told us that the **Timer** comes from the Swing framework, so we can just write **`import javax.swing.Timer;`** at the top of the file.

And just like that, we have a basic and rudimentary fire animation!

Practical

(There is no practical hand-in, this time!)

Homework

Task 1

Collaborate with others using a shared GitHub repository.

1. Go to [this link](#) and fill out your GitHub details (if you haven't already done so).
2. If you have been assigned **Task 1** in your group, do the following steps:
 - a. Sign in to GitHub.
 - b. Go to <https://github.com/cynic/sdp-prac2> and **fork** the repository. To “fork” a repository means to make your own independent copy of it. After forking, if you go to <https://github.com/your-username-here/sdp-prac2>, you will be looking at your own copy of the remote repository.
 - c. Click on the **Settings** link on the gray background. On the settings page, click on **Collaborators** from the left-side panel. (you may be asked to re-enter your password).
 - d. Use the usernames from the sheet to add *all members in your group* (except, of course, for you) as collaborators.
3. If you have NOT been assigned **Task 1** in your group, do the following steps:
 - a. Sign in to GitHub.
 - b. Wait to receive a link from GitHub that invites you to collaborate on a GitHub repository. Follow the instructions to accept the invitation and begin collaborating.
4. Get the complete DAG from the remote repository. Write down the command that you used.
5. Complete your assigned task, working in the **SimpleFunctions.java** file. Complete your task on your own branch, which should have your student number somewhere in the branch name.

Task #	Description of task
1	Complete the method Task1 , which accepts lists of integers a and b , and adds up the elements from a whose indexes are specified by b . If any index in b is out-of-range for a , ignore that index.
2	Complete the method Task2 , which removes the first character from

	every element in a list. If an element has no characters, remove it from the output.
3	Complete the method Task3 , which returns true only if there are a matching number of (and) brackets in a string, and those brackets are nested correctly.
4	Complete the method Task4 , which accepts lists of integers a and b , and multiplies the first element of a with the last element of b , then the second element of a with the second-last element of b , and so on. The resulting list is returned. If lists are different sizes, return null.
5	Complete the method Task5 , which returns true if the list is sorted.
6	Complete the method Task6 , which rounds up each number in the provided integer list to the next-highest multiple of 100. If a number is already a multiple of 100, it is left unchanged. Do not modify the list that is passed; instead, return a new list.

6. Push your branch to the remote. Write down the command that you used.
7. Fetch **master** from the remote. Write down the command that you used.
8. Merge your changes into **master**. Write down the command that you used.
9. Push **master** to the remote.
10. Write at least two tests for the person whose **Task #** is *directly after* yours. These tests must be committed to the **master** branch. (if you are the last person in the group, you will write a test for the person whose task is Task 1). The tests should follow FIRST principles and the Arrange-Act-Assert form.
11. Wait for tests to be written for your task. If your task does not pass all the tests, then
 - a. Change back to your branch. Write down the command that you used.
 - b. Extract the tests into your own branch. Write down the command that you used.
 - c. Modify the test file to only have those tests.
 - d. Fix your task code so that it passes the tests correctly.

-
- e. Commit your task code to your branch. Do not commit the extracted test code. Write down the command that you used.
 - f. Fetch **master** from the remote, and merge your changes into **master**. Write down the command that you used.
 - g. Push **master** to the remote.
12. If your task *does* pass all the tests, then write down the commands you *would* have used if you had to do (a), (b), (e), (f), and (g) above.
13. Add a new remote, with the name **original**, which points to <https://github.com/cynic/sdp-prac2> . Write down the command that you used.

Delete the **.gradle**, **app/bin**, and **app/build** folders (if they exist). Zip up your whole folder (which *must* include your **.git** folder). Submit your zipfile and answers via RUConnected.

Task 2

Changing data from one format to another is a popular Computer Science (and business!) pastime. An XML file has been uploaded to RUConnected. **Using AI**, and **committing to a repository** after each step,

1. Create a Java application to read this file and print out the field values.
2. Change your Java application to only output user-selected fields from that XML file.
3. Now change it to output the user-selected fields in JSON format.
4. Now include enough validation so that your program does not crash if the user types in something unexpected.
5. Now switch to using the SAX XML parser instead.

Push your repository to GitHub, using the repository name **tasktoo**, and ensure that it is public.

Submit a file containing the URL of your **tasktoo** GitHub repository, and all of the prompts that you used to achieve the above.