Lab Report Dennis Torsten Bröcker / Stepan Burlachenko - 07.05.19

1. Semester / IMI - WH C 579

Professor	Name of Exercise
Debora Weber-Wulff	Laboratory 3: Rock around the Clock

General Information

Assignment

1. Read the `ClockDisplay` code in order to see how it uses the `NumberDisplay` class that we spoke about in the lecture. Open up BlueJ and play with it to see how it works. Explain in your report in complete sentences how the hours increase.
2. Now adapt the `ClockDisplay` to display the time American-style (i.e. 12-hour clock and am / pm). You will have to include the Strings "am" or "pm" in the display! I strongly suggest researching how American time is kept before you start programming.
3. There are at least two ways in which you could have implemented exercise 2—one keeps the time internally as a 24-hour clock and adapts the output, the other keeps the time internally as it is displayed. Whichever way you chose for exercise 3, now implement the other in a new class. Which one was better? Why?
4. Make your clock into an alarm clock by adding an alarm. You should be able to set the alarm time and turn the alarm on and off. When the clock reaches the alarm time, it should ring—writing "Riiiiiiiing!" to the terminal is sufficient.
5. Use the MusicPlayer from MusicOrganizer to play some music(or a ringing sound) instead of writing to the terminal

Lab Report

Expectation & Plan

Dennis expected that the exercise was to improve his coding-skills and to practice. Also, at the first glance, it looked like an easy task to him but at the end, he really underestimated the exercise but also was enjoying that kind of task, since It was more difficult than the previous exercises.

Stepan's main goal was seeing in action how a class can be dependent on and use methods of other class.

Report

1. Task (15 minutes)

The ClockDisplay class is dependent on the NumberDisplay class. The NumberDisplay class has a method called increment. It takes no parameters and calculates the modulo of the current value + 1. So, for example, if we are increasing the minutes, while the limit is 60, it makes the increment of value 45 to 46 ((45 + 1) % 60 = 46). The ClockDisplay class uses the increment method of the NumberDisplay class as a part of its own method called timeTick. In this method, the hours only increment if the value of minutes reaches 0 (when the modulo becomes 0 as in (59 + 1) % 60). Another particular thing is that this method (hours.increment) is an external method call and consists of the object name, a dot and the method's name.

2. Task (50 minutes)

In order to change just the output, in this task, we focused only on the updateDisplay method.

First of all, we initialized a new local variable in that method: int hour, which gets the value from “hours.getValue()” (the current hour), and another local variable named ‘ampm’ of type String which assigned “am” as its value (Lines 69-70 of Appendix 1 code).

When the clock is in the 24-hour format and, say, it is 13 o´clock, it should subtract 12 from that value in order to convert it into the 12-hour format. In this example with 13 o’clock, now that we have 1 as the new value for hour, the ampm variable should get a new string-value “pm”. This is achived by the if-statement that subsctracts 12 from the hour variable if it equals or is bigger than 12, and changes the ampm variable to pm (Lines 77-80 of Appendix 1 code).

Now, if you enter 24 as the hour value, it hits the limit and gets rolled back to 0. But 0 is not a valid number in the 12-hour format clock. So that value gets set to 12, using the if condition and gets the am string, since it is not bigger than 12. But if you enter 12, it will still subtract 12, which equals 0 and changes the ampm value to “pm”. In the other if, it gets set back to 12.

To achieve this, we included another if-statement that assigned the value of 12 to the hour variable this variable equals 0 (Lines 90-92 of Appendix 1 code).

Now, the final step was making the displayString method show the new hour value and the am/pm (Line 93 of Appendix 1 code).

Another little modification we did was not showing 0 in the hour variable if it is a one-digit variable. We did it this way because we did additional research on the internet and saw a lot of images of clocks not having a 0 displaying the hour, hence the decision.

The main challenge of this task was not following the instructions entirely: we were sure that there is still 00:00 in the 12-hour clock format, and only 12:00 pm. Only upon some additional googling we realized our wrong way of thinking. This cost us a good half an hour.

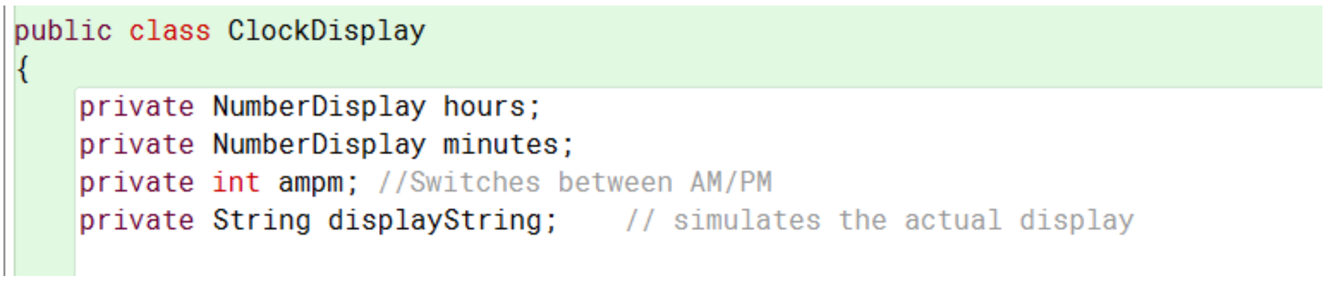
3. Task (20 minutes)

For the 12-hour format clock that keeps the time internally, we had to create a whole new class (named ClickDisplayIntern12), which we based on the original 24-hour format click class, because we wanted to keep a good overview and did not want to “destroy” something, which is working. There, we introduced a new field of type boolean named am. In the constructors, the hours value has a rollOverLimit of 12 instead of 24. In any new instance of this calls, the value of the am field is always set to am (Lines 3-6 of Appendix 2 code).

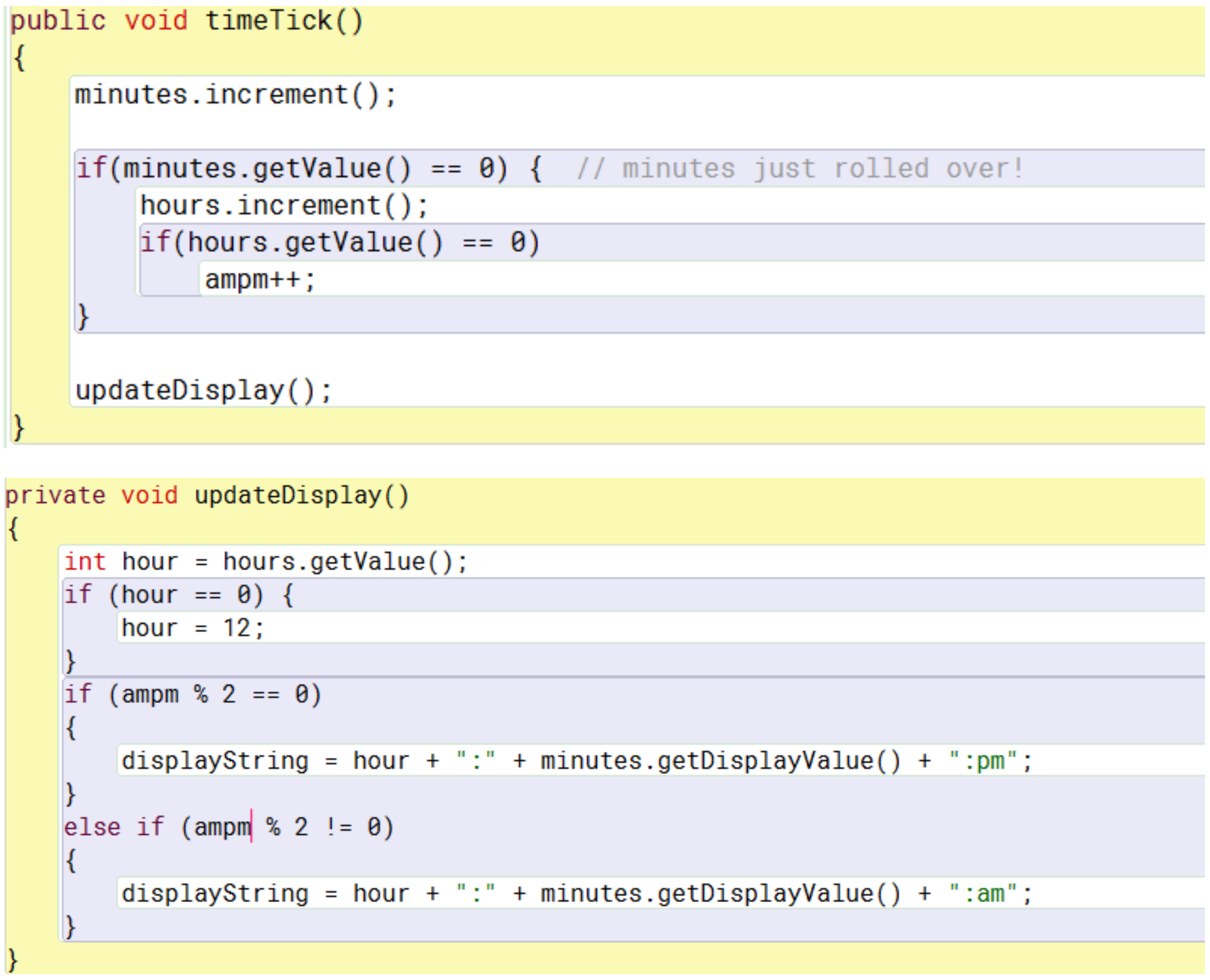
Now, the tricky part was making the am switch to pm after the midnight. Here, we modified the timeTick method which uses the increment method of the NumberDisplay class. In this version, we had to change the hours.incrment part so that depending if the boolean type value am is true or false it switches to the opposite (Lines 41-46 of Appendix 2 code).

Finally, for the updateDisplay method we introduced a conditional statement where am is displayed when the am value is true and pm when the am value is false (Lines 82-86 of Appendix 2 code) .

A little detour that we had here was using a new field of type int instead of boolean. In this version, we'd have to deal with a modulo (Screenshot 1). The downside of this was that whenever you set the time by hand, the suffix is the one, which currently is, and you couldn't change it yet. That problem could have been solved by adding a new parameter like "String ampm" to the constructor. If "am" is entered, the value of int ampm gest set to 1, so it triggers the ampm%2!=0 condition in updateDisplay and if "pm" is entered, the value of int ampm gets set to 2, so it triggers the ampm%2==0, which makes the updateDisplay() output pm. Same goes for the setTime() method. To protect that parameter from the wrong input, we could have added an else-condition after the other 2 conditions, where it outputs an error for an invalid string and sets the value ampm to 1 to make it am automatically. However, we sticked with the Boolean-solution since it was easier to understand and implement in general.



```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private int ampm; //Switches between AM/PM
    private String displayString;    // simulates the actual display
```



```
public void timeTick()
{
    minutes.increment();

    if(minutes.getValue() == 0) {  // minutes just rolled over!
        hours.increment();
        if(hours.getValue() == 0)
            ampm++;
    }

    updateDisplay();
}
```

```
private void updateDisplay()
{
    int hour = hours.getValue();
    if (hour == 0) {
        hour = 12;
    }
    if (ampm % 2 == 0)
    {
        displayString = hour + ":" + minutes.getDisplayValue() + ":pm";
    }
    else if (ampm % 2 != 0)
    {
        displayString = hour + ":" + minutes.getDisplayValue() + ":am";
    }
}
```

Screenshot 1: using an int type field switches

4. Task (20 minutes)

Here, we had to create another class, named it ClocIDisplayAlarm24. It’s values, in addition to the current time (hours and minutes), were the alarmHours and alarmMinutes fields of type int (for setting the alarm), the alarmStatus field of type boolean (which is true when there is an alarm set) and the displayString field of type String (to stimulate the actual display) (Lines 3-8 of Appendix 3 code).

Then, a new method named setAlatmOn was created. It is a setter and takes two parameters of type int: hour and minute. These parameters are assigned to the values of the alarmHours and alarmMinutes fields correspondingly. Plus, the value of the alarmStatus field is sets to true (Lines 52-56 of Appendix 3 code). Another new method called setAlarmOff sets the value of the alarmStatus to false, thus turning off the alarm (Lines 58-60 of Appendix 3 code).

The final step was printing to the terminal the corresponding message when the alarmHours and the alarmMinutes values equal the values of hours and minutes. This was implemented into the timeTick method using a new if-statement (Lines 44-48 of Appendix 3 code).

5. (For the bored) (Dennis only: without Stepan)

For that assignment we created a new project where we have the classes of MusicOrganizer and our AlarmClock in one project. In general, we worked with inheritance, since it is a simple way to get the attributes and methods over to the class ClockDisplayAlarm24. In that case ClockDisplayAlarm24 was the subclass and MusicOrganizer the superclass. Moreover, we needed to create a folder "audio" and put some songs in it to play. When the instance of ClockDisplayAlarm24 is created, it reads the folder automatically in order to find songs and confirms it by outputting a message into the terminal. We did not delete that message, because it is good to have a confirmation if things work or not.

All we did in order to get the sound get played was to replace the line of the “RIING” output (Line 47 of Appendix 3 code) to playTrack(0); . Now it played the first song, which is in the ArrayList, when the alarm gets triggered. We also noticed at that point, that we forgot to make the alarm turning off after it triggered one time. So, we added in that scope “alarmstatus = false;” (picture 2 of Appendix 4)

Now we were thinking of choosing our own song for the alarm. First of all, you need to have more than one song in the audio folder. In the setAlarmOn() method we added a new parameter for this called Index, which is the index number of the song in the ArrayList(picture 3 of Appendix 4). In order to get the track index, you can open the musicplayer and use the method “listAllTrack()”, which we upgraded by showing now the index number and not just the details(picture 4 of Appendix 4).

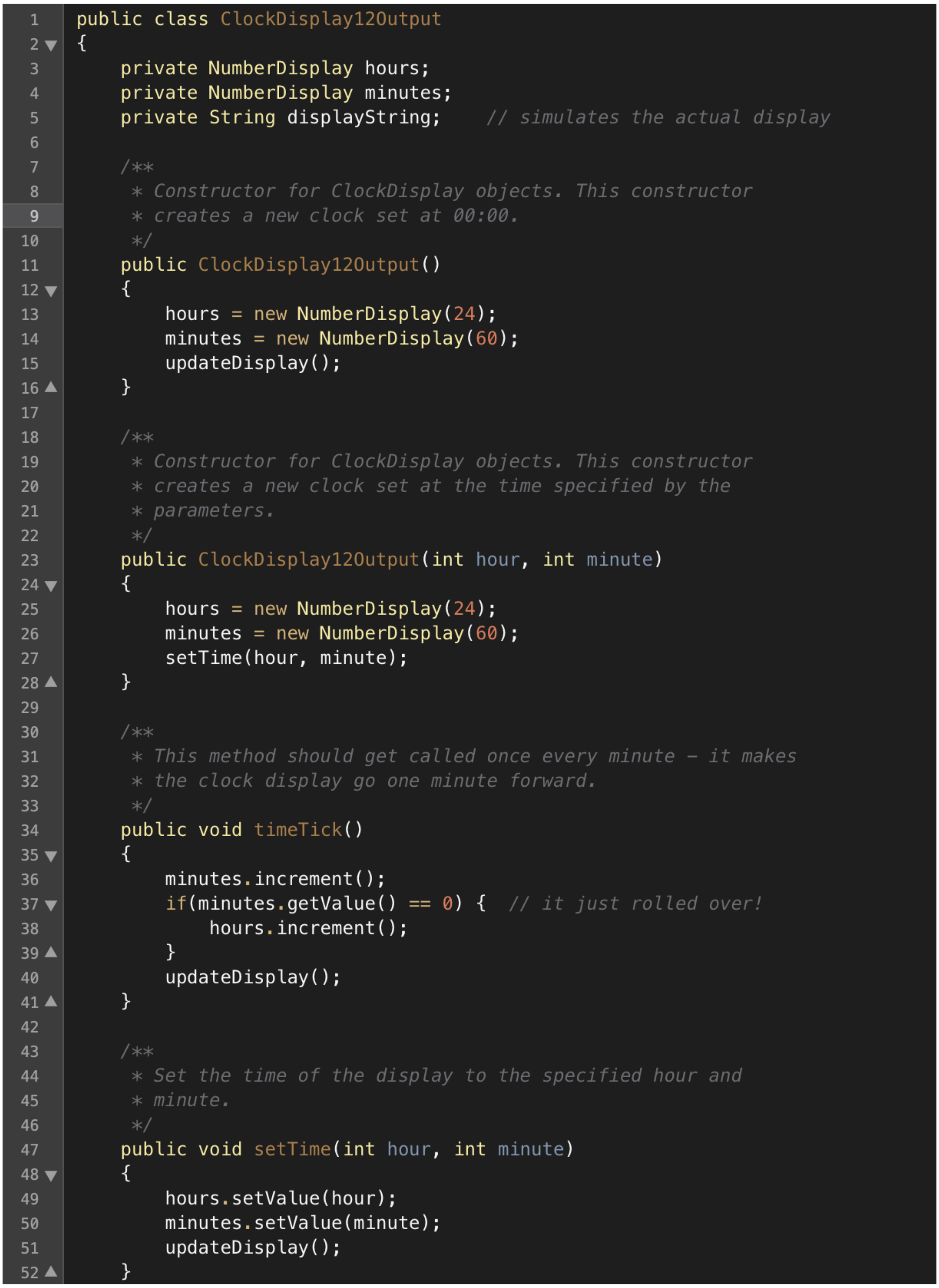
Reflection & Summary of the Lab

As Dennis already said in the expectations, he really enjoyed the exercise. It was more challenging than the previous ones and one can recognize, that the coding gets more and more advanced. Also, Dennis thinks Stepan was a nice partner, who helped him a lot in bringing him to new ideas; thanks to Stepan we came to the idea of using Boolean for the 12-hour-format task. Moreover, Dennis thinks we both learned, that we should not underestimate the 12-hour-format since we wasted the whole lab on the first task, because we were thinking that an American clock has a 0 as hour-output.

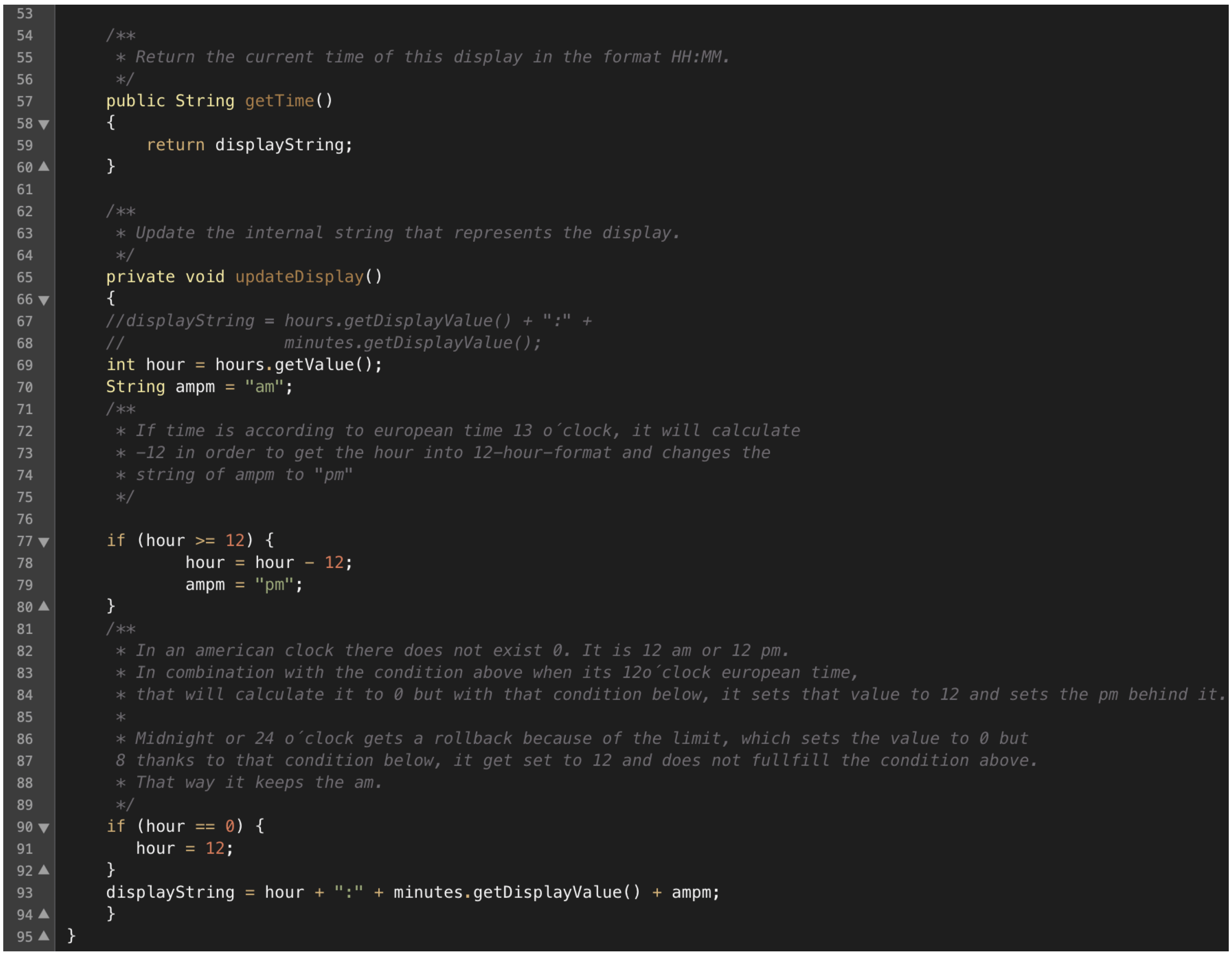
Stepan shares Dennis’ views on the tasks becoming more challenging and thinks that if not the for the ignorance on the way American clock works, they would have done everything in class instead of collaborating after-class. Reading the task carefully and following the instructions closely is a must.

Appendix

Appendix 1 – ClockDisplay12Output code

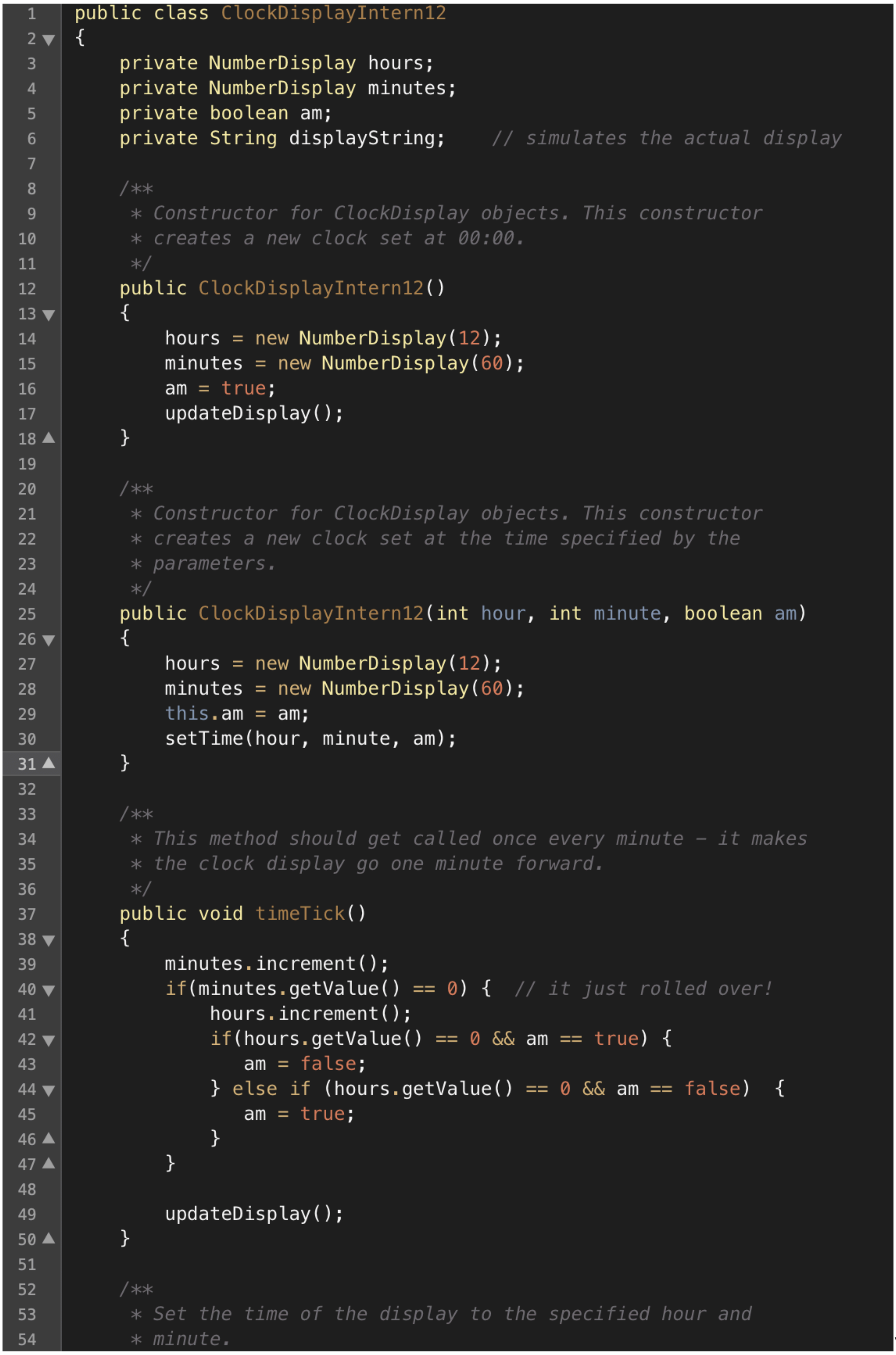


```java
public class ClockDisplay12Output
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplay12Output()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at the time specified by the
     * parameters.
     */
    public ClockDisplay12Output(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }

    /**
     * This method should get called once every minute - it makes
     * the clock display go one minute forward.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) {  // it just rolled over!
            hours.increment();
        }
        updateDisplay();
    }

    /**
     * Set the time of the display to the specified hour and
     * minute.
     */
    public void setTime(int hour, int minute)
    {
        hours.setValue(hour);
        minutes.setValue(minute);
        updateDisplay();
    }
```

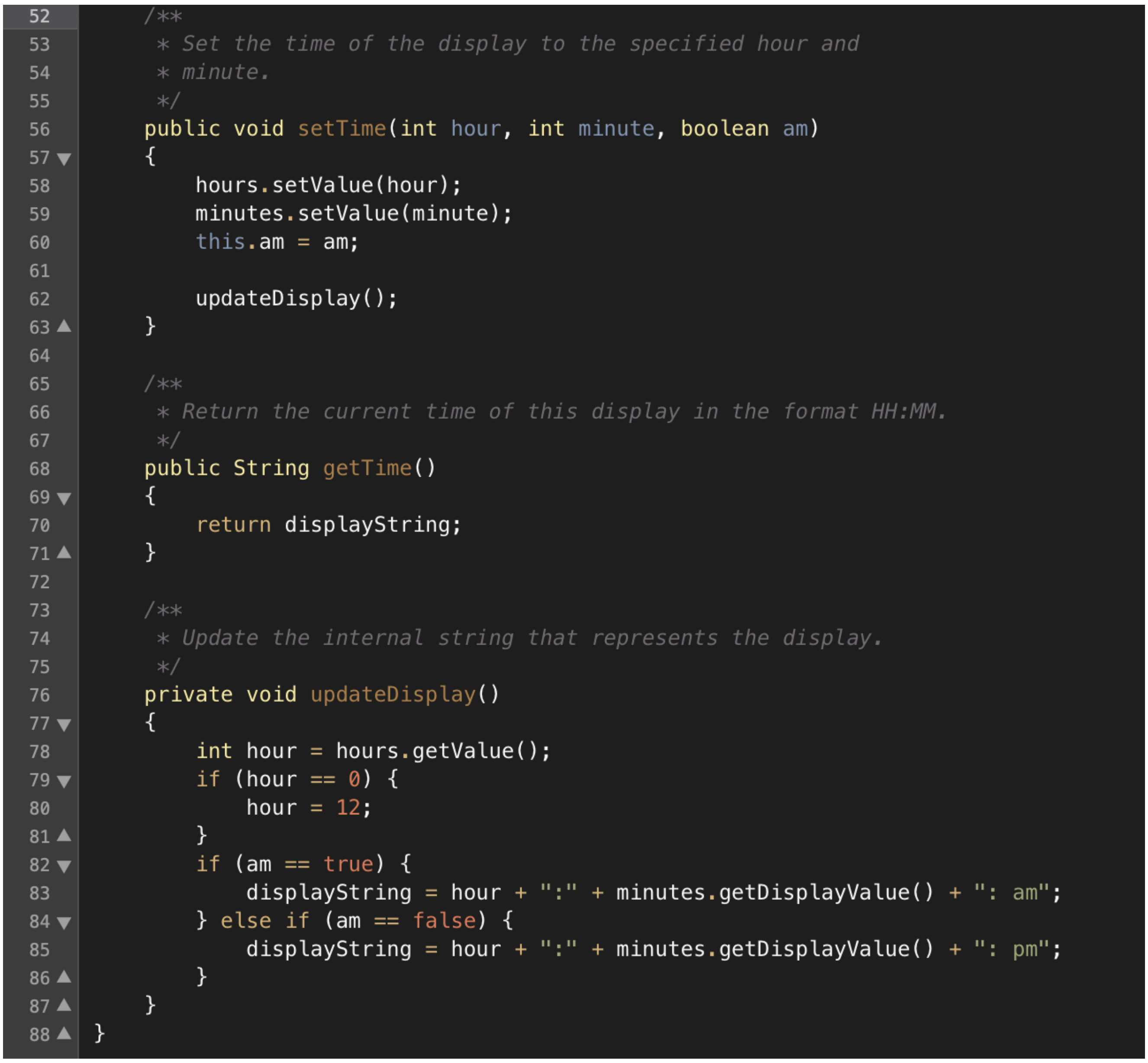


```java

    /**
     * Return the current time of this display in the format HH:MM.
     */
    public String getTime()
    {
        return displayString;
    }

    /**
     * Update the internal string that represents the display.
     */
    private void updateDisplay()
    {
    //displayString = hours.getDisplayValue() + ":" +
    //                minutes.getDisplayValue();
    int hour = hours.getValue();
    String ampm = "am";
    /**
     * If time is according to european time 13 o´clock, it will calculate
     * -12 in order to get the hour into 12-hour-format and changes the
     * string of ampm to "pm"
     */

    if (hour >= 12) {
            hour = hour - 12;
            ampm = "pm";
    }
    /**
     * In an american clock there does not exist 0. It is 12 am or 12 pm.
     * In combination with the condition above when its 12o´clock european time,
     * that will calculate it to 0 but with that condition below, it sets that value to 12 and sets the pm behind it.
     *
     * Midnight or 24 o´clock gets a rollback because of the limit, which sets the value to 0 but
     8 thanks to that condition below, it get set to 12 and does not fullfill the condition above.
     * That way it keeps the am.
     */
    if (hour == 0) {
       hour = 12;
    }
    displayString = hour + ":" + minutes.getDisplayValue() + ampm;
    }
}
```

Appendix 2 – ClockDisplayIntern12 code

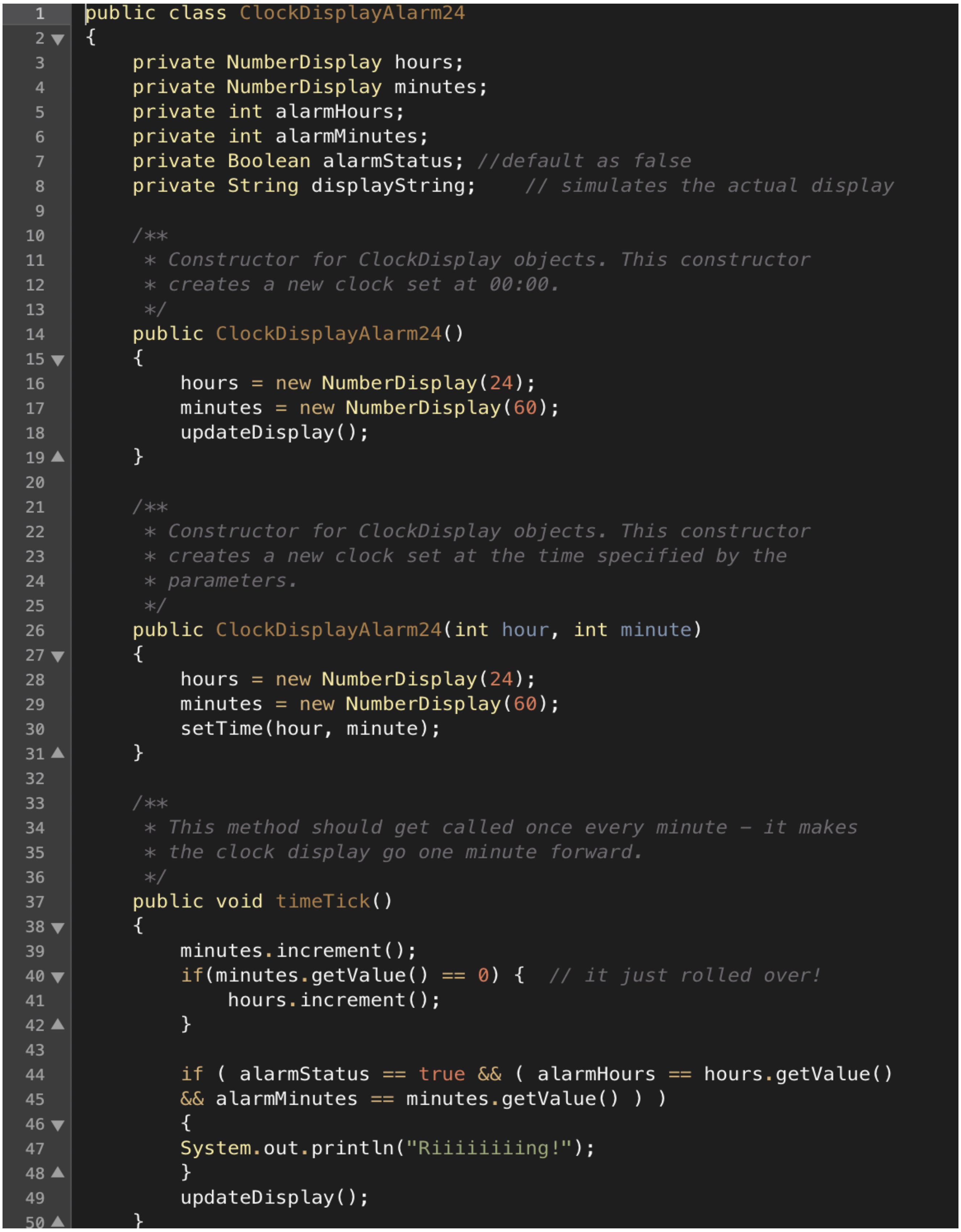


```java
public class ClockDisplayIntern12
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private boolean am;
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplayIntern12()
    {
        hours = new NumberDisplay(12);
        minutes = new NumberDisplay(60);
        am = true;
        updateDisplay();
    }

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at the time specified by the
     * parameters.
     */
    public ClockDisplayIntern12(int hour, int minute, boolean am)
    {
        hours = new NumberDisplay(12);
        minutes = new NumberDisplay(60);
        this.am = am;
        setTime(hour, minute, am);
    }

    /**
     * This method should get called once every minute - it makes
     * the clock display go one minute forward.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) {  // it just rolled over!
            hours.increment();
            if(hours.getValue() == 0 && am == true) {
               am = false;
            } else if (hours.getValue() == 0 && am == false)  {
               am = true;
            }
        }

        updateDisplay();
    }

    /**
     * Set the time of the display to the specified hour and
     * minute.
```
\

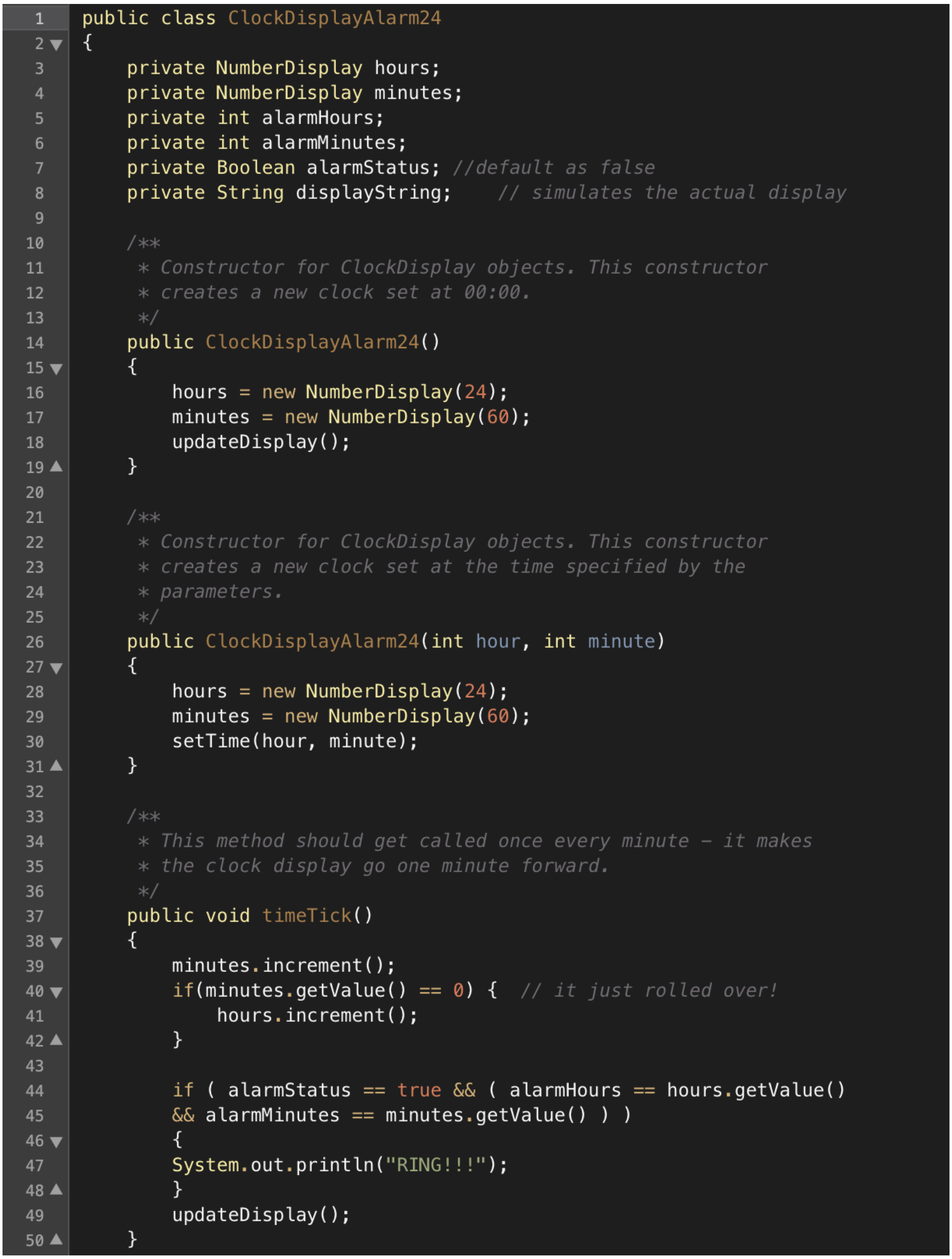


```java
    /**
     * Set the time of the display to the specified hour and
     * minute.
     */
    public void setTime(int hour, int minute, boolean am)
    {
        hours.setValue(hour);
        minutes.setValue(minute);
        this.am = am;

        updateDisplay();
    }

    /**
     * Return the current time of this display in the format HH:MM.
     */
    public String getTime()
    {
        return displayString;
    }

    /**
     * Update the internal string that represents the display.
     */
    private void updateDisplay()
    {
        int hour = hours.getValue();
        if (hour == 0) {
            hour = 12;
        }
        if (am == true) {
            displayString = hour + ":" + minutes.getDisplayValue() + ": am";
        } else if (am == false) {
            displayString = hour + ":" + minutes.getDisplayValue() + ": pm";
        }
    }
}
```

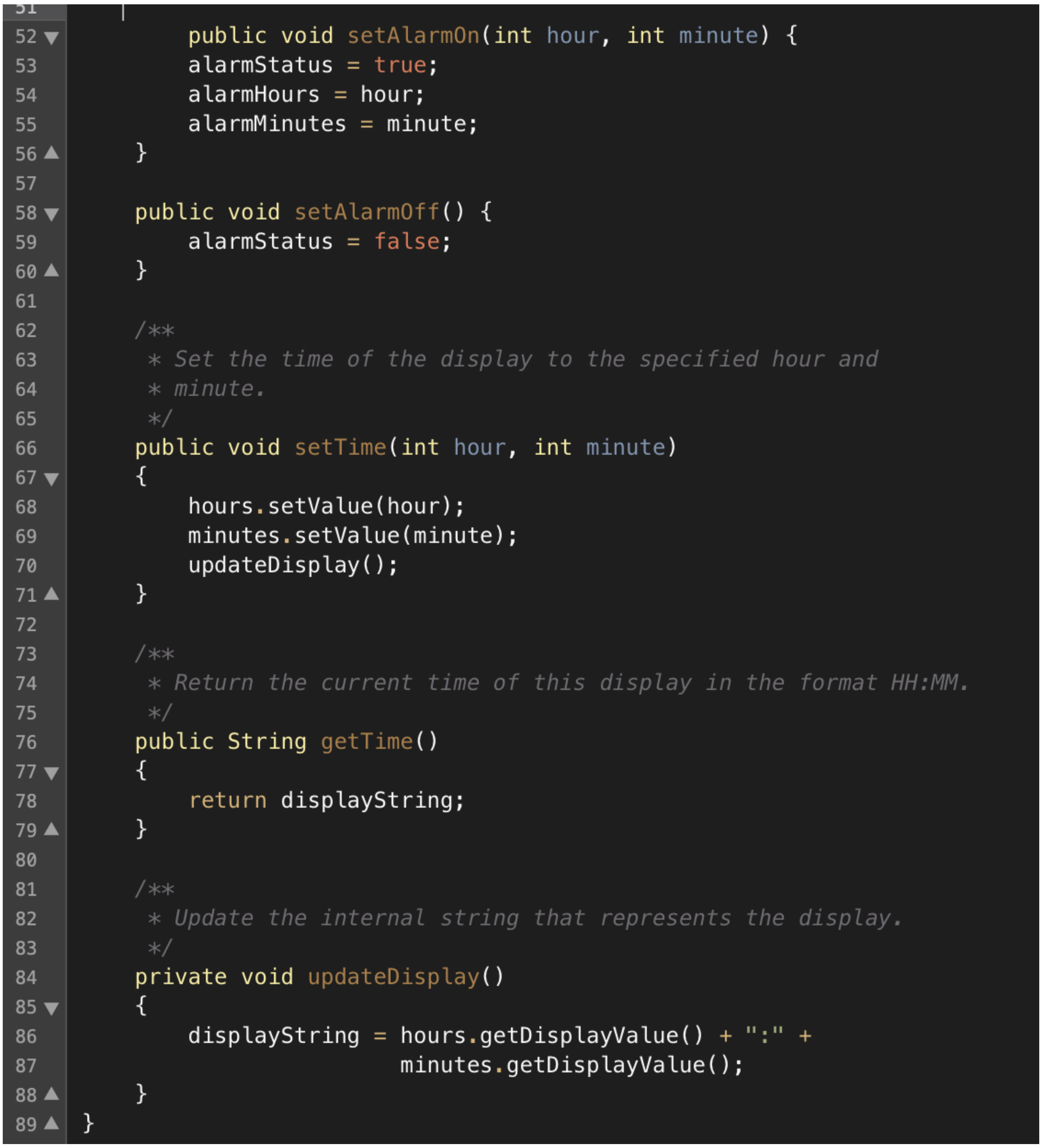
Appendix 3 – ClockDisplayAlarm24 code



```java
public class ClockDisplayAlarm24
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private int alarmHours;
    private int alarmMinutes;
    private Boolean alarmStatus; //default as false
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplayAlarm24()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at the time specified by the
     * parameters.
     */
    public ClockDisplayAlarm24(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }

    /**
     * This method should get called once every minute - it makes
     * the clock display go one minute forward.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) {  // it just rolled over!
            hours.increment();
        }

        if ( alarmStatus == true && ( alarmHours == hours.getValue()
        && alarmMinutes == minutes.getValue() ) )
        {
        System.out.println("Riiiiiiing!");
        }
        updateDisplay();
    }
```

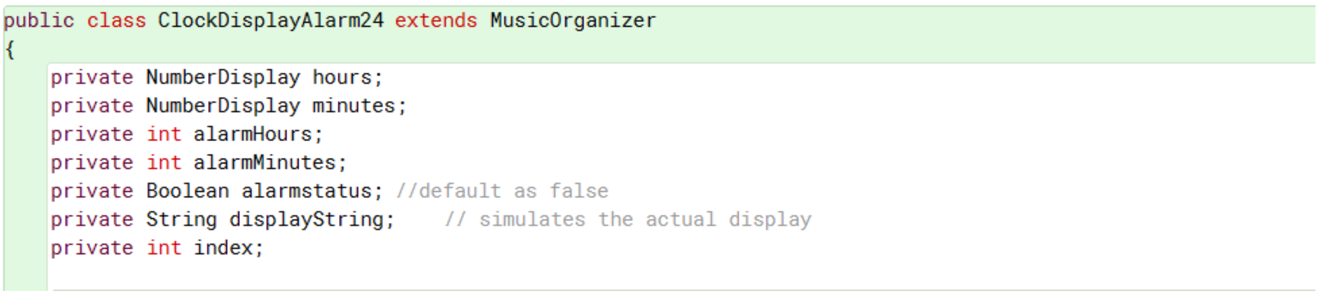


```java
public class ClockDisplayAlarm24
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private int alarmHours;
    private int alarmMinutes;
    private Boolean alarmStatus; //default as false
    private String displayString;    // simulates the actual display

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplayAlarm24()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    /**
     * Constructor for ClockDisplay objects. This constructor
     * creates a new clock set at the time specified by the
     * parameters.
     */
    public ClockDisplayAlarm24(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }

    /**
     * This method should get called once every minute - it makes
     * the clock display go one minute forward.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) {  // it just rolled over!
            hours.increment();
        }

        if ( alarmStatus == true && ( alarmHours == hours.getValue()
        && alarmMinutes == minutes.getValue() ) )
        {
        System.out.println("RING!!!");
        }
        updateDisplay();
    }
```



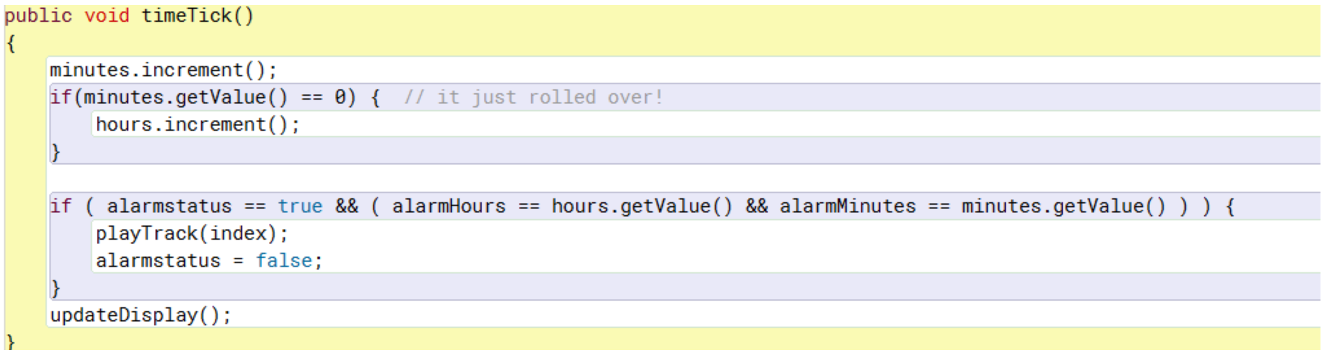
```java
        public void setAlarmOn(int hour, int minute) {
        alarmStatus = true;
        alarmHours = hour;
        alarmMinutes = minute;
    }

    public void setAlarmOff() {
        alarmStatus = false;
    }

    /**
     * Set the time of the display to the specified hour and
     * minute.
     */
    public void setTime(int hour, int minute)
    {
        hours.setValue(hour);
        minutes.setValue(minute);
        updateDisplay();
    }

    /**
     * Return the current time of this display in the format HH:MM.
     */
    public String getTime()
    {
        return displayString;
    }

    /**
     * Update the internal string that represents the display.
     */
    private void updateDisplay()
    {
        displayString = hours.getDisplayValue() + ":" +
                        minutes.getDisplayValue();
    }
}
```

Appendix 4 – ClockDisplayAlarm24(with sound)



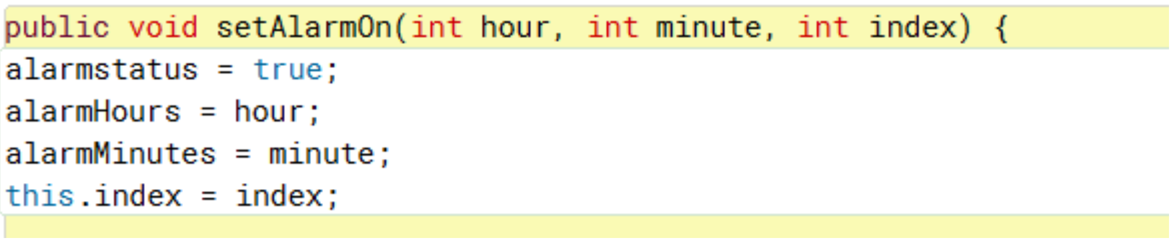
```
public class ClockDisplayAlarm24 extends MusicOrganizer
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private int alarmHours;
    private int alarmMinutes;
    private Boolean alarmstatus; //default as false
    private String displayString;    // simulates the actual display
    private int index;
```

1 new fields of the method



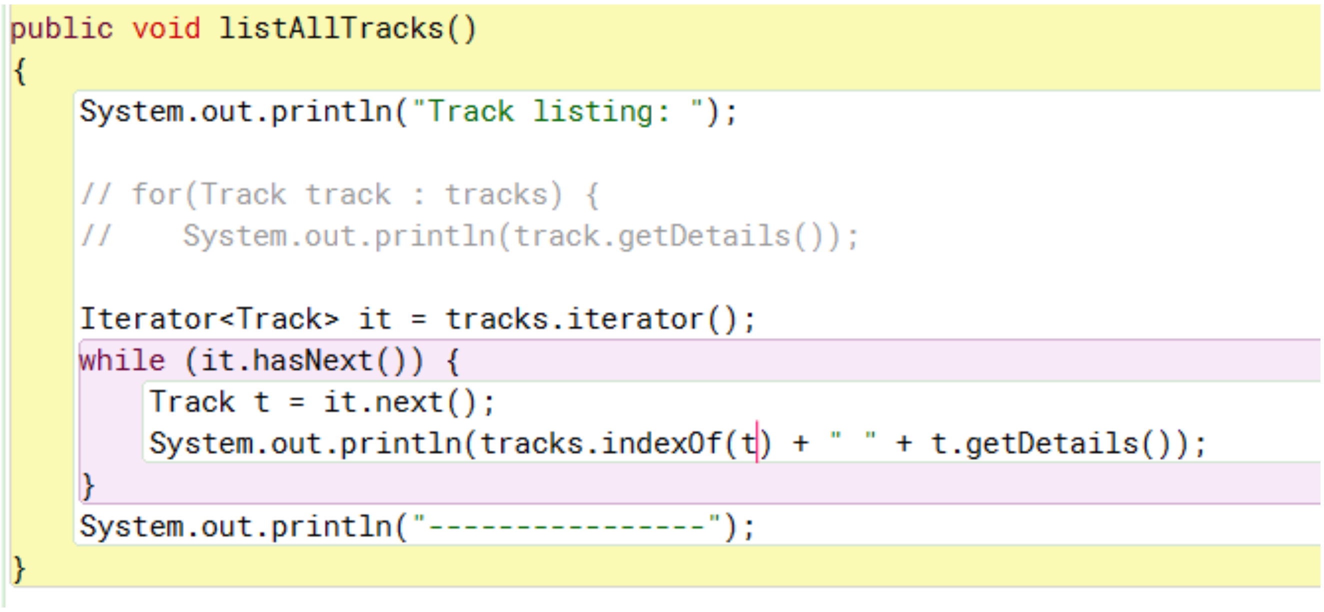
```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {  // it just rolled over!
        hours.increment();
    }

    if ( alarmstatus == true && ( alarmHours == hours.getValue() && alarmMinutes == minutes.getValue() ) ) {
        playTrack(index);
        alarmstatus = false;
    }
    updateDisplay();
}
```

2 updated timeTick() method



```
public void setAlarmOn(int hour, int minute, int index) {
alarmstatus = true;
alarmHours = hour;
alarmMinutes = minute;
this.index = index;
```

3 updated setAlarmOn() method



```
public void listAllTracks()
{
    System.out.println("Track listing: ");

    // for(Track track : tracks) {
    //     System.out.println(track.getDetails());

    Iterator<Track> it = tracks.iterator();
    while (it.hasNext()) {
        Track t = it.next();
        System.out.println(tracks.indexOf(t) + " " + t.getDetails());
    }
    System.out.println("----------------");
}
```

4 updated listAllTracks() method in MusicOrganizer