Send comments or corrections to alan.walworth@gmail.com.

Practical Java

Alan Walworth

Getting Off the Ground

The following tutorial shows how to develop useful Java software without knowing much Java, which is to say without having memorized lots of details about Java. Instead of learning Java details, you will learn techniques for accomplishing what you want without detailed knowledge of Java. You do need to know some basics, such as that '=' is used to assign a value and '==' is used to test for equality. You should know what a declaration is, understand the special role of the *main* method, realize statements are ended with semicolons, and so on. In addition, you should have a rudimentary knowledge of how to use Eclipse, the standard Java development environment. If you don't have it, you can download it for free from eclipse.org (search for "eclipse download"). If you don't know them already, the basics of Eclipse and Java are easy to learn from online tutorials, so I won't dwell on them here.

I want to explain frankly my own experience with this exercise. The first time I did it, I started out knowing Java basics like ending statements with semicolons, but I had no idea how to write the code for a timer in Java. I was going to say I had no idea how to implement a timer in Java, but that's not exactly the same thing. As you'll see, by going through the steps described in detail below, I managed to achieve the desired result by finding examples of code that would do something close enough to what was wanted to be useful, and assembling and fixing such code as needed. The first time I did this exercise, it took several hours, because I started with no idea how to accomplish the task, other than recognition that I should use the general method illustrated below. So I took wrong turns. But I persevered and got it to work, and you can too.

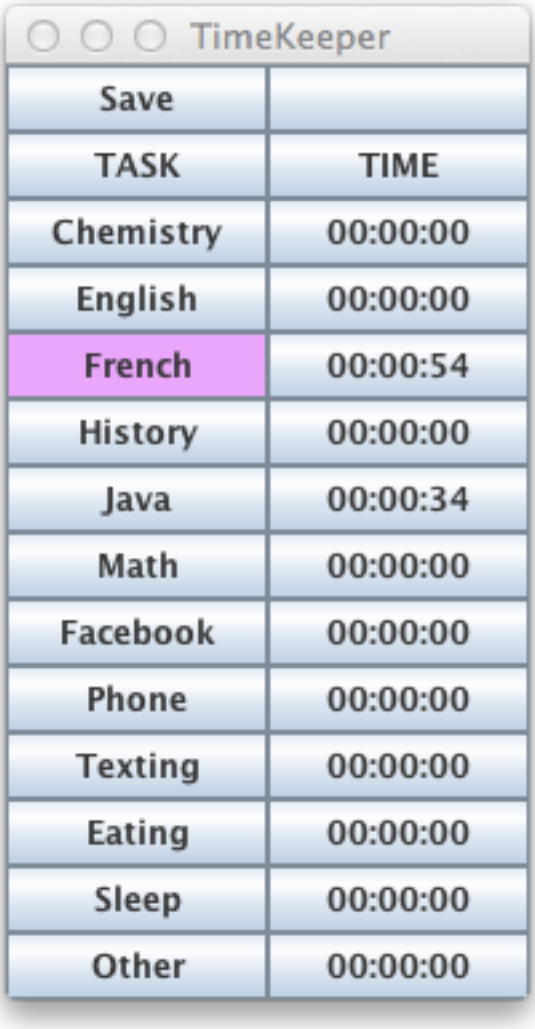
The second time I did this exercise, the time I wrote these notes, it went a lot faster (apart from the time spent writing), and I understood more, though still, to be honest, my understanding of how the program works is incomplete. Time permitting, I should read through the code more to better understand how the buttons are connected to the timers, the timers to the time displays, so on.

As is generally true of exercises, to get full benefit you must perform the exercise, not just read about it. Take time, as you first go through these steps, to try to understand why each step makes sense. Try to solve each challenge yourself rather than going immediately to the solution provided. Or at least think about how you would solve it.

Simply getting the right result by copying from the answer section will be no more educational than completing a math assignment by copying the answers from a solution guide. The solutions are for when you get stuck; they should be the last resort, not the first.

View what follows as a script, and make it your aim to get to the point where you can perform the play without reading from the script, and where you can do that not because you memorized the words of the script, but because you have the ability to envision what must be done to accomplish the task. To get maximum benefit, rehearse the script multiple times, until finally, as an accomplished actor, you can perform without it.

One more thing. Creating this device should be fun. Consider the process as a sequence of more or less challenging puzzles. Solving each one is rewarding, and at the end you will have a nice software device that can be a helpful tool for keeping track of time. Here's a screenshot of the finished application:



TimeKeeper

We want to create a tool that will keep track of time spent on various activities. To manage limited time better, a good way to start is by tracking how much is used in various ways, and this tool will make it easy to do that.

Imagine a GUI (Graphical User Interface) that consists of a list of tasks. Suppose you want to keep track of how much time you spend on homework for various subjects, how much you spend on Facebook, eating, and sleeping. Next to each task name, the application we'll call TimeKeeper will display the amount of time spent on that task. We'll need a timer to keep track of time. Let's control it as simply as possible: clicking on the name of the task will start the timer for that task, and clicking on the task name again will stop it, as will clicking on any other task name.

Let's make the active task red so it's easy to find. Let's display the time as hours:minutes:seconds -- the seconds will be good during development so we can quickly see whether the time is being updated.

How can we create such a program using Java, without a lot of Java expertise?

Challenge 1. What are the essential elements needed?

Solution:

Think about what this device we're constructing needs to do its job, the essentials without which it cannot work. To keep it simple, aim first for the minimal type of such a device, one that works for a single task, with a control button and an associated time display. This minimal device needs

- a timer to keep track of time,
- a control button to start and stop the timer, and
- a time display.

Challenge 2. How can we create a timer?

No idea how to write code for a timer in Java? That on the face of it poses a difficult challenge. But implementing this part of the Java program is not very hard if you know

how to approach it. Start by searching on the internet for an example of Java timer code.

Solution:

Google for "*java timer example gui start stop*". Start with *java* to narrow the scope to Java programming. We want an *example*, so it may help to request one. If you think about it, you can imagine someone might want a timer in a program without an associated GUI, for instance in a torpedo controller programmed to explode a certain time after launch. So specify *gui*. And since we want a start and stop button, we can improve chances of getting one by adding *start* and *stop*.

At the top of the results is **Java** - Updating a **GUI** made in Swing - Stack Overflow. Looking at it, we see right at the start:

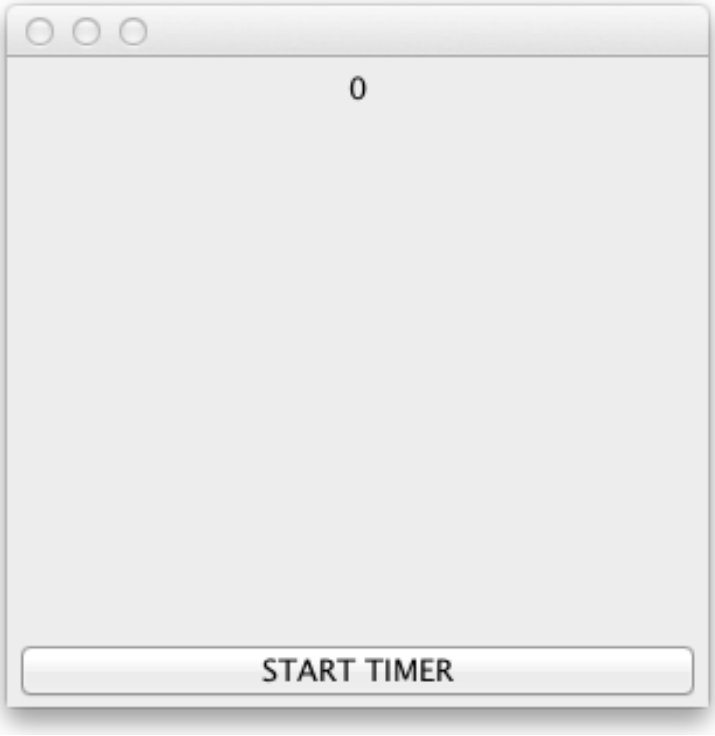
I am trying to create a simple GUI form which has only 2 elements - a simple label and a button.

This looks promising. Below we see someone's code that doesn't work, with a request for help. Just below that, some kind guru has responded with sample code for an UpdateWithTimer class. With luck this is what we need. If you're less lucky, you'll have to search around more, but generally, for any reasonably normal thing you'd like to do with Java, you can find sample code that will help.

1. Create a Java Project in Eclipse in which to develop this program, named TimeKeeperProject (use defaults).
2. Create a new Package, named TimeKeeperPackage.
3. Create a new Class, named TimeKeeper1.
4. Copy the sample code for UpdateWithTimer. Paste it into the TimeKeeper1 class, replacing the default class code there (leave the package statement). Change each occurrence of *UpdateWithTimer* to *TimeKeeper1*.

Suggestion: Repeat step 3 to create another class, TimeKeeper0, as a convenient way to keep the original code available.

Run TimeKeeper1 (if asked, choose "as a Java Application"):



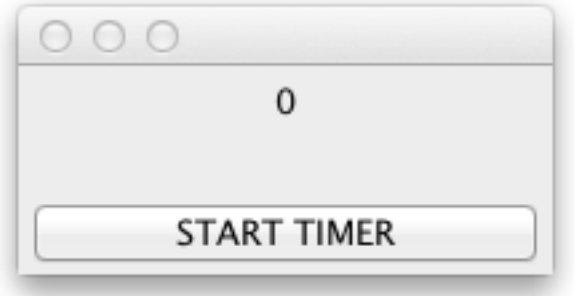
Click to start it, click to stop. Confirm it works.

Challenge 3. How does this timer program work?

Solution:

Look at the code. Try to understand how it works, what the different parts do. Main creates a new instance of our class, then runs createAndDisplayGUI, which creates a Panel, Label, and Button, and puts the Label and the Button on the Panel, and makes them visible. Along the way, it also adds the buttonAction listener, named buttonAction. And it creates a timer. And it sets the size to 300 x 300 pixels. Try changing 300, 300 to 200, 100 and run again to see how that works.

This looks closer to what we want:



Challenge 4. How can we arrange multiple buttons in a grid?

Solution:

The BorderLayout used here, which puts things at borders such as PAGE_END, presumably meaning at the bottom of the Panel, does not look promising for adding multiple buttons. What we want is a grid with 2 columns, one for buttons and one for times, something like this:

TASK	TIME
Chemistry	00:00:00
English	00:00:00
French	00:00:00
History	00:00:00
Java	00:02:42
Math	00:00:00
Facebook	00:00:00
Phone	00:00:00
Texting	00:00:00
Eating	00:00:00
Sleeping	00:00:00
Other	00:00:00

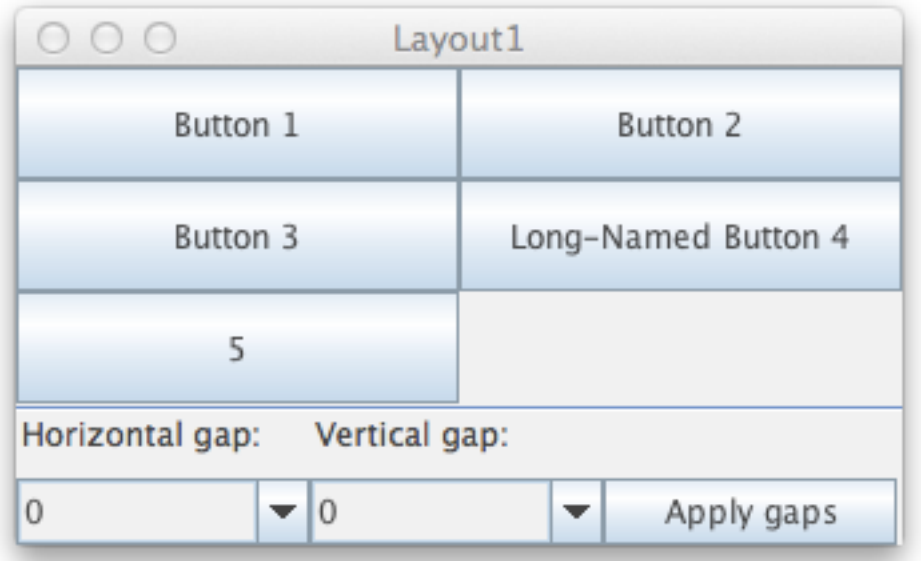
So look for a grid example. Search for *java grid example* or *java layout grid example.*

The first result is How to Use **GridLayout** (The **Java**™ Tutorials...

Looking there, we find the GridLayoutDemo, with a few buttons in a couple columns and some fancier stuff below. Let's take this GridLayoutDemo.java code and see if we can turn it into the sort of display we want, just 2 simple columns.

Create a new Class called Layout1

Copy the code (except the package line at the top) paste it in, replacing the default Layout1 definition (and leaving the package statement).
Change all occurrences of GridLayoutDemo to Layout1.
Run it to confirm it works.
You should see:



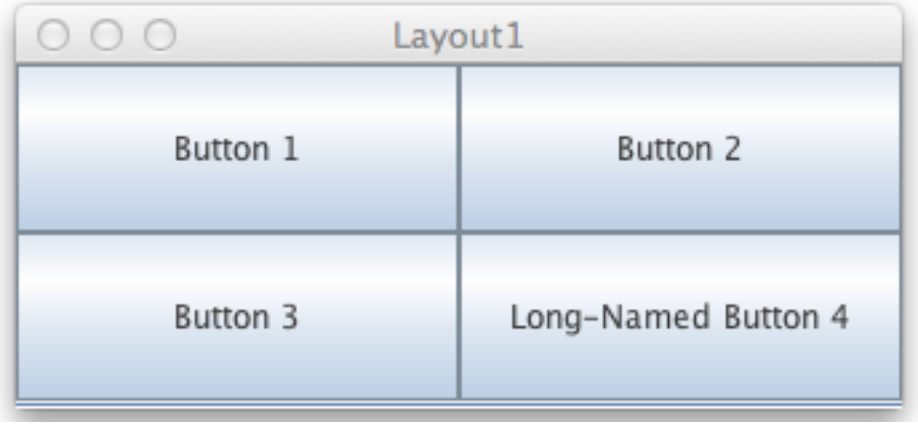
Again, you might want to make another class, Layout0, where you keep the original code readily at hand for reference (or you could call it GridLayoutDemo and not bother with changing the name, the only disadvantage being that the relation to Layout1 is less clear).

As before, we should examine the sample code to try to understand how it works. We see the same sort of invokeLater stuff in *main,* preceded by *setLookAndFeel* code that doesn't look very essential. *CreateAndShowGUI* looks a lot like *TimeKeeper1*'s *CreateAndDisplahyGUI,* except it calls *addComponentsToPane* to do some setup.

Let's simplify Layout1 to get rid of what's not needed for our purposes. If we could keep just the top four buttons, and turn the ones on the right into Labels, we'd have the sort of structure we want.

In removing what's not necessary, you might want to comment out the lines to be deleted and make sure the program still runs before actually deleting them, or if you break it, you might try relying on undoing with Ctrl-Z (or Cmd-Z on Mac) to get back to a working version.

So comment out the compsToExperiment.add statement for JButton 5, and all the controls.add statements. Try running the program. You should get:



This is closer to what we want. Since we don't care about them, let's try getting rid of the JComboBoxes, horGapComboBox and verGapComboBox, and the applyButton. Commenting out their declarations makes warning x's appear where they are used. Comment out these uses to fix that. When commenting out the first line of applyButton.addActionListener, comment out the whole block down to its closing "});" Since the initGaps method no longer does anything once its contents are commented out, comment out initGaps as well.

Run the program to confirm all is well, then to reduce clutter delete the lines commented out so far.

Change buttons 2 and 4 to labels by changing JButton to JLabel and their text to Label 1 and Label 2. Change "Button 3" to "Button 2".

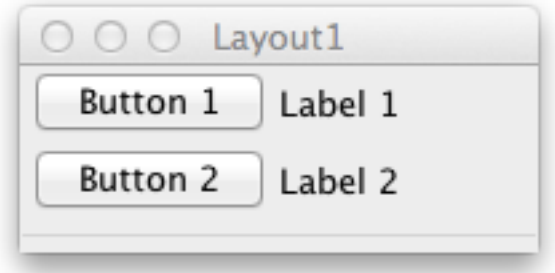
Run to confirm all is well:

To make clear it's a panel, change the name *compsToExperiment* to *myPanel*. Since we don't care about the metal look and feel, try removing the "Use an appropriate Look and Feel" and "Turn off... bold fonts" parts of main.

Run to confirm all is well:

Try getting rid of the setPreferredSize statement and since that's the only place buttonSize is used, delete that too.

Run to test:



JButton b is not used, so remove it. Everything we care about is in the myPanel Panel. The *controls* Panel has nothing in it -- it was used for the labels and buttons we threw away -- so get rid of it (delete all statements using it). Run to test.

Get rid of gapList and maxGap, which are no longer used, and the JSeparator, that we don't care about. Test.

Now we have a fairly simple example of how to create a grid of Buttons and Labels.

Challenge 5. How can we use this layout in our TimeKeeper program?

Solution:

Before we begin surgery, let's make a new Class, **TimeKeeper2**, and copy TimeKeeper1 into it, changing all the TimeTracker1's to TimeKeeper2's. This way if we mangle it too badly, we can always refer back to TimeKeeper1 to find a way to fix it (if necessary simply by recopying TimeKeeper1).

Let's start by copying the addComponentsToPane method from Layout1, putting it after the declarations at the start of the TimeKeeper2 class. This produces an error x, which when we hover over it shows the message "experimentalLayout cannot be resolved to a variable" -- which means there is no such variable declared. So we need to copy in the declaration for experimentalLayout. (Let's rename it myLayout and put it after the declarations already at the start of the TimeKeeper2 class.)

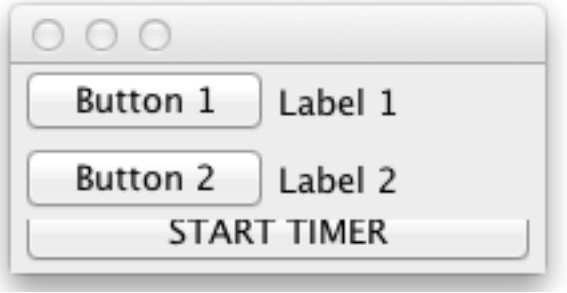
Then to make use of it, copy the frame.addComponentsToPane statement from Layout1. Before copying, note that in Layout1, frame.addComponents is followed by frame.pack(), whereas in TimeKeeper2 there is no corresponding pack() statement. To see what pack() does, search for *java pack*. The first result, How to Make Frames (Main Windows) tells us:

The `pack` method sizes the frame so that all its contents are at or above their preferred sizes.

Since this seems like it might be useful, copy the frame.pack() statement along with frame.addComponentsToPane to createAndDisplayGUI after the setDefaultCloseOperation statement. (While we're at it, let's change the name to addComponentsToPanel to better reflect the fact that this method adds to a Panel.)

This results in the error "frame cannot be resolved". Note that TimeKeeper2 has just setDefaultCloseOperation whereas Layout1 has frame.setDefaultCloseOperation, and just setVisible instead of frame.setVisible. So we can try deleting the frame. prefix before addComponents and before getContentPane.

Testing shows the program will run, producing a combination of our grid buttons and labels from Layout1 together with the startStopButton from TimeKeeper1:



Let's get rid of the startStopButton and give its role to Button 1. So basically we want to replace occurrences of startStopButton with the name of the Button 1 variable, but there is no such name, because this button is defined where it is used:

```
myPanel.add(new JButton("Button 1"));
```

We need to create a declaration separate from this use, so we can refer to the button elsewhere:

So up where we have declarations of startStopButton and changingLabel, let's add:

```
    private JButton button1;
```

And while we're at it, let's do likewise for button2, label1, and label2.

Having declared these items, we can now create them before adding them, replacing

```
myPanel.add(new JButton("Button 1"));
myPanel.add(new JLabel("Label 1"));
myPanel.add(new JButton("Button 2"));
myPanel.add(new JLabel("Label 2"));
```

with

```
button1 = new JButton("Math");
button2 = new JButton("Button 2");
label1 = new JLabel("Label 1");
label2 = new JLabel("Label 2");

myPanel.add(button1);
myPanel.add(label1);
myPanel.add(button2);
myPanel.add(label2);
```

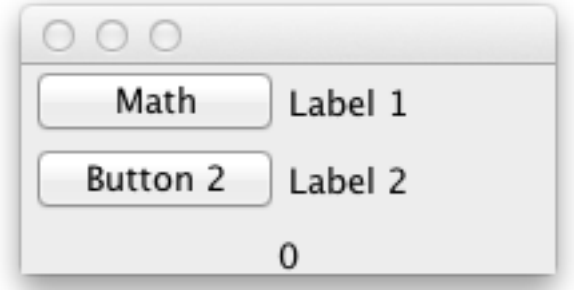
Now we can remove the declaration for startStopButton and replace startStopButton with button1. Since we already created button1, we can delete the statement that created startStopButton:

```
startStopButton = new JButton("START TIMER");
```

Also, since we already added button1 in addComponentsToPanel, we can delete

```
add(startStopButton, BorderLayout.PAGE_END);
```

Test:



Click on button1 (labeled Math). We see button1 is now controlling the timer, as we'd like, and the results are showing up in the changingLabel, which is now at the bottom of the display (because it was added to the contentPane, and the contentPane was added after our new buttons and labels were added by addComponentsToPanel).

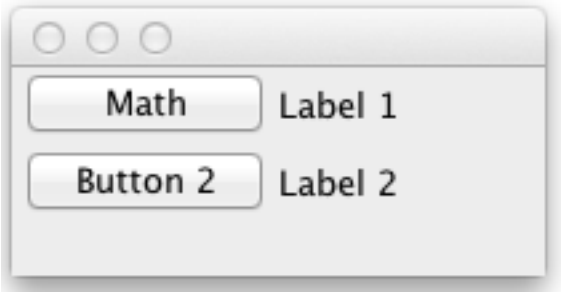
Comment out the statements where button1's text is set to STOP TIMER and START TIMER, because we don't really want to change the button text.

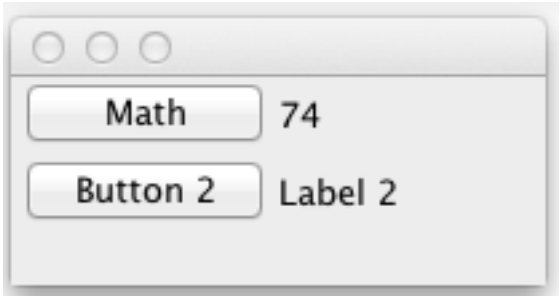
Next let's give the role of changingLabel to label1, much as we gave the role of startStopButton to button1.

Generally, where changingLabel appears, replace it with label1. But as before, delete the declaration for changingLabel, and delete the statement that creates changingLabel, since we already created label1. Moreover, since we already added label1, delete the statement that adds changingLabel:

```
contentPane.add(changingLabel);
```

Test:





Now clicking on button1 results in the timer result showing in label1.
We now have the basic functionality we want with the sort of layout we want.

Challenge 6. What can we learn from progress so far?

Solution:

We still have substantial work to do to produce the desired device, but the biggest obstacles have been overcome. We've implemented the fundamental functionality, a timer controlled by a button with results displayed next to it in a grid, and we've done it with little if any understanding of how to write the code for a timer or how to connect a timer to a controlling button or how to lay out objects on a grid.

Now is a good time to review what we can learn from this exercise:

- One can learn Java in the sense of acquiring the ability to produce useful programs without learning Java in the sense of learning lots of details about how Java works, the sort of detailed knowledge that would enable someone to sit down and write such a program from scratch. From a practical perspective, the former is more important than the latter. Of course the second sort of learning Java has value too, and it's a good idea, when working on a program like this, to examine the code and learn what you can about how it works. But the good news is you can accomplish a lot with Java without first learning a lot about it.
- To write a program without knowing how to do it from scratch, first consider what key functionalities are needed, then search for helpful examples. Simplify such examples as needed to get rid of confusing embellishments. Modify what remains to evolve it into what you want.
- Test very frequently, so if you break the program it's easy to identify what change broke it, and thus relatively easy to fix it.
- Start by creating a minimal, rudimentary version of your desired program. Then refine and extend it, bit by bit, feature by feature, changing it into what you want, but making sure at each step along the way that you still have a functioning program. If you spend lots of time without testing, making changes to a broken program that doesn't run, it can be far more difficult to figure out later how to repair it.

Examine the TimeKeeper2 program and understand as best you can how it works. Note, for instance, that the JPanel named contentPane is created and added, but nothing has been added to this panel; it is unused. So remove the statements involving it, and as always, test to make sure nothing's broken.

Challenge 7. How can we turn off resizability?

Solution:

Note that the Layout1 GUI cannot be resized, whereas the TimeKeeper1 GUI can be. Looking for the code that accounts for the difference, we see the following in Layout1:

```
public Layout1(String name) {
    super(name);
    setResizable(false);
}
```

This is a Constructor, a method with the same name as the class. With super it executes the default constructor for this class, which, since this class extends JFrame (that is, it is a JFrame class, with whatever additions we make), is the JFrame class constructor. Then it sets Resizable to false. In other words, it makes the GUI unresizable. To get the same behavior in TimeKeeper2, copy this constructor into TimeKeeper2 (after the GridLayout declaration is fine). Turn it into a TimeKeeper2 constructor by changing Layout1 to TimeKeeper2.

Now there's a problem: The constructor TimeKeeper2() is undefined... apparently because we have overridden the default no arguments constructor by specifying a constructor with a name. Fix it by adding the class name as the name argument in the constructor. (Or you could remove the name argument from our new constructor.)

Test to confirm the app is no longer resizable.

Challenge 8. How can we make the active task a distinctive color?

Solution:

Search for *java button background color example.*
Where the ActionListener used setText to change the button1 text, add statements to change the button1 background color:

```
button1.setBackground(Color.red);
```

To set the background color back to the default, specify the color as null:

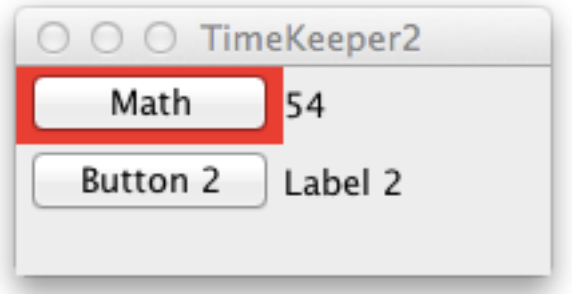
```
button1.setBackground(null);
```

Test. On a Mac, the test fails.
Search for *java button background color not working.*

java - How to Set the **Background Color** of a **JButton** on the Mac OS explains that to make this work on a Mac you need to add

button1.setOpaque(true);

Try that and test:



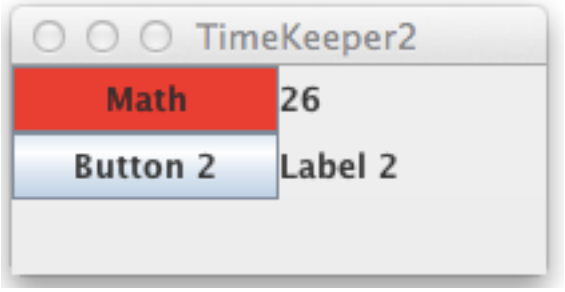
This looks a bit odd.

Further down on the same "How to Set..." page is the advice that you can instead add

```
try {
      UIManager.setLookAndFeel( UIManager.getCrossPlatformLookAndFeelClassName() );
} catch (Exception e) {
      e.printStackTrace();
}
```

before you add the GUI components. So put that at the top of addComponentsToPanel and remove the setOpaque statement.

Test.



If you like it better this way, keep it so; otherwise revert.

If you're interested in using a color other than the standard red, green, etc., search for *java color example.*

http://www.otherwise.com/Lessons/ColorsInJava.html explains that you can create your own color with

```
new Color(red, green, blue)
```

where red, green, and blue are integers from 0 to 255 (because one byte, 8 bits are used for each), giving a choice of $2^8 * 2^8 * 2^8 = 2^{24} = 2^{10} * 2^{10} * 2^4 \sim= 16{,}000{,}000$ colors.

Try

```
button1.setBackground(new Color(255, 200, 50));
```

Search for *color selector* to find a site like ColorPicker.com : Quick Online **Color Picker** Tool | HTML Color Codes that makes it easy to find the RGB (red-green-blue) values for whatever color you like; for instance, find and use a pink color you like, perhaps 250, 150, 250.

Challenge 9. How can we display hours and minutes?

Solution:

So far our timer is displaying seconds, because we created it with

```
timer = new Timer(1000, timerAction);
```

The 1000 tells the timer to perform the timerAction every 1000 milliseconds.

How can we display hours and minutes, instead of just seconds?

Search for *java timer hours minutes seconds hh:mm:ss.*

java - How to convert milliseconds to "**hh**:**mm**:**ss**" format? recommends:

```
String hms = String.format("%02d:%02d:%02d", TimeUnit.MILLISECONDS.toHours(millis),
```

```
    TimeUnit.MILLISECONDS.toMinutes(millis) -
              TimeUnit.HOURS.toMinutes(TimeUnit.MILLISECONDS.toHours(millis)),
    TimeUnit.MILLISECONDS.toSeconds(millis) -
              TimeUnit.MINUTES.toSeconds(TimeUnit.MILLISECONDS.toMinutes(millis)));
```

Paste this just before the label1 setText statement. Since instead of a millis variable representing milliseconds, we have a counter variable whose values are seconds, we can try changing this code to:

```
String hms = String.format("%02d:%02d:%02d", TimeUnit.SECONDS.toHours(counter),
TimeUnit.SECONDS.toMinutes(counter) -
       TimeUnit.HOURS.toMinutes(TimeUnit.SECONDS.toHours(counter)),
TimeUnit.SECONDS.toSeconds(counter) -
       TimeUnit.MINUTES.toSeconds(TimeUnit.SECONDS.toMinutes(counter)));
```

and since TimeUnit.SECONDS.toSeconds(counter) is apparently just converting seconds to seconds, we can replace it with simply *counter*:

```
String hms = String.format("%02d:%02d:%02d", TimeUnit.SECONDS.toHours(counter),
TimeUnit.SECONDS.toMinutes(counter) -
       TimeUnit.HOURS.toMinutes(TimeUnit.SECONDS.toHours(counter)),
counter - TimeUnit.MINUTES.toSeconds(TimeUnit.SECONDS.toMinutes(counter)));
```

Now change the setText statement to use "+ hms".

This results in the error "TimeUnit cannot be resolved."

Hovering over TimeUnit, we get suggested fixes, starting with Import TimeUnit. Select that option and not that an import TimeUnit statement is added along with other imports before our class definition. (TimeUnit is defined in a package; we need to import the definition so our class knows what TimeUnit is.)

Test.

Challenge 10. How can we add more buttons and labels?

Solution:

We need to add more tasks, hence more buttons and labels. We can imagine doing so by copying our code for button1 and label1, and adding a new ActionListener for each button, and perhaps a new Timer for each as well. But this would be cumbersome and difficult to maintain. When we're going to have a lot of buttons, labels, or whatevers, it makes sense to think of keeping them in arrays. Then whatever we need to do to each one of them, we may be able to do in one place, in a loop that walks through the array taking care of each one in turn. The program will end up being much shorter, thus easier to examine and understand, and if we need to make a change to an operation on the objects in an array, we can make the change just once in a single place instead of another time in another place for each additional object.

So let's try putting first the buttons, then the labels, in arrays.

Replace the button1 declaration with an Array declaration:

```
private static final int nTasks = 12;
private JButton[] buttons = new JButton[nTasks];
```

When we create (not just declare) the button array with `new JButton`, we need to say how many JButtons will be in the array (so the computer will know how much space to reserve). To make it easy to change this number (and make it easy to use the same number elsewhere, e.g. in our label definition and in loops walking through the arrays), we keep it in the variable *nTasks*. Since this number is a constant, that will stay the same throughout the program, we declare that with the word *final*.

Now that button1 is no longer defined, go through the program and fix each occurrence of it, replacing button1 with buttons[0].

Test.

Now do the same for labels, starting with

```
private JLabel[] labels = new JLabel[nTasks];
```

Test.

We're creating new buttons and labels as follows:

```
buttons[0] = new JButton("Math");
button2 = new JButton("Button 2");
labels[0] = new JLabel("Label 1");
label2 = new JLabel("Label 2");
```

We want to convert to using a loop. But to do that, we need to somehow supply the text values, strings like *Math* and *Label 1*. We need to be able to refer to the appropriate text when all we know about the button being created is its index in the array. We'd like to use that index to get the appropriate test string.

We can accomplish that by putting the text items we need for button creation in an array of Strings, so that "Math" will be the first item in that array, and so on. It will be convenient to supply the strings when we declare the array:

```
String[] names = new String[] { "Chemistry", "English", "French",
"History", "Java", "Math", "Facebook", "Phone", "Texting", "Eating",
"Sleep", "Other" };
```

```
for (int i = 0; i < nTasks; i++) {
      buttons[i] = new JButton(names[i]);
      labels[i] = new JLabel("00:00:00");
}
```

For the labels, we don't need to bother with an array, because we can set them all to the same string, 00:00:00.

Test.

When we try to run, we get a NullPointerException, meaning the program is being asked to reference something, to act on something, to which it has a pointer. The pointer is supposed to point to the thing by giving its address, but in this case the pointer contains no address; it is empty, or null. The exception error message shows a Stack Trace, or call stack, a sequence of things the program was trying to do, most recent at

the top. Looking at the lines containing TimeKeeper2 (the others refer to actions elsewhere, in library code that the program makes use of), we can see *run* called *createAndDisplay*, which called addComponentsToPanel. Clicking on the line number links in the stack trace, we see that the problem occurred in *myPanel.add(button2)* -- which in turn called some code in Container.java, but that's not our code so we don't need to worry about the details there. We're trying to add button2, but where is button2? We replaced the old

```
button2 = new JButton("Button 2");
```

statement that created button2 with a loop that creates button[1]. So button2 does not exist, yet we're asking myPanel to add it. The pointer that should tell the program where to find it is empty, so the program halts with a NullPointerException.

Let's fix this by replacing button2 and label2 with buttons[1] and labels[1]. Since we're done with button2 and label2, delete their declarations.

Next let's put the adding of buttons and labels into a loop. Instead of creating a new loop, we can add them in the loop we just made.
where myPanel is adding buttons and labels, do it in a for loop. Each time through the loop we want to take care of a button and label pair:

```
for (int i = 0; i < ntypes; i++) {
      buttons[i] = new JButton(names[i]);
      labels[i] = new JLabel("00:00:00");
      myPanel.add(buttons[i]);
      myPanel.add(labels[i]);
}
```

Test.

Notice only the first few labels and buttons show. The rest don't fit in the size we specified with

```
setSize(200, 100);
```

Commenting that out fixes the problem. (But only because we included pack() to set the size of our JFrame. Try commenting out pack() to see what happens.)

Challenge 11. How can we activate all the buttons?

Solution:

Now is a good time to examine the program to get more understanding of how it works, especially how the button gets activated.

In CreateAndDisplayGUI we find

```
buttons[0].addActionListener(buttonAction);
```

Here the first button is connected up to the ActionListener named buttonAction. A bit above, we see that buttonAction is an ActionListener -- something that listens for actions like button presses -- and when the button it's connected to is pressed, it does this:

```
private ActionListener buttonAction = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        if (!flag) {
            //buttons[0].setText("STOP TIMER");
             buttons[0].setBackground(new Color(250,150,250));
            timer.start();
            flag = true;
        }
        else if (flag) {
            //buttons[0].setText("START TIMER");
             buttons[0].setBackground(null);
            timer.stop();
            flag = false;
        }
    }
};
```

There's a flag -- something called a boolean that can have just two values, true or false. (The name relates to Boolean logic, a system based on true and false, invented by George Boole.) If the flag is not true, the backgroundColor is set and the timer is started and the flag is set true. If the flag is true, on the other hand, the color is turned back to the default and the timer is stopped and the flag is set true. Here we have some simple logic: the flag keeps track of whether the button has been pushed (an odd number of times). If the button is off (pressed an even number of times), when it's pressed set the pink color and start the timer; if the button is on, when it's pressed restore the default

color and stop the timer. To program, it is very helpful to clearly understand such a logical system and to comprehend how it's implemented in the code.

One other piece is worth a look. How does the time get displayed in the label? This is taken care of by another ActionListener, called timerAction. It makes sense if you think about it. This listener is waiting for timer events.

```
        timer = new Timer(1000, timerAction);
```

Here we see the timer constructed as concisely as possible, as is the way in good programming languages. The timer is based on two essential pieces of information, the interval (1000 milliseconds) after which the action should occur and the action, timerAction, that should be executed. Here is timerAction:

```
    private ActionListener timerAction = new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            counter++;
            String hms = String.format("%02d:%02d:%02d",
TimeUnit.SECONDS.toHours(counter),
                    TimeUnit.SECONDS.toMinutes(counter) -
TimeUnit.HOURS.toMinutes(TimeUnit.SECONDS.toHours(counter)),
                    counter -
TimeUnit.MINUTES.toSeconds(TimeUnit.SECONDS.toMinutes(counter)));
            labels[0].setText("" + hms);
        }
    };
```

Again, it's as simple as can be: Add one to the counter. Format the string to be displayed. Display it on the label.

How can we get the timer working with other buttons, not just button[0]? Or do we need a separate timer for each button? That would get a bit messy. Do we need an array of timers?

What if, to keep it simple, where we connect button0 to the buttonAction ActionListener (as shown just below) we also connect up the other buttons to the same ActionListener (with a loop)?

```
        buttons[0].addActionListener(buttonAction);
```

Here's the buttonAction ActionListener code:

```
private ActionListener buttonAction = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        if (!flag) {
            //buttons[0].setText("STOP TIMER");
         buttons[0].setBackground(new Color(250,150,250));
            timer.start();
            flag = true;
        }
        else if (flag) {
            //buttons[0].setText("START TIMER");
         buttons[0].setBackground(null);
            timer.stop();
            flag = false;
        }
    }
};
```

You might wonder if it's possible to add the same ActionListener to multiple buttons. An easy way to test is to add another addActionListener line, for buttons[1], and see if there's any problem. Try this and confirm it behaves as expected.

The trouble is buttonAction ActionListener sets the color for buttons[0], regardless of which button was pressed to activate it. If only there were some way for buttonAction to determine which button had activated it...

Challenge 12. How can we make each button set its own color?

Solution:

Note that actionPerformed takes an argument, some sort of ActionEvent, evidently the event of a button being pressed. Wouldn't it make sense for this ActionEvent to contain information about which button instigated it, and shouldn't the actionPerformed method be able to extract that information from its argument (why else would it have that argument?) and do something useful with it?

Let's search for *java actionlistener actionevent*.

The second result, **java** - How to add **action listener** that listens to multiple buttons, looks most promising. It contains the example:

```
public void actionPerformed(ActionEvent e) {
    if(e.getSource() == button1)
}
```

So buttonAction can determine what button called it and do the right thing.

There is still a question of whether we can get a single timer to do what is needed for the different task buttons, but let's set that aside for the moment while we focus on getting the colors working right, so each button sets its own color.
Let's start with a little experiment to see if buttonAction can identify which button was clicked. Add the following at the start of actionPerformed in buttonAction.

```
for (int nButtonPressed = 0; nButtonPressed < nTasks; nButtonPressed++) {
     if ((JButton) ae.getSource() == buttons[nButtonPressed]) {
            System.out.println("button " + nButtonPressed + " was pressed");
     }
}
```

In createAndDisplayGUI, replace button[0].addAction (and button[1].addAction if you have that] with a loop:

```
for (int i = 0; i < nTasks; i++) {
    buttons[i].addActionListener(buttonAction);
}
```

Test.

To get each button setting its own color, instead of a single flag to keep track of whether the button of interest has already been pressed, we need an array of flags, one to keep track of each button.

Replace

```
private boolean flag = false;
```

with

```
private boolean[] pressedFlags = new boolean[nTasks];
```

Look at the code executed when the button is pressed:

```
if (!flag) {
    //buttons[0].setText("STOP TIMER");
     buttons[0].setBackground(new Color(250,150,250));
    timer.start();
    flag = true;
}
else if (flag) {
    //buttons[0].setText("START TIMER");
     buttons[0].setBackground(null);
    timer.stop();
    flag = false;
}
```

Move it inside the loop that determines which button was pressed, just after the println statement. This way it will execute only for the button that was pressed. Change buttons[0] to buttons[nButtonPressed] to set color for the right button. Similarly, replace flag with pressedFlags[nButtonPressed].

Test.

Now each button sets its own color.

By the way, we're assuming all the flags start out false. To confirm that, seach for *java boolean default*. If in doubt, you can always initialize, e.g. by adding, in the loop where the buttons and labels are added,

```
pressedFlags[i] = false;
```

Challenge 13. How can a button reset color of the earlier pressed one?

Solution:

An imperfection you may have noticed in testing is that although pressing a button sets its own color as we want, it fails to set back the color of the previously pressed button. To fix that, we can keep track of the previously pressed button (if any) and reset its color to the default.

```
private int activeButton = -1;
```

An initial value of -1 will indicate that no button has yet been pressed.

At the end of the actions taken by actionPerformed (at the end of the *if* statement), set activeButton to the button that was pressed.

```
activeButton = nButtonPressed;
```

Now, once we identify the button that was pressed (just after the *if* condition), we can reset the color of the active button, if any (that is, if nButtonPress is not -1) with the statement. At the same time, we should update the pressedFlag for that formerly active button to indicate it is no longer pressed:

```
if (activeButton >= 0) {
      buttons[activeButton].setBackground(null);
      pressedFlags[activeButton] = false;
}
```

Put this just after we identify and print out the number of the button that was just pressed, so it will be called whenever a button is pressed.

Test.

Challenge 14. How can we make each button interact with the timer?

Solution:

We want each label to display the accumulated time for its button. For this, we need to keep track of these accumulated times, one for each button. We already have an integer called *counter* that keeps track of accumulated time regardless of which button was pressed. Let's replace it with an array of counters, one for each button.

Replace

```
private int counter = 0;
```

with

```
private int[] counters = new int[ntypes];
```

Wherever the code does something with counter, it will now have to do the same thing with the counter for the appropriate button, the active button.

So

```
counter++;
String hms = String.format("%02d:%02d:%02d", TimeUnit.SECONDS.toHours(counter),
    TimeUnit.SECONDS.toMinutes(counter) - TimeUnit.HOURS.toMinutes
              (TimeUnit.SECONDS.toHours(counter)),
    counter - TimeUnit.MINUTES.toSeconds(TimeUnit.SECONDS.toMinutes(counter)));
```

must become

```
counters[activeButton]++;
String hms = String.format("%02d:%02d:%02d",
       TimeUnit.SECONDS.toHours(counters[activeButton]),
       TimeUnit.SECONDS.toMinutes(counters[activeButton]) -
 TimeUnit.HOURS.toMinutes(TimeUnit.SECONDS.toHours(counters[activeButton])),
       counters[activeButton] -
             TimeUnit.MINUTES.toSeconds(TimeUnit.SECONDS.toMinutes
                                        (counters[activeButton])));
```

Test.

The buttons are all active, but the times are all showing up in label0. Why?

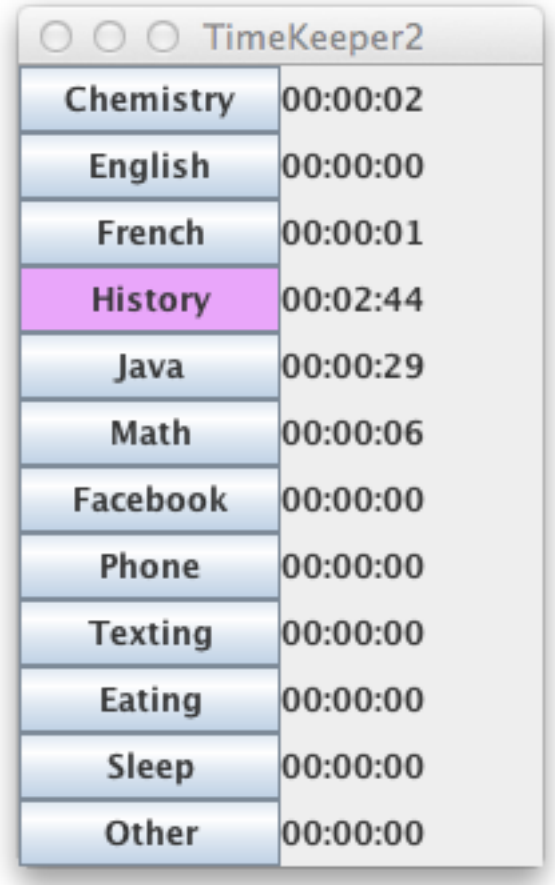
Because timerAction is setting it with

```
labels[0].setText("" + hms);
```

This needs to be changed to

```
labels[activeButton].setText("" + hms);
```

Now we have a program that does what we want.



Challenge 15. How can we spruce it up?

Solution:

There are a few ways we can make this look better. First, the labels are not displaying the time very nicely -- it's all the way over to the left. Let's try turning the labels into buttons.

```
private JButton[] labels = new JButton[nTasks];
```

Second, it would be nice to have headings at the top of the columns, "Task" and "Time".

To do this, add one to the size of each array:

```
private static final int nTasks = 13;
```

Look at the button creation code:

```
for (int i = 0; i < nTasks; i++) {
      buttons[i] = new JButton(names[i]);
      labels[i] = new JButton("00:00:00");
      myPanel.add(buttons[i]);
      myPanel.add(labels[i]);
      pressedFlags[i] = false; // not needed, but makes initial value clear
}
```

We now want this to operate starting with buttons[1] and labels[1] rather than buttons[0] and labels[0], so we need to change "int i = 0" to "int i = 1".

Similarly, in the code below for adding ActionListeners, we should change "int i = 0" to "int i = 1" so we don't add an ActionListener for the top button that says TASK.

```
for (int i = 0; i < nTasks; i++) {
    buttons[i].addActionListener(buttonAction);
}
```

And before creating those usual buttons and labels, we should take care of creating the headers, buttons[0] and labels[0]:

```
buttons[0] = new JButton("TASK");
labels[0] = new JButton("TIME");
myPanel.add(buttons[0]);
myPanel.add(labels[0]);
```

Test.

We get an ArrayIndexOutOfBounds Exception when creating new buttons, at the second line below:

```
for (int i = 1; i < nTasks; i++) {
     buttons[i] = new JButton(names[i]);
```

We changed nTasks to 13, but there are still only 12 names in the names array, so when i is 12, the program cannot find names[i]. Since the numbering starts at zero, it's looking for the 13th name in the array, which does not exist.

The fix is easy: just add another string at the start of the names array. It doesn't matter what if anything is in it, because this names[i] string is not used:

```
String[] names = new String[] { "", "Chemistry", "English", "French",
      "History", "Java", "Math", "Facebook", "Phone", "Texting",
      "Eating", "Sleep", "Other" };
```

Test again.

If you test well, you should find a problem.

If you click on a button, and then click on that button again, the color does not go away. To figure out what's going wrong, we can get help from the debugger.

Click on Run - Debug configurations... Click on *Stop in main* if it is not already selected. This will make the program stop and wait at the start of main. Click Apply, then Close.

Instead of clicking on the triangle in the green circle to run the program, click on the bug icon to debug it. If you see "This kind of launch is configured to open the Debug perspective...", select "Remember my decision" and click Yes.

Notice the highlighting indicating the program is waiting at the start of main. Scroll up to where buttonAction is defined, and for each line that calls setBackground, double click on the far left. Little blue dots will appear, like this:

```
        if ((JButton) ae.getSource() == buttons[nButtonPressed]) {
            System.out.println("button " + nButtonPressed + " was pressed");
             if (activeButton >= 0){
                buttons[activeButton].setBackground(null);
                pressedFlags[activeButton] = false;
            }
            // now take care of the button just pressed
            if (!pressedFlags[nButtonPressed]) {
                buttons[nButtonPressed].setBackground(new Color(250,150,250));
                timer.start();  // or  if (!timer.isRunning()) timer.start();
                //if (!timer.isRunning()) timer.start();
                pressedFlags[nButtonPressed] = true;
            }
            else if (pressedFlags[nButtonPressed]) {
                buttons[nButtonPressed].setBackground(null);
                timer.stop();
                pressedFlags[nButtonPressed] = false;
            }
            activeButton = nButtonPressed;
        }
    }
```

Each dot indicates a *breakpoint*, a stop sign telling the program to stop and wait when it reaches one.

Now click on the Resume icon, a green triangle to the right of a gold bar:

This tells the program to continue running from where it is currently stopped (at the start of main). The GUI appears; the program is waiting for input.

Click on the French button. Highlighting shows the program got to the second setBackground statement. It did not get to the first one, because at the outset activeButton is -1. The pressedFlag is not true yet for the button pressed, so

```
if (!pressedFlags[nButtonPressed]) {
```

is true, and the statements in the body of that *if* statement are executed.

Click on the Resume icon again (or use F8). Now the statement it was stopped at is executed (the button background color is changed), and again the program is waiting for input.

Click on the French button again. Now activeButton is 2 (the French button). Here's where we expect something to go wrong, so to step through (execute) the program one line at a time, click on the "Step Over" icon:

The next statement to be executed sets pressedFlags[activeButton] to false. Step again. And step once more. Now we're in the code that should be run if the button just clicked was not pressed earlier, because pressedFlags[nButtonPressed] is false. The active button, in this case, is the same as nButtonPressed, because we pressed this button twice in a row. Making the previously pressed button inactive is something we need to do only if that's a *different* button than the one we just pressed. The code just below that takes care of the button just pressed.

So to fix this, we can reset the pressedFlag for the active button only if that is not the same as the button we just now pressed. To accomplish that, stop the program by clicking on the red square, then change

```
        if (activeButton >= 0) {
to

        if (activeButton >= 0 && activeButton != nButtonPressed) {
```

Now run the debugger again, following the same procedure as before:

- Continue to move beyond the start of main.
- Click French.
- Continue.
- Continue.
- Click French.
- Step repeatedly. Observe how the program executes statement after statement, this time doing what we want.

Notice as you keep stepping that execution starts alternating between these two lines:

```
        if ((JButton) ae.getSource() == buttons[nButtonPressed]) {
            System.out.println("button " + nButtonPressed + " was pressed");
```

Why is this?

After we took care of the button just pressed, the program continues walking through the buttons array, checking each button to see if it's the one just pressed. Of course each time it is not, because we already reached the one just pressed and took care of it.

This wasted effort is harmless, but to avoid it we can add a break statement when we've finished what we want to accomplish in the loop. Add it at the bottom of the block that is executed when the button is found -- when ae.getSource equals buttons[nButtonPressed]:

```
                else if (pressedFlags[nButtonPressed]) {
                      buttons[nButtonPressed].setBackground(null);
                      timer.stop();
                      pressedFlags[nButtonPressed] = false;
                }
                activeButton = nButtonPressed;
                break;
        }
```

When done debugging, click the red square to stop the program.

Test again.

Now we have a nice looking, usable device for keeping track of time spent on several tasks.

Although TimeKeeper2 is usable, one can imagine ways of improving it. For instance, there might be some way for a user who is not a Java programmer to add or change tasks by means of a GUI. There might be a way to save results, perhaps on a daily basis, so a chart could be produced (perhaps with the help of a spreadsheet program) showing how time spent each day on various tasks changed over a week or month. It would be nice to be able to run the program without having to run Eclipse. Here, we'll take care of just the last item on this wish list.

Select File - Export - Java - Runnable JAR file. (JAR stands for Java ARchive.)
Under Launch Configuration, select TimeKeeper2.
For Export destination, click on Browse, then give a filename, TimeKeeper2 and choose the directory where you want the executable program to be placed. On a Mac, you might want to put it in the Applications directory. Then click on Finish.

Challenge 16. How can we save the data?

Solution:

This little gadget we've created is nice, but the initial customer has a couple requests. *Allow me*, she requests, *to time two tasks at once, for I'm a multitasker: I might be eating while studying math. And please, give me a way to save the data, so that I can collect data daily, store each day's data, and produce a chart showing the trend over the course of a week or month or year.*

Reasonable enough requests, except who will program it? As I found years ago when I tried to find someone to volunteer to do the programming for an on-line education system for teaching writing, there's a shortage of people with the needed skills. So sometimes the only way to get what you want is to construct it yourself.

Let's think about how we could save the data, first focusing on how we want this feature to work. There will have to be a way to control it, so we can add a button, clicking which, we might imagine, will bring up a small window that will allow entry of a location and filename and OK and Cancel buttons. We have some idea how to add a button and how to connect it to an ActionListener, but we'll need to learn how to bring up another window or frame or panel or whatever it is we need, how to take care of its controls, and how to do the actually saving. When the saving is done, we'll reset the times to zero.

We need to consider the format in which the data will be saved. We have an array of buttons with meaningful text and a corresponding array of times. It's useful to know that in such a situation, it works well to use a csv format. CSV stands for comma separated values, which is just what it sounds like. On the first line we can put the task strings, separated by commas, and on the next line we can put the corresponding times, separated by commas. Or we might put the task names just once, the first time we add anything to the file, and then at the end of each day add the times for that day. For added usefulness, lets precede each of these strings with the date, say in MM/DD/YY format (followed of course by a comma).

A file in this format could be read into a spreadsheet program such as Excel that can then display the requested chart.

Instead of forcing the user to request saving each morning, we should let the user set a time at which the saving and resetting will be done each day. The default could be midnight. For a start, we could make midnight the only option. Then all we would need is a button to turn saving on and off. And at the outset, to get this under way as simply

as possible, we don't even need that button. We'll just make saving and resetting at midnight a feature.

But when developing this feature, we don't want the critical event to happen just once a day at midnight. Testing that could take days and keep us up late at night. So during development, we want to be able to easily set when the saving will be done, or better, we arrange for it to happen frequently, say every 30 seconds.

By the way, when we save the times, it would be good to have some simpler format, such as just hours, a single number that we can be sure the spreadsheet program can handle.

How about this? A button, a timer, an ActionListener that saves and resets. Simpler yet, let's start with a button that saves and resets: a button and an ActionListener. We can see from our existing program how to create the button and ActionListener. The challenge then will be to get the ActionListener to save and reset.

So, implement the button code. Ideally, you will look at the source code developed so far, see how to declare and create and add another button. (Let's add it at the top, keep it separate from our existing button array, and give it a new ActionListener.)

I just realized, if we add just one button at the top, it will mess up our grid. We had better add two buttons. The second one can do nothing.

```
private JButton saveButton;
private JButton blankButton;
...

private ActionListener saveAction = new ActionListener() {
   public void actionPerformed(ActionEvent ae) {
      System.out.println("Hello from saveAction!");
   }
};
...

saveButton = new JButton("Save");
blankButton = new JButton("");
myPanel.add(saveButton);
myPanel.add(blankButton);
...

saveButton.addActionListener(saveAction);
```

Test.

Now we need to add some saving. Search for *java save string to file example*. The top result, How do I **save** a **String** to a text **file** using **Java**? - Stack Overflow, looks promising, but I didn't find anything really appealing there. The second result, How to **write** to **file** in **Java** – BufferedWriter - Mkyong.com, looks better. Copy the import statements to the import section at the top of our class file. Then copy the lines from try to the right brace at the end of the *catch* section. Paste them after the print statement in saveAction. Change the file path (after "new File") to "TimeKeeperData.txt".

Test. (Look for the file in your workspace.)

Notice the section that creates the file if it doesn't exist. We can set a flag here and then use it to add the first row, of task names, when the file is created. The seventh result of the search, Writing **string** to a **file** | DaniWeb, advises that the API (Application Programming Interface) for BufferedWriter says:

A newLine() method is provided, which uses the platform's own notion of line separator as defined by the system property line.separator. Not all platforms use the newline character ('\n') to terminate lines. Calling this method to terminate each output line is therefore preferred to writing a newline character directly.

So it looks best to use newLine() to end our lines.

```
private boolean newFileFlag = false;
...

if (!file.exists()) {
	file.createNewFile();
	newFileFlag = true;
}
...
		if (newFileFlag){
			newFileFlag = false;
			String taskNameString = "";
			for (int i = 1; i < nTasks; i++){
				taskNameString = taskNameString + names[i] + ",";
			}
			bw.write(taskNameString);
			bw.newLine();
		}
		//bw.write(content);
		bw.close();
```

Test. (Remove the file first.)

Now add the code to construct a string containing the times followed by commas, followed by a statement that writes it to the file, followed by writing a newline to the file.

We set the label text with:

```
labels[activeButton].setText("" + hms);
```

It's reasonable to suppose we might get the label (actually button since we turned labels into buttons) text with

```
labels[i].getText();
```

So:

```
// write the times to the file
String timesString = "";
for (int i = 1; i < nTasks; i++){
      timesString = timesString + labels[i].getText() + ",";
}
bw.write(timesString);
bw.newLine();
//bw.write(content);
```

Test. (Remove the file first.)

Notice if you click Save a second time, the contents are overwritten. We need to append rather than add at the start.

Search for *java append to file example.*

The first result, How to **append** content to **file** in **Java** - Mkyong.com, tells us that using the argument *true* when creating a FileWriter will result in appending.

```
FileWriter fileWritter = new FileWriter(file.getName(),true);
```

Try adding that:

```
FileWriter fw = new FileWriter(file.getAbsoluteFile(), true);
```

Test. (Remove the file first.)

Now let's save the results as hours, rather than as hh:mm:ss. For this we can get the hours and the minutes and seconds from the hh:mm:ss string.

Search for *java parsing strings.*

The first result, **Parsing Strings** with split, shows one way we can extract what we want from the button text.

```
String employee = "Smith,Katie,3014,,8.25,6.5,,,10.75,8.5";
String delims = "[,]";
String[] tokens = employee.split(delims);
```

So we can use

```
for (int i = 1; i < nTasks; i++){
      tokens = labels[i].getText().split(delims);
      hourString = tokens[0];
      minuteString = tokens[1];
      secondString = tokens[2];
```

Now we have strings for hours and minutes, but we need to convert them to numbers.

Search *java convert string to number.*

The first result, **How to convert string to int** in Java? - Stack Overflow, shows we can use Integer.parseInt(), as shown below:

```
  System.out.println("Hello from saveAction!");
try {

      File file = new File("TimeKeeperData.txt");

      // if file doesn't exist, then create it
      if (!file.exists()) {
            file.createNewFile();
            newFileFlag = true;
      }

      FileWriter fw = new FileWriter(file.getAbsoluteFile(), true);
      BufferedWriter bw = new BufferedWriter(fw);

      if (newFileFlag){
            newFileFlag = false;
            String taskNameString = "";
            for (int i = 1; i < nTasks; i++){
                  taskNameString = taskNameString + names[i] + ",";
```

```
                }
                bw.write(taskNameString);
                bw.newLine();
        }
        // write the times to the file
        String timesString = "";
        String hourString = "";
        String minuteString = "";
        String secondString = "";
        String delims = "[:]";
        String[] tokens;
        double hours;
        int nHours = 0;
        int nMinutes = 0;
        int nSeconds = 0;

        for (int i = 1; i < nTasks; i++){
                tokens = labels[i].getText().split(delims);
                hourString = tokens[0];
                minuteString = tokens[1];
                secondString = tokens[2];
                nHours = Integer.parseInt(hourString);
                nMinutes = Integer.parseInt(minuteString);
                nSeconds = Integer.parseInt(secondString);
                if (nSeconds >= 30){
                        nMinutes++;
                }
                hours = nHours + (double)nMinutes / 60.0;
                timesString = timesString + hours + ",";
        }
        System.out.println(timesString); // just for testing
        bw.write(timesString);
        bw.newLine();
        bw.close();
} catch (IOException e) {
        e.printStackTrace();
}
```

Test.

Note that if you get up to 30 seconds for a task, saving results in something like:

```
0.016666666666666666,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
```

We don't need so many decimal places in our saved hours. Let's round to 2 places, still more that needed but this much precision will enable us to test after waiting only 30 seconds.

Search for *java round example*.

The first result, How to **round** double / float value to 2 decimal points in **Java**, from our friend Yong Mook Kim, indicates we can use *Math.round*, like this:

```
hours = Math.round(hours * 100.0) / 100.0;
timesString = timesString + hours + ",";
```

Using 100.0 ensures the number will be interpreted as a double rather than as an integer.

Test.

Now we need to make the saving reset the times to zero, and make it occur automatically at midnight.

Resetting times to zero should be easy. Just walk through the array of counters and set each to zero.

```
                bw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            for (int i = 1; i < nTasks; i++){
                counters[i] = 0;
            }
        }
    };
```

Test.

One problem: The times are getting set to zero in the array, but are not redisplaying (except for the active task). We need to use setText to get the zero values displayed:

```
for (int i = 1; i < nTasks; i++){
      counters[i] = 0;
    labels[i].setText("00:00:00");
}
```

Test.

Challenge 17. How can we automatically save data at a given time?

Solution:

It would be good to have the program automatically save the data at the same time each day, say at midnight, without the user having to manually save it. (Apart from the nuisance of manually saving each day, and the risk of forgetting, it will be difficult to arrange to manually save it at the same time each day. Requiring the user to be awake to save data at midnight is not reasonable.)

Search for *java get time example.*

The first result, **Java** – How to **get** current date **time** – date() and calender(), again from Yong Mook Kim, looks promising. It offers the following sample code:

```
DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
Date date = new Date();
System.out.println(dateFormat.format(date));
```

We'll want to have the program check the date (or time) every so often, say every 50 seconds, and when the time in HH:MM form is 00:00, save the data. Checking at an interval less than a minute will guarantee we don't skip over 00:00.

So let's create another timer, like the existing one but specifying an interval of 50000 milliseconds:

```
private Timer savingTimer;
```

```
private ActionListener savingTimerAction = new ActionListener() {
       public void actionPerformed(ActionEvent ae) {
          DateFormat dateFormat = new SimpleDateFormat("HH:mm");
          Date date = new Date();
          System.out.println(dateFormat.format(date));

       }
   };



       savingTimer = new Timer(50000, savingTimerAction);
       savingTimer.start();
       setVisible(true);
```

To fix the error *DateFormat cannot be resolved to a type*, mouse over *DateFormat* and select the first option, Import. Likewise for *SimpleDateFormat*. For *Date*, choose the second option, Import 'Date' (java.util). SQL is a database query language; here we just want the utility definition.

For testing, set the new Timer value to 2000 instead of 50000.

Test.

Now all we need is to compare the dateFormat.format(date) String with "00:00". How do we compare strings in Java? We need to be careful here, because == may test to see whether 2 string objects are the same object, rather than whether they contain the same character sequence.

Search for *java string comparison example.*

The first result, How do I **compare strings** in **Java**? - Stack Overflow, explains how this works:

> == tests for reference equality.
>
> `.equals()` tests for value equality.

Consequently, if you actually want to test whether two strings have the same value you should use `.equals()` (except in a few situations where you can guarantee that two strings with the same value will be represented by the same object eg: String interning).

== is for testing whether two strings are the same *object*.

So we need to use .equals(). To reuse the code inside saveAction, we'll make it a separate method:

```
    private void saveAndReset(){
            try {

                    File file = new File("TimeKeeperData.txt");

                    // if file doesn't exist, then create it
                    if (!file.exists()) {
                           file.createNewFile();
                           newFileFlag = true;
                    }

                    FileWriter fw = new FileWriter(file.getAbsoluteFile(), true);
                    BufferedWriter bw = new BufferedWriter(fw);

                    if (newFileFlag){
                           newFileFlag = false;
                           String taskNameString = "";
                           for (int i = 1; i < nTasks; i++){
                                  taskNameString = taskNameString + names[i] + ",";
                           }
                           bw.write(taskNameString);
                           bw.newLine();
                    }
                    // write the times to the file
                           String timesString = "";
                           String hourString = "";
                           String minuteString = "";
                           String secondString = "";
                           String delims = "[:]";
                           String[] tokens;
                           double hours;
                           int nHours = 0;
                           int nMinutes = 0;
                           int nSeconds = 0;

                    for (int i = 1; i < nTasks; i++){
                           tokens = labels[i].getText().split(delims);
                           hourString = tokens[0];
                           minuteString = tokens[1];
                           secondString = tokens[2];
                           nHours = Integer.parseInt(hourString);
```

```
                        nMinutes = Integer.parseInt(minuteString);
                        nSeconds = Integer.parseInt(secondString);
                        if (nSeconds >= 30){
                              nMinutes++;
                        }
                        hours = nHours + (double)nMinutes / 60;
                        hours = Math.round(hours * 100.0) / 100.0;
                        timesString = timesString + hours + ",";
                  }
            DateFormat dateFormat = new SimpleDateFormat("HH:mm");
            Date date = new Date();
            System.out.println("saveAndReset time is " + dateFormat.format(date));
                  bw.write(timesString);
                  bw.newLine();
                  bw.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            for (int i = 1; i < nTasks; i++){
                  counters[i] = 0;
           labels[i].setText("00:00:00");
            }
}

private ActionListener savingTimerAction = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
          //DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
          DateFormat dateFormat = new SimpleDateFormat("HH:mm");
          Date date = new Date();
          //System.out.println("savingTimerAction time is: " + dateFormat.format(date));
          if (dateFormat.format(date).equals("00:00")){
                saveAndReset();
          }
    }
};

private ActionListener saveAction = new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
          System.out.println("saveAction: calling saveAndReset due to button
                                          press");
          saveAndReset();
    }
};
```

Test.

Challenge 18. How can we put the date at the start of each saved row?

Solution:

Let's put the date at the start of each output row in the form mm/dd/yy. (We should also put "Date" at the start of the first row with the task names, so it will have the same number of items as later rows.)

We can make a method that will return the date in this format, using the date mechanism we already employed in savingTimerAction:

```
private String getDateMMDDYY(){
  DateFormat dateFormat = new SimpleDateFormat("MM/dd/yy");
  Date date = new Date();
  String mmddyyDate = dateFormat.format(date);
  System.out.println("savingTimerAction time is " + mmddyyDate);
  return mmddyyDate;
}
```

We had an example of a format using yyyy -- let's see if yy works.

In saveAndReset, just before writing out the timesString, we'll write out the date plus a comma:

```
bw.write(getDateMMDDYY() + ",");
bw.write(timesString);
```

Don't forget to put "Date," at the start of the taskNameString. Replace

```
String taskNameString = "";
```

with

```
String taskNameString = "Date,";
```

Test.