Homework Four

For this assignment we will develop (only) one program and in the process experiment with new and important features of the Java programming language: graphical user interfaces and events handling. The course website presents a functional model for this assignment; when you first start the program you are immediately greeted with a question made out of two random integers in [-50, 50]:

Welcome. What is -11 + 13?
Please enter your answer here: 24
Press Proceed

This is a GUI, a Graphical User Interface. A web browser’s GUI is the most conspicuous part of the browser and largely responsible for the (initial, continued) popularity of the WWW. In the program’s GUI we see, next to the question, a prompt (“Please enter your answer here:”) and a text field, along with a submit button (labeled “Proceed”). The user enters the answer and pushes the button for the feedback.

If the answer is wrong (as in the example above) the user has two more chances to get it right.

But if we made two more mistakes we’d lose the point. We make one more mistake, to face a loss.

If we avoid making three mistakes in a row (as in the snapshot above, on the right) we receive one point, the score is updated and a new question is shown. Below our second question was -17 + 20 we answer it well, go 4-0 overall and then we get 42 + 34, which we mistake three times in a row, last attempting an 86 as the right answer. We lose a point and it’s 4-1. New question shown is 28 + 39.
So that’s how the program needs to behave.

1. What is a JFrame?

When we discussed inheritance we presented a number of examples: Sophomore extends Freshman, Unicorn extends Horse, later in the help session we proposed that Circle extends Point (adding a radius). In all these examples these four phenomena are happening:

- we’re using the class extension mechanism
- objects of the extended class exhibit polymorphism
- objects of the extended class inherit the features present in instances of the superclass
- dynamic method lookup may occur upon invoking overridden methods of the superclass

Let’s discuss JFrame by association with any of the examples mentioned earlier. JFrame is the Horse that we want to extend to create a Unicorn (our application). Whoever wants to build a Unicorn better read up on Horses in advance.

So we can read about JFrame here. There is also a tutorial by Marty Hall (see picture below) and the official Java tutorial plus numerous other resources including the book in Books24x7 I recommended.

http://java.sun.com/javase/6/docs/api/javax/swing/JFrame.html
http://www.apl.jhu.edu/~hall/java/Swing-Tutorial/
http://java.sun.com/docs/books/tutorial/uiswing/components/frame.html
The first of these three links is the official documentation and it immediately sends us to the official Java tutorial (third link). Here’s the basic code we are being explained:

```java
import javax.swing.*;
import java.awt.*;

class One {
    public static void main(String[] args) {
        // 1. Create the frame.
        JFrame frame = new JFrame("FrameDemo");

        // 2. Optional: What happens when the frame closes?
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // 3. Create components and put them in the frame.
        // ...create emptyLabel...
        JLabel emptyLabel = new JLabel("A goal without a plan is just a wish.");
        frame.getContentPane().add(emptyLabel, BorderLayout.CENTER);

        // 4. Size the frame.
        frame.pack();

        // 5. Show it.
        frame.setVisible(true);
    }
}
```

There are many confusing methods, one use of JLabel, also some of the methods appear to produce objects that are subsequently used (getContentPane() locates one such object, that is then told to add the new label). We need some patience if we are to attempt to understand even the simple code above.

A JFrame can be constructed in any of 4 ways: see the constructor summary.

A newly created JFrame needs to be told:

- its default closing behavior (or mode of operation⁴)
- what components it has (to add and maintain) inside, including the desired layout
- its size,
- and whether it’s visible or not.

If you compile and run the program above we get:

![Image of command prompt showing Java output]

The pack method finds the smallest size that fits the bill, but there is also a setSize method inherited⁵ from the Window class that would allow you to give the JFrame the size you want.

---


⁵ [http://java.sun.com/javase/6/docs/api/java/awt/Window.html#setSize(int, int)](http://java.sun.com/javase/6/docs/api/java/awt/Window.html#setSize(int, int))
Finally, there is a way to bypass the layout manager altogether by indicating precisely where the components need to be placed in the JFrame and what size (dimension) they would have. This may have the disadvantage that depending on the system and the specific run-time circumstances the dimensions might not be entirely adequate. We’ll try to be careful with that.

### 2. What is a JLabel?

Just as we did with the JFrame we need to take a look at the official documentation for JLabel:

![Java Swing JLabel Class](http://java.sun.com/javase/6/docs/api/javax/swing/JLabel.html)

We see that a JLabel is a JComponent, which is a Container, which is a Component, which is an Object. Notice that the classes on the inheritance hierarchy are not all in the same package. Likewise, a JFrame is a(n awt) Frame, which is a Window, which is a Container, which is a Component, which is an Object.

Here’s the basic usage we want to propose for a JLabel:

```java
JFrame f = new JFrame(); // create a JFrame

Container c = f.getContentPane(); // find a place to hang paintings
    c.setLayout(null); // indicate that we’ll take care of the layout

JLabel output = new JLabel(); // the name of the label is “output”
    output.setBounds(50, 70, 220, 20); // how big, where the label is
    output.setText("0 clicks so far. Push the button."); // text on it
    c.add(output); // note how the label is added to the content pane
```

---


7. So both JFrames and JLabels are Components! This doesn’t really mean anything here but sounds interesting.

8. This is common and done once for all the buttons, labels, text fields and such to be placed in the JFrame.

9. This indicates we don’t want any automatic layout manager.
3. What is a JButton, JTextField, etc.?

Obviously both support the method setBounds because it's defined in their common superclass.

So let's get going putting together what will be a basic sketch for our program's GUI:

```java
import javax.swing.

public class One extends JFrame {
    public static void main(String[] args) {
        One a = new One();
    }
}
```

So One is the Unicorn to the JFrame class. What if we want to place a title in it? Is there any constructor chaining going on, and if so, how is it working? What happens if we compile and run this program?

---

12 [http://java.sun.com/javase/6/docs/api/java/awt/Component.html#setBounds(int, int, int, int)](http://java.sun.com/javase/6/docs/api/java/awt/Component.html#setBounds(int, int, int, int))
13 Java.awt.Component – which means even JFrame has it
This is what happens if you compile and run this program:

Of course there is constructor chaining. If I want to add the title I need to get involved in it and ask for the right version of the superclass’ constructor to be invoked in the chain of constructor invocations:

Besides that I have also removed the references to “this.” (they’re not needed but I recommend them, since they make things appear a bit more clearly) and one of them is commented out between /* ... and */ so we don’t overlook this point.

If you compile and run this you get (initial frame on the left, extended on the right below):

Once this relatively clear we can start adding the GUI components. We need:

- a textfield
- a button
- a few labels
Here's what we need to do to add the first JLabel:

```java
import javax.swing.*;
import java.awt.*;

class One extends JFrame {
    public static void main(String[] args) {
        One a = new One("This is my title");
    }

    One(String title) {
        super(title);
        this.createGUI(); // see definition below
        setSize(400, 400); // width, height
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        /*this.*/setVisible(true);
    }

    void createGUI() { // this places the components
        Container c = this.getContentPane();
        c.setLayout(null);

        JLabel question, feedback;
        JButton submit;
        JTextField answer;

        question = new JLabel();
        question.setBounds(20, 40, 180, 20);
        question.setText("What is 3 + 5? Answer here: ");
        c.add(question);
    }
}
```

So we need to get a hold of the content pane (which is a Container, hence the import) and after setting the bounds and the text we add the label. You won’t get the numbers right from the first time, so keep adjusting. You can also try to learn working with a layout manager but that increases the complexity of the program and doesn’t make it look much much better so I recommend we stay the course that we have charted thus far. Below see how we add the textfield and the submit button to the program’s GUI:

```java
    answer = new JTextField();
    answer.setBounds(20+180, 40, 40, 20);
    c.add(answer);

    submit = new JButton();
    submit.setBounds(20+180+40+20, 40, 105, 20);
    submit.setText("Proceed");
    c.add(submit);
```
We need to tackle two things:

- how do we generate (new) random numbers for our questions, and
- how do we get the user input and grade it?

Let’s think about what would make the state\(^4\) of our program:

- \(n_1, n_2\)
- userScore, computerScore
- currentScoreOfMistakes

I have chosen a long name for the last variable because it also communicates its meaning.

We write a method called initializeState then:

```java
import javax.swing.*;
import java.awt.*;

class One extends JFrame {
    public static void main(String[] args) {
        One a = new One("This is my title");
    }

    int n1, n2, userScore, computerScore, currentScoreOfMistakes;

    void initializeState() {
        this.n1 = (int) (Math.random() * 100 - 50);
        this.n2 = (int) (Math.random() * 100 - 50);
        this.userScore = 0;
        this.computerScore = 0;
        currentScoreOfMistakes = 0;
    }

    One(String title) {
        super(title);
        this.initializeState();
        this.createGUI();
        setSize(450, 200); // width, height
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // this.setVisible(true);
    }

    void createGUI() { // this places the components
        Container c = this.getContentPane();
        c.setLayout(null);
    }
}
```

We’d have to use this state in the questions we ask (report state).

```java
question.setText("what is " + this.n1 + " + " + this.n2 + " ? Answer here: ");
```

\(^4\) Recall that state is something that changes that we need to remember; input is something we can’t remember because we can’t even predict it, as it comes from the user; output is what we want to communicate (perhaps report the state among other things) and constants are things that never change.
Now we need to get input from the user.

4. What is an ActionListener?

Rest assured that if you press the button in your GUI the system you’re running this one notices. Whether it passes the information along to your program is another issue. Just like a referee is a very specialized entity that calls a game of basketball you could also put together an entity that could take some action when a certain circumstance warrants it.

An ActionListener is such an entity. ActionListener\(^\text{15}\) is an interface, so any class implementing it \((\text{extends is just for classes})\) would qualify as an ActionListener.

You read and find out there’s only one method we need to implement (above, right).

So let’s create such an object by defining a class satisfying these requirements:

If you compile and run this you will notice nothing happens when you press the button.

\(^{15}\) [http://java.sun.com/javase/6/docs/api/java/awt/event/ActionListener.html](http://java.sun.com/javase/6/docs/api/java/awt/event/ActionListener.html)
We have compared an action listener to a referee. The referee needs to be an instance of some class\(^{16}\), wear the right uniform\(^{17}\) and provide the knowledge that it implies. We have that. But if the referee is at home, asleep on the couch, in front of the TV, (s)he can’t call anything in the game. So we have to assign him or her to the game. Objects of type JButton need to have the ability to assign an action listener. Do they? The answer is: yes. JButton\(^{18}\) is an extension of AbstractButton and inherits that method from it.

```
Methods inherited from class javax.swing.AbstractButton
actionPropertyChange, addActionListener, addChangeListener,
configurePropertiesFromAction, createActionListeners,
doClick, doClick, fireActionPerformed, fireChange
getChangeListener, getDisabledIcon, getEnabledIcon,
gestHorizontalTextPosition, getIcon, getIconText,
gestHorizTextSize, getMultiClickThreshold, getPressedIcon, getRollover
getVerticalTextPosition, getVerticalTextSize, is
isRolloverEnabled, isPressed, paintBorder, remove
getBorderPainted, setComponentAreaFilled, setEnabled,
setHideActionText, setHorizontalAlignment, setHoriz
setMnemonic, setMnemonicText, setModel,
setMultiClickThreshold,
setRolloverSelectedIcon, setSelected, setSelected;
```

```
Methods inherited from class javax.swing.JComponent
addAncestorListener, addNotify, addPropertyChangeListener,
firePropertyChange, firePropertyChange, fireProperty
getPropertyChange, getAlignmentY, getAncestorListeners,
```

Here’s the description for that method:

```
addActionListener

public void addActionListener(ActionListener l)

Adds an ActionListener to the button.

Parameters:

  1 - the ActionListener to be added
```

And here’s how we use it:

```
submit = new JButton();
submit.setBounds(20+180+40+20, 40, 105, 20);
submit.setText("Proceed");
c.add(submit);
```

Now if you compile and run it every time you press the button the MS-DOS window will print: Ouch!

\(^{16}\) We called that Broker here because it will intermediate between the button and the rest of the program.

\(^{17}\) The interface is the uniform, which makes the promise that all included methods will be implemented. Likewise a uniform in real life is an advertisement for a very specific functionality (traffic agent, mailman, soldier etc.)

\(^{18}\) http://java.sun.com/javase/6/docs/api/javax/swing/JButton.html
Our action listener listens and takes action, just not the type of action we had envisioned. It needs to grab the input from the text field and compare it with the question, give feedback, update the score and ask a new question. How can it grab the input from the text field?

Getting input from the textfield is easy, as long as we have a reference to it. Maybe we need to pass that to the Broker constructor? Why not – whatever it takes ... but is this the only thing it needs? It also needs access to the two numbers n1, n2 and instead of printing in the MS-DOS window it better stick its output into the JFrame itself, as we originally discussed. So it looks like it needs a lot of things from the JFrame itself. Can we pass the entire JFrame to it? We sure can:

```java
submit = new JButton();
submit.setBounds(20+180+40+20, 40, 105, 20);
submit.setText("Proceed");
submit.addActionListener(new Broker(this));
c.add(submit);
```

This means that the definition of the Broker constructor needs to change accordingly:

```java
class Broker implements ActionListener {
One frame;
Broker(One frame) {
this.frame = frame;
}
public void actionPerformed(ActionEvent e) {
    System.out.println("Ouch!");
}
}
```

Thus actionPerformed will be able to access everything it needs in the original JFrame through its instance member frame. Except this would be a good time to realize that there are no instance variables in One whatsoever. All the GUI elements are defined as local variables in the instance method that creates the user interface and are thus lost, gone, vanished after the method ends. The objects remain attached to the content pane, in the GUI, but the variables are lost and the objects are anonymous and thus inaccessible from anywhere. So we change:

```java
void createGUI() { // this
Container c = this.getContentPane();
JLabel question, feedback;
JButton submit;
JTextField answer;
question = new JLabel();
}
```
As can be seen in the picture above the four variables now belong to the (JFrame, One) instance and thus survive the successive invocation of the various methods involved in the program – basically they exist and can be accessed for as long as the JFrame exists. So actionPerformed becomes:

```java
public void actionPerformed(ActionEvent e) {
    String message;
    if (frame.n1 + frame.n2 == Integer.parseInt(frame.answer.getText())) {
        message = "Good job."
        frame.userScore += 1;
    } else {
        frame.computerScore += 1;
        message = "No, that was not it."
    }
    message += " Score now: " + frame.userScore + " - " + frame.computerScore;
    System.out.println(message);
    frame.n1 = (int) (Math.random() * 100 - 50);
    frame.n2 = (int) (Math.random() * 100 - 50);
    frame.question.setText("What is " + frame.n1 + " + " + frame.n2 + " ? Answer here: ");
    frame.answer.setText(""");
}
```

With this the program is basically finished. The code is shown on the next page (magnify if it’s too small).

Here’s the program in action:

![Program in action](image)

To prevent the program from crashing when no integer is passed from the frame’s textfield one can try and catch the exception thrown. This is not needed here but should you be interested this is how it can be done:

```java
public void actionPerformed(ActionEvent e) {
    String message;
    try {
        if (frame.n1 + frame.n2 == Integer.parseInt(frame.answer.getText())) {
            message = "Good job."
            frame.userScore += 1;
        } else {
            frame.computerScore += 1;
            message = "No, that was not it."
        }
        message += " Score now: " + frame.userScore + " - " + frame.computerScore;
        System.out.println(message);
        frame.n1 = (int) (Math.random() * 100 - 50);
        frame.n2 = (int) (Math.random() * 100 - 50);
        frame.question.setText("What is " + frame.n1 + " + " + frame.n2 + " ? Answer here: ");
        frame.answer.setText(""");
    } catch (Exception exception) {
        System.out.println("Please enter a valid number in the text field!");
    }
}
```

There’s very little left to finish this program now. Please do it. The code is on next page.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class Broker implements ActionListener {
    One frame;
    Broker(One frame) {
        this.frame = frame;
    }
    public void actionPerformed(ActionEvent e) {
        String message;
        try {
            if (frame.n1 + frame.n2 == Integer.parseInt(frame.answer.getText())) {
                message = "Good job."
                frame.userScore += 1;
            } else {
                frame.computerScore += 1;
                message = "No, that was not it."
            }
        } catch (Exception exception) {
            System.out.println("Please enter a valid number in the text field!");
        }
        message += " Score now: " + frame.userScore + " - " + frame.computerScore;
        System.out.println(message);
        frame.n1 = (int) (Math.random() * 100 - 50);
        frame.n2 = (int) (Math.random() * 100 - 50);
        frame.question.setText("What is " + frame.n1 + " + " + frame.n2 + " ? Answer here: ");
        frame.answer.setText("");
    }
}

class One extends JFrame {
    public static void main(String[] args) {
        One a = new One("This is my title");
    }
    int n1, n2, userScore, computerScore, currentStreakOfMistakes;
    void initializeState() {
        this.n1 = (int) (Math.random() * 100 - 50);
        this.n2 = (int) (Math.random() * 100 - 50);
        this.userScore = 0;
        this.computerScore = 0;
        this.currentStreakOfMistakes = 0;
    }
    One(String title) {
        super(title);
        this.initializeState();
        this.createGUI();
        setSize(450, 200); // width, height
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        /*this.*/setVisible(true);
    }
    JLabel question, feedback;
    JButton submit;
    JTextField answer;
    void createGUI() { // this places the components
        Container c = this.getContentPane();
        c.setLayout(null);
        question = new JLabel();
        question.setBounds(20, 40, 190, 20);
        question.setText("What is " + this.n1 + " + " + this.n2 + " ? Answer here: ");
        c.add(question);
        answer = new JTextField();
        answer.setBounds(20+190, 40, 40, 20);
        c.add(answer);
        submit = new JButton();
        submit.setBounds(20+180+40+20, 40, 105, 20);
        submit.setText("Proceed");
        submit.addActionListener(new Broker(this));
        c.add(submit);
    }
}
Note that this program still uses the following template:

```
start the program

retrieve state

is state empty?

no
read user input
update state(input)
store state
report state

get ready for more input

yes
initialize state
```

This, of course, is the behavioral pattern of the event-handling method actionPerformed.

Other (generally) useful notes are located here\(^{19}\) or here\(^{20}\) or even here\(^{21}\). And don’t forget your text.