UI Tutorial 2
Swing

Tutorial Outline

The purpose of this tutorial is to give you an introduction to UI programming in Swing. We will follow the example from last week's React tutorial closely.

- This is only an introduction to Swing components.
- Lots of good examples are available from Oracle on both Swing and JavaFX.
- We will implement a simple UI for our existing FaceSpace REST API.

What We Will Cover

- Swing basics
- Using out-of-the-box and custom Swing components
- **Synchronous** communication between UI and back-end
- Design and implementation of a new Swing component

Note:

This example is meant to cover basic UI components, organization, data flow, and actions – not aesthetics.

Swing and JavaFX Resources

Swing was once a relatively popular Java framework for implementing desktop applications.

JavaFX is an enhanced UI framework with more modern network and multimedia features.

Swing and JavaFX are both implemented as a collection of extensible classes that are available by default in the JDK.

Even for Java programmers, UI programming is a different beast. Good UI coding requires care – no matter the framework.

Swing and JavaFX Resources

Swing examples:
https://docs.oracle.com/javase/tutorial/uiswing/index.html

JavaFX samples:
http://www.oracle.com/technetwork/java/javase/overview/javafx-samples-2158687.html

Our Plan

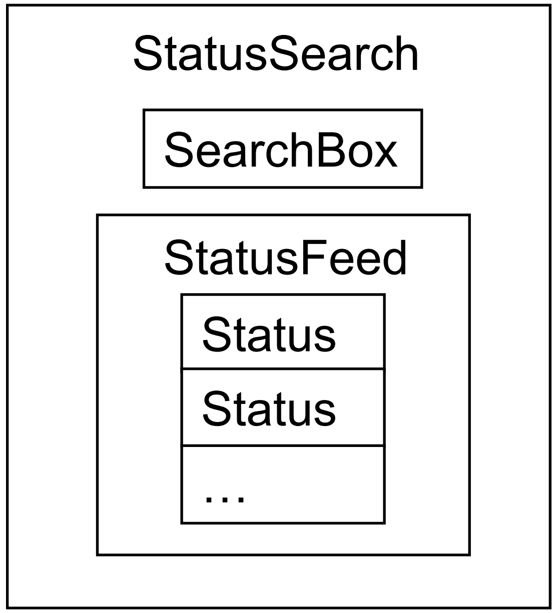
For this tutorial, we are going to try to apply the lessons we learned in last week’s tutorial to developing a similar UI for FaceSpace in Java.

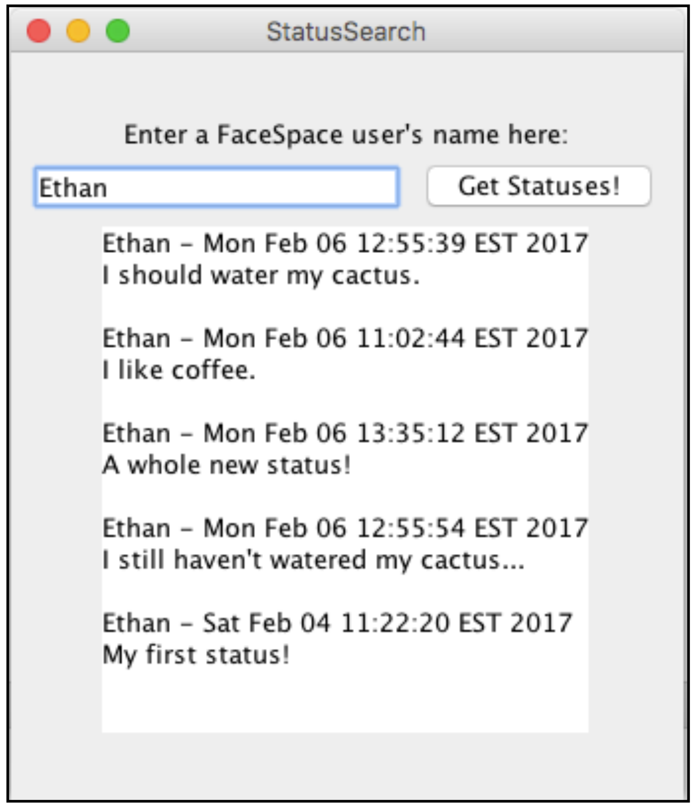
Since we are communicating with a Grails REST API, our Java UI will also strictly represent the V in MVC.

We will follow the same design principles wherever possible and highlight the differences.

Overall Design

Last week we used the following high-level design to help with our React UI implementation. Let's use the same approach for our Java version.





Enter a FaceSpace user's name here:

Ethan

Ethan – Mon Feb 06 12:55:39 EST 2017
I should water my cactus.

Ethan – Mon Feb 06 11:02:44 EST 2017
I like coffee.

Ethan – Mon Feb 06 13:35:12 EST 2017
A whole new status!

Ethan – Mon Feb 06 12:55:54 EST 2017
I still haven't watered my cactus...

Ethan – Sat Feb 04 11:22:20 EST 2017
My first status!

Java / Swing

Unlike React, Java does not dictate any rules about the flow of information in UI programs.

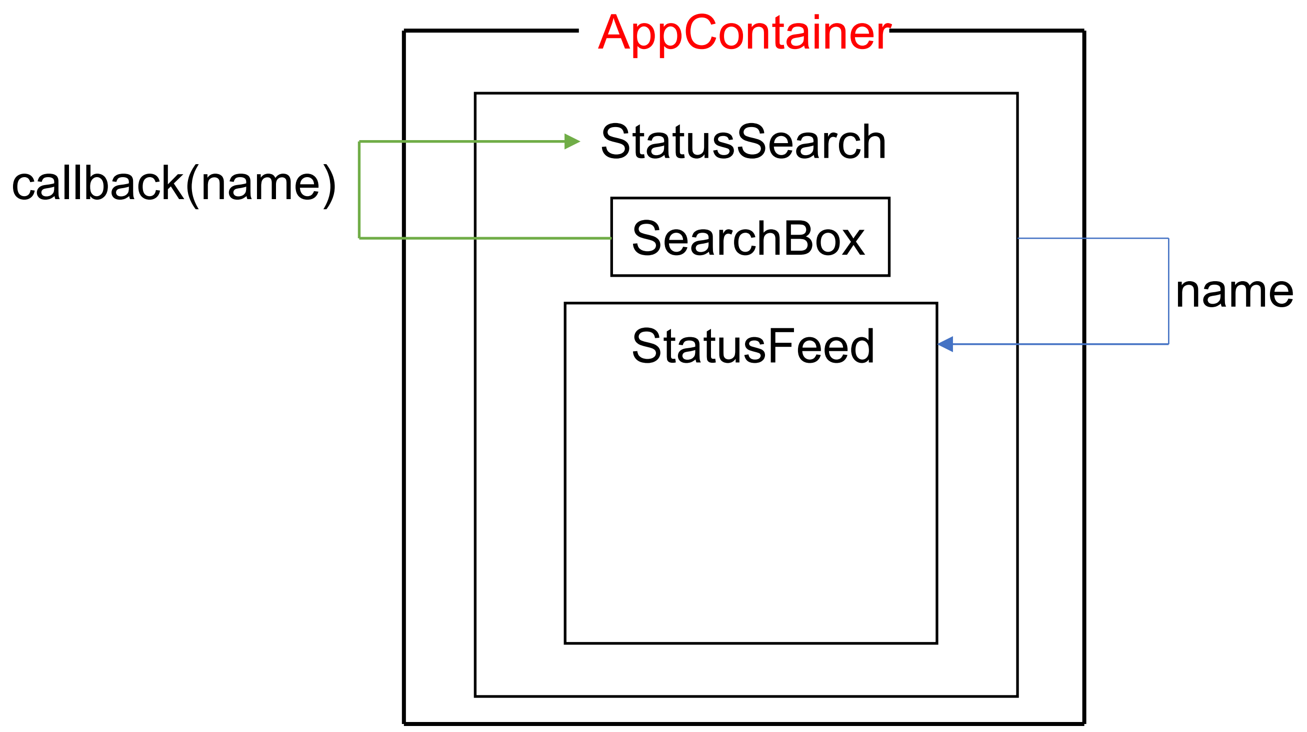
- Programmers must manage state, communication channels
- Any component may reference any other component
- No training wheels: more freedom → more challenging?

Java / Swing

Without a strict design pattern in place, we must take even more care in our design.

Let’s break down our high-level React design and determine whether it is also applicable to Java.

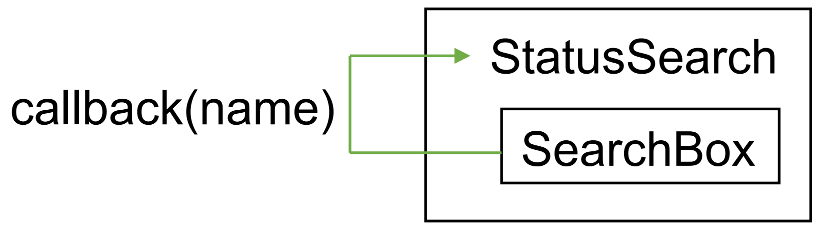
Java / Swing Example



Nested Components

- We can organize our Java program as a hierarchical composition of Swing components
- For this example, we can simplify StatusFeed by eliminating the need for a separate Status class – a status can be modelled internally using a String
- We will use inheritance to define these custom components

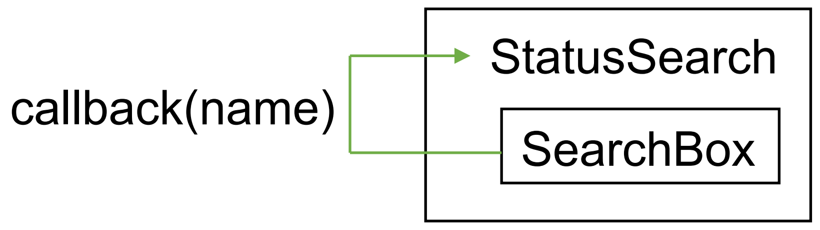
Java / Swing Data Flow Example



Similar to our data flow in React, a name needs to go from SearchBox back to StatusSearch before it can be passed into StatusFeed.

Similar again, we will implement **ActionListeners** in SearchBox to make a callback to StatusSearch once a new name has been submitted.

Java / Swing Data Flow Example



In StatusSearch, we define a variable called **searchName**.

We also define a method that will be called whenever a new name is submitted from SearchBox.

Ideally, this method would be passed to SearchBox as an instantiation parameter.

Java / Swing Data Flow Example

File: StatusSearch.java

```
interface Callback{
    void updateName(String newName);
}
```

```
public StatusSearch(JFrame root){
    super();
    …
    this.searchBox = new SearchBox(this::updateName);
    …
    this.statusFeed = new StatusFeed(root);
}
```

```
public void updateName(String newName){
    this.searchName = newName;
    this.statusFeed.updateName(this.searchName);
}
```

This code sets up the communication between StatusSearch and its child component SearchBox.

To enable callback, we define an interface that StatusSearch will implement.

In the constructor, we instantiate SearchBox with our desired callback method.

The updateName method takes the new name and passes it down to its other child component StatusFeed.

Java / Swing Data Flow Example

File: SearchBox.java

```
public class SearchBox extends JPanel {
    // Subcomponents
    private JTextField textEntry;
    private JButton submitButton;
    …
```

```
public SearchBox(Callback callback){
    …
    this.submitButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String text = textEntry.getText();
            callback.updateName(text);
        }
    });
    …
```

SearchBox (and StatusSearch) both extend the JPanel class. This makes them into Swing components that can be placed into a JFrame.

SearchBox has two child components of its own.

The constructor takes the callback **object** as a parameter. Then we are telling the program to call the method with the new name once the button has been pressed – very similar to our approach in React!

Java / Swing Data Flow Example

Now let’s look at our implementation for StatusFeed starting with the constructor. Similar to our use of React, we are initializing the component before any data is loaded.

File: StatusFeed.java

```
public class StatusFeed extends JTextArea {
    …
```

Our implementation of StatusFeed is as a subclass of JTextArea. This gives us all the methods and properties we need to render a StatusFeed to the UI.

Java / Swing Data Flow Example

File: StatusFeed.java

```
// A reference to the root JFrame (i.e. AppContainer)
// used only for calling its 'pack' method.
JFrame root;

// Basic initialization
public StatusFeed(JFrame root){
    super();
    this.setLayout(new GridLayout(0,1));
    this.root = root;
    this.setText(defaultText);
}
```

The reference root is meant to be used as a reference to the JFrame inside which the entire UI will be hosted. Having the reference in this class makes it possible to ask the JFrame to resize, for example.

Notice our use of the inherited method **setText**. Since the StatusFeed is just a subclass of JTextArea, we can use its methods directly to update status data.

Java / Swing Data Flow Example

File: StatusFeed.java

```
public void updateName(String username){
    this.setText(defaultText);
    this.setText(callAPI(username));
    this.root.pack();
    this.setVisible(false);
    this.setVisible(true);
}
```

This method will be called as a result of a SearchBox button click.

It will 1) clear the current text, 2) request new statuses from the API, and 3) redraw the component.

Calling the REST API

Though our UI design and implementation has so far mimicked our React UI quite closely, we still have not addressed a large issue: aside from ActionListeners, all of this program is **synchronous** or **blocking**.

By default, most Java libraries use such operations. In this example, our REST API calls will also be blocking.

To make a REST API call, we can use the Java.net set of classes. Unlike JavaScript, Java does not have a built-in JSON parser. We will use a package called GSON (by Google) to help.

GSON, Maven

GSON is an external library that needs to be imported into the project. You could do this from sources, or by adding a dependency to a Maven project.

Maven is a build automation tool, and it can be used to download dependencies for your project. The version of this tutorial posted on GitHub is already an IntelliJ/Maven project and is configured to download GSON automatically.

Calling the REST API

Before calling the API, we need to tell GSON how to interpret the responses. In this example, we are only asking the API for StatusPost objects. We need to write a corresponding class.

File: StatusFeed.java

```
private class StatusJSON{
    String statusText;
    Date dateCreated;
    Date dateLastEdited;
    public String toString(){
        return statusText;
    }
}
```

This class must reflect the EXACT names of your JSON object names, including case.

GSON will use this class as a template for converting JSON to instances of Java classes – very handy.

Calling the REST API

File: StatusFeed.java

```
public String callAPI(String name){
  try {
     // Initialize GSON
     Gson gson = new GsonBuilder().create();
     // Initialize result String
     String fromAPI = "";
     // Call the API
     URL myURL = new URL("http://localhost:8080/profileDisplay/getUserPosts?userName=" + name);
     URLConnection api = myURL.openConnection();
     // Call the API (ASYNCHRONOUS)
     api.connect();
     // Set up input stream reader
     BufferedReader in = new BufferedReader(new InputStreamReader(api.getInputStream()));
     String inputLine;
     …
```

Calling the REST API

File: StatusFeed.java

```
    …
    // Read from input steam
    while ((inputLine = in.readLine()) != null){
        // Declare the expected type of JSON results - in this case an array of Status objects
        StatusJSON[] statusArray = gson.fromJson(inputLine, StatusJSON[].class);
        // For each status found in JSON, append name, date, and status to result String
        for(StatusJSON sj : statusArray){
            fromAPI += String.format("%s - %s\n%s\n\n", name, sj.dateCreated, sj.statusText);
        }
    }
    // Close the stream and return the result
    in.close();
    return fromAPI;
}
```

Calling the REST API

File: StatusFeed.java

```
…
catch (MalformedURLException e) {
   // new URL() failed
   System.out.println("Bad URL...");
}
catch (IOException e) {
   // Catches refusals such as 'not found' or 'unauthorized'
   System.out.println("HTTP Error");
}
```

Typical try/catch blocks can be used to handle errors, including HTTP response codes.

Calling the REST API

Using a combination of GSON and the Java.net classes, we have written standard Java code to call the REST API and parse the JSON response.

Our only remaining task is to load the StatusSearch into a JFrame object so that our UI components can be displayed.

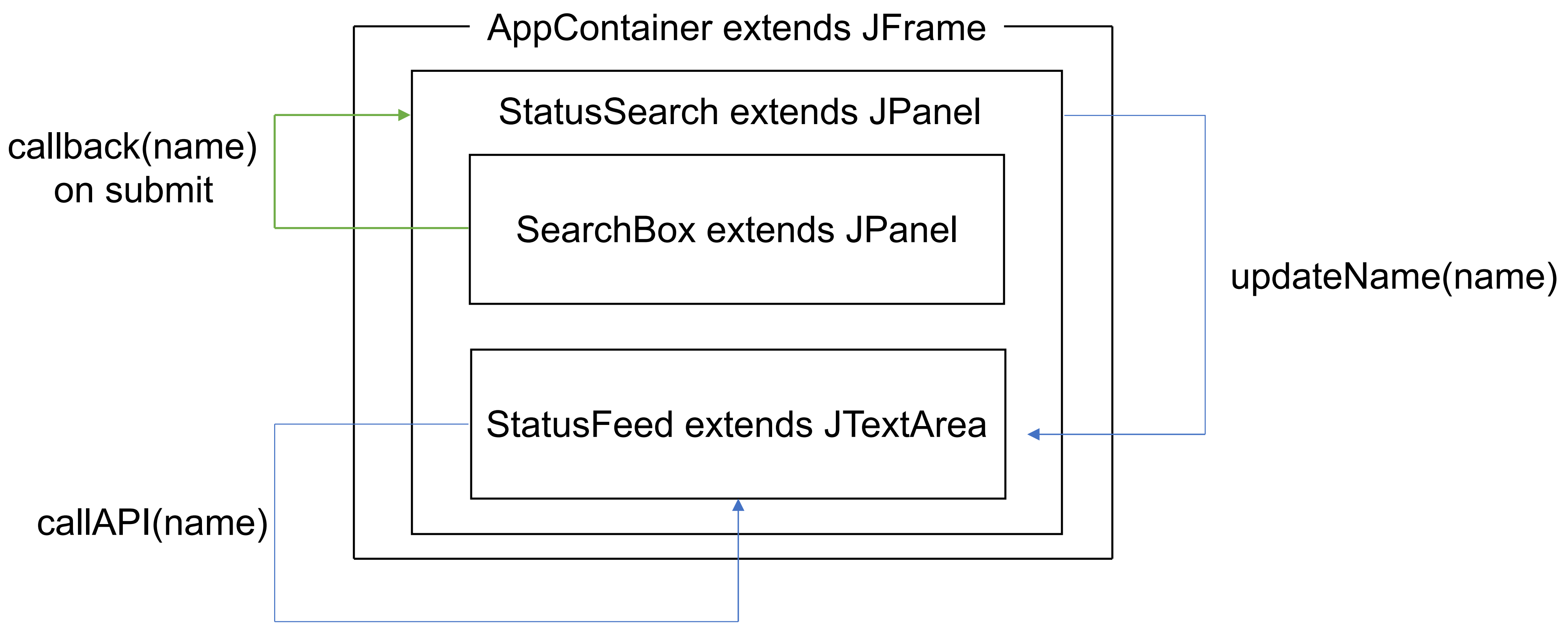
Creating a JFrame

```
public class AppContainer extends JFrame {

    // Subcomponents
    private StatusSearch statusSearch;

    public AppContainer(){
        super("StatusSearch");

        // Create and add the main component
        this.statusSearch = new StatusSearch(this);
        this.add(statusSearch);
        this.getContentPane().add(this.statusSearch);

        // Set size parameters, 'pack' everything tightly together, and set visible.
        this.setMinimumSize(new Dimension(340, 400));
        this.pack();
        this.setVisible(true);
    }
}
```

Using a similar strategy as other classes, we can create an AppContainer class (the main UI window) by extending JFrame.

Running the UI is then as simple as creating an instance of this class in the program main method.

Design Reprise


AppContainer extends JFrame
StatusSearch extends JPanel
SearchBox extends JPanel
StatusFeed extends JTextArea
callback(name)
on submit
updateName(name)
callAPI(name)

Your Tutorial Task

For this tutorial, you will implement a Java version of your AccountCreation component from the React tutorial.

Features

- Uses JTextField and JButton to accept username
- Makes POST request to Grails API from Tutorial 1 (has to be running to work)
- Displays success / error message based on response using JLabel

Your Tutorial Task

Notes

- Example code for StatusSearch is provided on GitHub
- Implement your AccountCreation UI as a single subclass of JFrame
- It must be contained in a single file called AccountCreation.java
- Create an instance of this class in the Main class
- Make sure you can run AccountCreation and StatusSearch simultaneously
- Search online to learn how to make a POST call using Java.net

Your Tutorial Task

Submission on OWL Due Friday Wed March 8th at 11:55pm

- Submit a single Java file called **AccountCreation.java** (2.5%)
- Must be compatible with example project from GitHub