

Swing JavaBuilder

Achieving maximum productivity with minimum code via declarative UIs

Jacek Furmankiewicz

Version 1.0.DEV

© Copyright 2008-2009 Jacek Furmankiewicz

Contents

Installation	6
Overview	7
YAML	7
Compact YAML syntax	9
Development tools	10
Benefits	10
Drawbacks	11
Swing JavaBuilder in 60 seconds or less	12
Core Features	17
Obtaining references to created components	17
Hooking up event listeners to Java methods	17
Databinding	18
Input validation	19
Executing long running methods on a background thread	20
Internationalization	22
Enum property values	22
Static int constant property values	23
Using custom components	23
Custom global commands	24
Build events	24
Hot deployment of UI components	25
Swing Features	26
Overview	26
Actions and menus	26
Borders	28
Button Group	28
Colors	29
Fonts	29
Icons and images	29
JComboBox	30
JFrame	30
JList	30
JScrollPane	30
JSplitPane	31
JTabbedPane	31
JTable	31
Event handlers	32
Swing Layout Management	35
MigLayout DSL	35
MigLayout	38
CardLayout	39
FlowLayout	39
Extending the JavaBuilders engine	40
Appendix	41
Layout Samples	41

Swing JavaBuilder - Achieving maximum productivity with minimum code via declarative UIs

Swing JavaBuilder Achieving maximum productivity with minimum code via declarative UIs

Abstract

Swing JavaBuilder : making Swing development productive

“Just started on using the Swing JavaBuilder and i must say i like it. Just replaced 170 rules of Java code with only 13 lines YAML” *Comment posted the JavaBuilders forum*

The Swing JavaBuilder is a library whose sole goal is to maximize the productivity of a Swing developer.

It's main goal is tackling all the Swing pain points, in particular the complexity and verbosity of the API and reducing it to the smallest amount of code possible.

This is accomplished by moving all the boring gruntwork of Swing interface creation to an external YAML file, which has a 1-to-1 match with a backing Java class (e.g. a JFrame or JPanel) that is built from that file. This allows to follow a pure MVC pattern where the YAML contains nothing but the view, while the Java class is (mostly) the controller.

As an added bonus, the Swing JavaBuilder offers integrated support for data binding (using Beans Binding), input validation (using Apache Commons Validators), background task processing (using SwingWorker) and last but not least, an integrated layout management DSL built-on top of the amazing MigLayout layout manager

In essence, the Swing JavaBuilder is an aggregator of best-of-breed Swing solutions into one common, integrated toolkit that makes creating Swing user interfaces a breeze

Note:

YAML is a file format that is a superset of JSON. We will cover it in more detail in future chapters. It's very simple to understand, edit and maintain. It's main advantage over both XML and JSON is the lack of any opening or closing tags, since it implements hierarchical data relationships via whitespace indentation (similar to the Python programming language).

License

All JavaBuilders code is released under the business-friendly Apache 2.0 license.

It is free to use in all projects, both open source and commercial.

Third party libraries

The Swing JavaBuilder depends on a number of well known open-source components, all of which are released under business-friendly licenses such as BSD, Apache or LGPL.

We never link to any open source components released under viral licenses such as GPL.

Nevertheless, please make sure to evaluate each third party license with your legal team to ensure compliance with its terms.

Preface

In 2007 or so, Sun Microsystems announced their JavaFX project, which aimed to deliver declarative UIs and rich desktop functionality. Unfortunately, in what I've always believed to be a severely misguided decision, this was accomplished by introducing a totally new language, instead of enhancing the core Java abilities and the existing Swing UI toolkit.

I decided that there had to be a middle-of-the-road approach that could give Java UI developers the productivity of declarative UIs without the need to throw out their current language skills out and focus on an unproved and untested new language (whose features I wasn't particularly fond of anyway, but that's a different story).

The JavaBuilders project was a result of this desire. It started off with many weeks of research and evaluation of different options. This resulted finally in the creation of a generic declarative UI based around the YAML format (which has many advantages over the XML or JSON formats) and the integration of many leading open source libraries (for features such as databinding or input validation) into one integrated solution.

The Swing JavaBuilder is the first production-ready implementation of the JavaBuilder engine, but it's generic nature allows it to be configured for other UI toolkits as well. In the future a SWT JavaBuilder is planned (and maybe even GTK+ and Qt versions as well).

I hope its adoption by you and your team will greatly increase your productivity and ensure a long and healthy future for Java rich client development.

Jacek Furmankiewicz

JavaBuilders Technical Architect

Installation

Start off with downloading the latest Swing JavaBuilder ZIP file distribution off the JavaBuilders.org website.

In the root folder you will find the Swing JavaBuilder jar and in the /lib folder you will find all of its dependencies. Add all of them to your project's classpath.

In the /samples folder you will find a sample application that you can use to get a better understanding of how you can build complex user interfaces using this library.

Overview

What is JavaBuilders all about?

In short any object that is built using a JavaBuilder consists of two files:

- a YAML text file that provides a declarative definition of the subject, most commonly the user interface. This would include items such as the controls that get instantiated, their properties, which methods should be called from event listeners, layout definition, data binding definition, predefined validations on controls or their properties.
- a Java class with all the actual code that represents the object being built. So for example, in Swing JavaBuilder the Java class may be a `JFrame` with all the relevant methods (e.g. `save()`, `close()`, `validateInput()`), as well as public properties that refer to the data being entered/maintained in the window).

Using a convention over configuration approach inspired by the Apache Wicket web framework, both files reside in the same package and with the same name, but just with a different file extension, e.g.:

```
MainApplicationFrame.java  
MainApplicationFrame.yaml
```

If you are using an inner class, e.g.

```
public class CommonPanels  
{  
    public static class SomePanel  
    {  
        SwingBuilder.build(this);  
    }  
}
```

then you can define a YAML file using the "`DeclaringClass.InnerClass.yaml`" format, e.g.

```
CommonPanels.SomePanel.yaml
```

in order to build an instance of the inner class.

Alternatively, you may even specify a build file explicitly by using the class-level `@BuildFile` annotation, which accepts a local or absolute file path within the classpath:

Local package file path

```
@BuildFile("Common.yaml")  
public class LocalBuildFilePanel extends JPanel
```

or:

Absolute file path

```
@BuildFile("/org/javabuilders/test/resources/Common.yaml")  
public class GlobalBuildFilePanel extends JPanel
```

Why would I use this instead of regular coding by hand?

Because you will have to write a lot less code to the same thing if you use a JavaBuilder. This is what it's all about.

Note:

The YAML file contains only a declaration of the interface, which methods (on the Java side) should be fired when the user pressed a button, data binding instructions, data validation definitions, etc. It has zero code (of any type, Java, Javascript, etc.) embedded in it. The idea is that 100% of actual code you write is in the Java file and nowhere else.

YAML

What is YAML?

I discovered YAML while reading about Ruby on Rails. It is used by that web framework as the default file format for all configuration files. It has a very simple approach to define hierarchical data structures/maps/list, based on straightforward whitespace indentation. Also, it handles text transparently. There is usually no need to input text in quotes, you can just type it as is, e.g.

```
text: This is the text for my control
```

The only time you need to escape into quotes is if your text contains YAML-reserved characters such as ":", e.g.

```
text: "First name:"
```

Whitespace indentation

Unless you are a Python programmer, the concept of anything that relies on whitespace probably makes you uncomfortable. Trust me, it's actually very simple to get used to it, does not require any particularly specialized development tools. The main benefit of whitespace indentation is that it automatically handles defining the "end" of an item (hence there is no need for XML-closing tags or JSON-closing brackets).

Why not XML?

It is simply too verbose. Too much typing. Most of the file seems to be tags and closing tags instead of the content. In YAML the majority of the file is the actual content (and the whitespace of course).

Why not JSON?

JSON is very concise and the perfect tool for let's say invoking Ajax requests. However, for maintainable files it suffers from what I call "closing bracket hell", especially when dealing with complex object graphs. Every type needs to be closed with a "}" and every collection needs to be opened and closed with a "[" and "]". Once you start mixing the two together you start having horrendous closing statements such as this:

```
}
}
}
}
}
}
}
}
```

Scroll to the bottom of this JavaFX code sample to see what I mean:
<http://jfx.wikia.com/wiki/JFXPresentation>

YAML is a superset of JSON

Although YAML relies on whitespace indentation to indicate hierarchy, you can at any point in the document switch to JSON-style brackets. This allows to keep the file shorter and more concise and should be used on all bottom-level nodes (i.e. those that have no children).

Pure whitespace YAML example:

```
JFrame:
  name: myFrame
  title: My Frame
  content:
    - JLabel:
      name: myLabel2
      text: My First Label
    - JLabel:
      name: myLabel2
      text: My Second Label
```

The same content can be compressed using JSON-style brackets to:

```
JFrame:
  name: myFrame
  title: My Frame
  content:
    - JLabel: {name: myLabel2, text: My First Label}
    - JLabel: {name: myLabel2, text: My Second Label}
```

However, in most cases you will not be coding in either traditional YAML or JSON. We have enhanced the standard YAML syntax to make it even more compact (more on that in the next sections). In most cases your YAML content will look like this:

```
JFrame(name=myFrame,title=My Frame):
  - JLabel(name=myLabel2, text=My First Label)
  - JLabel(name=myLabel2, text=My Second Label)
```

This is still valid YAML syntax and our custom YAML pre-processor takes care of "exploding" this compact syntax to the equivalent "full" YAML content

Tabs in YAML

Tabs are simply not allowed in YAML, period. You always indent using explicit whitespace. Putting a tab into a YAML file will cause it to fail to parse

YAML syntax samples

Values:

```
text: Some text
```

Maps:

```
JFrame:
  name: myFrame
  title: My Frame
```

Lists (via the "-" indicator):

```
content:
  - Item1
  - Item2 : {somePropertyForItem2: someValueforItem2}
```

Free-form text with new lines preserved (accomplished with the "|" indicator):

```
quote: |
  To code by hand or not?
  There is no question.
  You should just be using JavaBuilders.
    Will Shakespeare (JavaBuilders early adopter)
```

Related links

YAML on Wikipedia: <http://en.wikipedia.org/wiki/YAML>

Compact YAML syntax

Although the base YAML format is already pretty concise, JavaBuilders adds a custom extension to it that we call "virtual constructor flow", otherwise referred to simply as compact YAML. It allows to specify the child properties of an object in the same line of text as the object definition.

Here's a pure YAML example:

```
JFrame:
  name: frame
  title: My Frame
  content:
    - JButton:
        name: buttonClose
        text: Close
        onAction: close
    - JButton:
        name: buttonSave
        text: Save
        onAction: save
```

The same content can be entered in much less lines using our compact syntax:

```
JFrame(name=frame,title=My Frame):
  - JButton(name=buttonClose,text=Close,onAction=close)
  - JButton(name=buttonSave,text=Save,onAction=save)
```

Let's be clear: this is not part of the official YAML standard. This is something specific to JavaBuilders that was added to make the YAML file even smaller.

Basic concepts

- properties and their values are entered between (and) on the same line as the object they refer to
- instead of the default YAML "name: value" format it uses "name=value"
- instead of the default YAML collection indicator [and] (e.g. "list: [listItem1, listItem2]") it uses regular brackets, e.g. "list=(listItem1,listItem2)"
- if an object has a collection of object defined directly underneath it, they automatically get moved to the default "content" node (just as in the example shown above)

Note:

All the code samples from this point will use the compact syntax, in order to promote its use.

Development tools

JavaBuilders requires just any decent Java IDE with a YAML editor. Remember to select fixed width font (e.g. Courier New, Monospaced) for the editor, otherwise you will not be able to line up the spacing correctly in the file.

Eclipse

Eclipse YAML Editor: <http://code.google.com/p/yamleditor/>

NetBeans

As of NetBeans 6.5 a YAML editor is included in the core distribution.

IntelliJ IDEA

A YAML editor is included in the core distribution.

Benefits

What are the benefits compared to coding by hand?

You have to write a lot less code. JavaBuilders introduces dynamic language-level productivity (think Ruby/Groovy) to Java. See this typical Java Swing example:

```
ResourceBundle bundle = ResourceBundle.getBundle("Resources");

JButton button = new JButton();
button.setName("okButton");
button.setText(bundle.getString("button.ok"));
button.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e) {  
            //execute the save method  
            save();  
        }  
    }  
};
```

The equivalent compact YAML content would be just:

```
JButton(name=okButton,text=button.ok,onAction=save)
```

and all you need to build this Swing Java class from this YAML file is this single line of code somewhere in your constructor:

```
SwingJavaBuilder.build(this);
```

The equivalent code for any other UI toolkit (e.g. SWTJavaBuilder) would be just as compact.

What are the benefits compared to using GUI Builders, such as NetBeans Matisse?

Mostly maintainability. For smaller examples it's probably not much of a difference (since so much of the code is generated for you by Matisse), but once you get into larger, more complex forms it becomes harder to maintain them in a GUI builder, especially if you have to move the layout around a lot. In JavaBuilders, it's just a matter of changing a few lines of text in a YAML file.

Also, e can add "custom" properties to existing objects, so we can enhance APIs or make them easier, e.g.:

```
JFrame(size=800x400)
```

The Swing JFrame class does not have a property called "size". But JavaBuilders can support virtual properties which trigger some Java code that will magically call the proper equivalent methods, in order to achieve the same functionality in much less code.

Last, but not least, JavaBuilders provide support for functionality not provided by GUI builders, such as integrated input validators or executing cancellable long running methods on a background thread.

Drawbacks

Nothing is perfect, so JavaBuilders have weak points too.

- Lose some of the static, compile-time safety: since you are defining all the layouts/event wiring in a YAML text file, some of the referenced objects may have a different name than their corresponding equivalents in the Java file, especially if using refactoring. This can be overcome with the `@Alias` annotation, which hardcodes a link between a Java-side object and its definition in the YAML file.
- No code completion (at least not yet). YAML is just a pure text file. You won't know what the known properties are for any particular object type unless you know them already. But in most cases it's the basic ones: name, text, onAction, onClicked, etc.
- You have to get acquainted with YAML...sorry, can't help you there. Sometimes we just need to learn new things. The bottom line though is that all your code stays in Java, YAML is just used for declarative UI building.

On the upside, UI components built with JavaBuilders are easily unit testable. You just need to do `new MyComponent()` in your unit test, that's all. When an object is built, the JavaBuilder automatically validates that not only the properties are defined correctly, but also all the event listeners point to actual existing methods in the Java class. If not, a `BuildException` will be thrown right away.

Swing JavaBuilder in 60 seconds or less

Is it worth my time?

Here's a sample of what you can do with the Swing JavaBuilder in 60 seconds or less. Hopefully, it will make it clear as to what the productivity benefits are.

To show off its abilities we will create a simple app that prompts for a person's data and simulates saving it to a database via a long running task on a background thread.

1. Download the latest Swing JavaBuilder ZIP from <http://javabuilders.org>
2. In Eclipse, create a new Java project called "PersonApp" and create a default package "person.app"
3. Add the Swing JavaBuilder jar and all of its dependencies (from the "/lib" folder) to the project's build path
4. Create the Person class that will represent our model
person/app/Person.java

```
package person.app;

import java.text.MessageFormat;

public class Person {

    private String firstName;
    private String lastName;
    private String emailAddress;

    /**
     * @return the firstName
     */
    public String getFirstName() {
        return firstName;
    }
    /**
     * @param firstName the firstName to set
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    /**
     * @return the lastName
     */
    public String getLastName() {
        return lastName;
    }
    /**
     * @param lastName the lastName to set
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    /**
     * @return the emailAddress
     */
    public String getEmailAddress() {
        return emailAddress;
    }
    /**
     * @param emailAddress the emailAddress to set
     */
    public void setEmailAddress(String emailAddress) {
        this.emailAddress = emailAddress;
    }

    @Override
    public String toString() {
        return MessageFormat.format("{0} {1} : {2}", getFirstName(), getLastName(),
        getEmailAddress());
    }
}
```

5. Create a "PersonApp.properties" file in the root package with the internationalized resources:
PersonApp.properties

```
button.save=Save
button.cancel=Cancel

label.firstName=First Name:
```

```
label.lastName=Last Name:
label.email=Email
```

6. Create the view YAML file PersonApp.yaml: **person/app/PersonApp.yaml**

```
JFrame(name=frame, title=frame.title, size=packed, defaultCloseOperation=exitOnClose):
- JLabel(name=fNameLbl, text=label.firstName)
- JLabel(name=lNameLbl, text=label.lastName)
- JLabel(name=emailLbl, text=label.email)
- JTextField(name=fName)
- JTextField(name=lName)
- JTextField(name=email)
- JButton(name=save, text=button.save, onAction=($validate,save,done))
- JButton(name=cancel, text=button.cancel, onAction=($confirm,cancel))
- MigLayout: |
    [pref]      [grow,100]    [pref]      [grow,100]
    fNameLbl    fName        lNameLbl    lName
    emailLbl    email+*
    >save+*=1,cancel=1
bind:
- fName.text: this.person.firstName
- lName.text: this.person.lastName
- email.text: this.person.emailAddress
validate:
- fName.text: {mandatory: true, label: label.firstName}
- lName.text: {mandatory: true, label: label.lastName}
- email.text: {mandatory: true, emailAddress: true, label: label.email}
```

7. Create the controller Java class PersonApp

```
package person.app;

import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;

import org.javabuilders.BuildResult;
import org.javabuilders.annotations.DoInBackground;
import org.javabuilders.event.BackgroundEvent;
import org.javabuilders.event.CancelStatus;
import org.javabuilders.swing.SwingJavaBuilder;

public class PersonApp extends JFrame {
    private Person person;
    private BuildResult result;

    public PersonApp() {
        person = new Person();
        person.setFirstName("John");
        person.setLastName("Smith");
        result = SwingJavaBuilder.build(this);
    }

    public Person getPerson() {
        return person;
    }

    private void cancel() {
        setVisible(false);
    }

    @DoInBackground(cancelable=true, indeterminateProgress=false, progressStart=1,
progressEnd=100)
    private void save(BackgroundEvent evt) {
        //simulate a long running save to a database
        for(int i = 0; i < 100; i++) {
            //progress indicator
            evt.setProgressValue(i + 1);
            evt.setProgressMessage("" + i + "% done...");

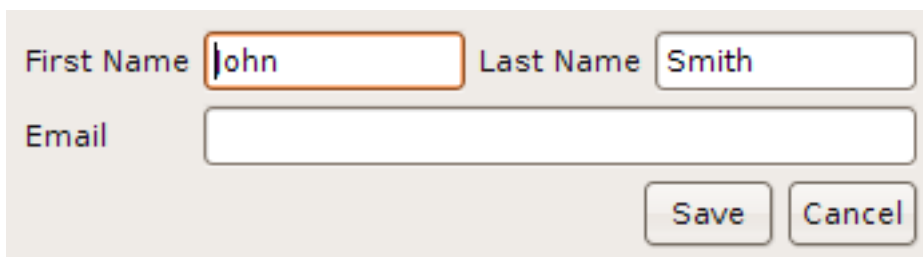
            //check if cancel was requested
            if (evt.getCancelStatus() != CancelStatus.REQUESTED) {

                //sleep
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {}
            } else {
                //cancel requested, let's abort
                evt.setCancelStatus(CancelStatus.COMPLETED);
                break;
            }
        }
    }
}
```

```
//runs after successful save
private void done() {
    JOptionPane.showMessageDialog(this, "Person data: " + person.toString());
}

/**
 * @param args
 */
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            //activate internationalization
            SwingJavaBuilder.getConfig().addResourceBundle("PersonApp");
            try {
                UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                new PersonApp().setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}
```

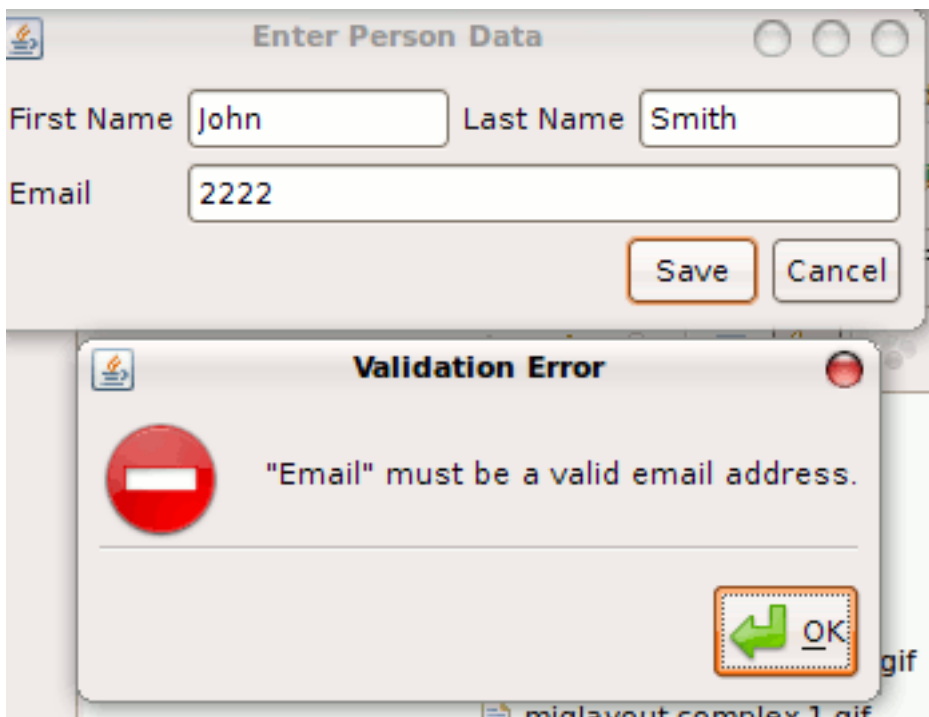
8. Run the PersonApp.main() method. You should see an input dialog like this appear:



The image shows a standard Java Swing dialog box titled "Enter Person Data". It contains three text input fields: "First Name" with the value "John", "Last Name" with the value "Smith", and "Email" which is currently empty. At the bottom right of the dialog are two buttons labeled "Save" and "Cancel".

Notice that the default person name is propagated from the Java code to the UI via data binding. Also, all the controls are created and the layout is executed without the need for an IDE-specific GUI builder.

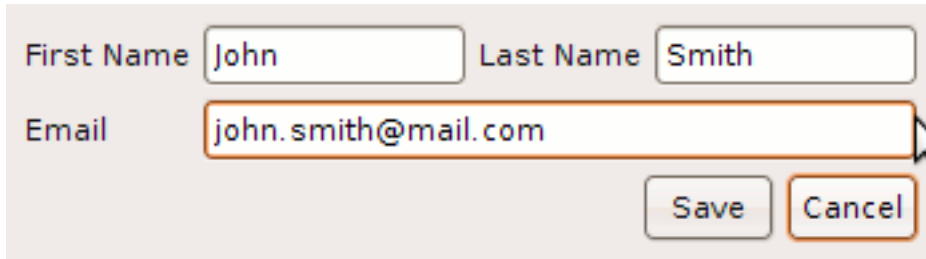
9. Enter an invalid email address for the person and press Save:



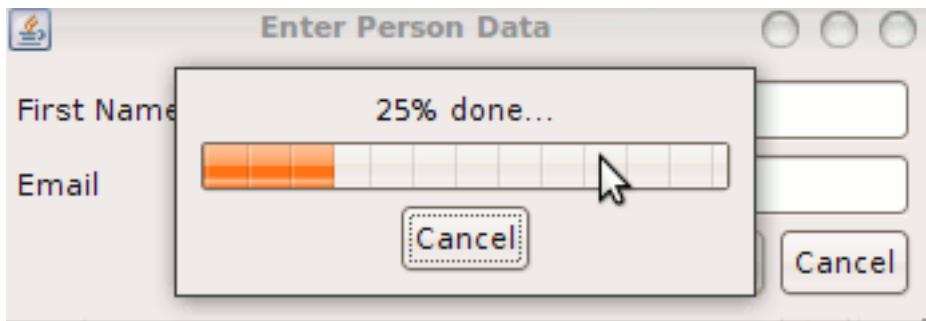
The image shows the "Enter Person Data" dialog box with the "Email" field now containing the text "2222". A "Validation Error" dialog box is overlaid on top of it. The "Validation Error" dialog has a red circular icon with a white exclamation mark and the text "Email must be a valid email address." At the bottom right of the "Validation Error" dialog is an "OK" button with a green arrow icon.

The validation logic (invoked via "\$validate") executed and perform basic input validation.

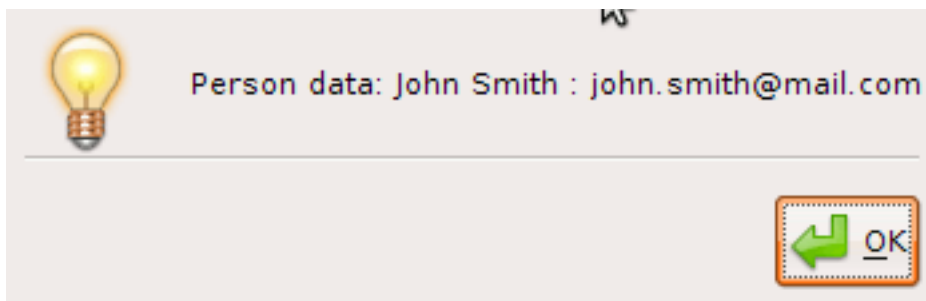
10. Enter a valid email address:



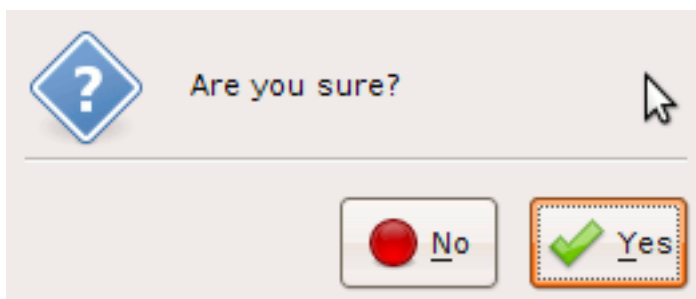
11. Press "Save". The `save()` Java method is executed (which simulates a long running database save with a progress bar) and since it is annotated with the `@DoInBackground` annotation it will automatically run on a background thread using the `SwingWorker` library.



12. After the save logic executes, the `done()` Java method is executed to inform the user the save was successful. Notice that the email address we entered was automatically propagated back to the underlying bean using databinding.



13. Press 'Cancel' to close the window. Since you specified `"$confirm"` in the action handler, it will automatically prompt the user to confirm the action. If they select "Yes", the `cancel()` Java method will be called and the window will close.



Summary

- 22 lines of YAML
- 3 simple Java methods to handle `save()`, `done()` and `cancel()` (and without any of the logic to create and layout the controls)

That is all we needed to create a fully functional application with control creation and layout, data input validation and executing long running business methods on a background thread via `SwingWorker`. Not to mention it's fully localized with all the labels being automatically fetched from a `ResourceBundle`

Swing JavaBuilder - Achieving maximum productivity with minimum code via declarative UIs without any additional coding overhead.

Core Features

Obtaining references to created components

Convention over configuration

In most cases, we use a straightforward convention-over-configuration approach. If you define an object in YAML and then define a Java instance instance variable with the same name and of compatible type, then JavaBuilders will set the reference on it automatically (even if it is a private variable, it does not need to be public).

Simple example:

MyFrame.yaml

```
JFrame:
  - JButton(name=okButton,text=OK,onAction=save)
```

MyFrame.java

```
public class MyFrame extends JFrame {
    //this object's reference will be set automatically
    private JButton okButton;
    private BuildResult result = SwingJavaBuilder.build(this)
    public MyFrame(){
        //reference is set! NullPointerException will not occur
        okButton.setText("New text");
    }
    private void save() {
        //execute some business logic...
    }
}
```

Obtaining references manually

You can also just fetch the object reference manually from the returned `BuildResult` object:

```
public MyFrame() {
    JButton okButton = (JButton)result.get("okButton");
}
```

But the convention over configuration approach is much preferred.

Hooking up event listeners to Java methods

Overview

The standard approach is to provide a standard "onEvent" property (e.g. "onAction", "onClicked", "onDoubleClicked") and then pass it a single method name or a collection of method names.

Single method

```
JButton(text=OK, onAction=save)
```

Multiple methods to be executed in sequence

```
JButton(text=OK, onAction=(validateInput,save,close))
```

If any of the methods return a boolean **false** , then the other methods get aborted and will not be called.
Simple convention over configuration approach

Mapping to methods on the Java side

When you specify a method name (e.g. "save") in the YAML file, it will attempt to execute the corresponding method in the Java class. Different signatures of the method are supported, in order of preference:

method(calling object type or its superclass, event specific class)

```
private void save(JButton button,(ActionEvent event) {}
```

method(event specific class)

```
private void save(ActionEvent event) {}
```

method(calling object type or its superclass)

```
private void save(JButton button) {}
```

method()

```
private void save() {}
```

Enter whichever one you want and JavaBuilders will find it and execute it. If it finds multiple ones, it will execute the first one it finds based on the preference above. If none are found, a BuildException will be thrown **right away during build time**. So, you do not have to actually test your event listener logic by manually clicking on the button or menu item, the validation occurs right away as part of the build process. This simplifies unit testing and limits the risk of lost type safety.

Databinding

Binding is defined by adding a "bind" root node after all the controls have been defined. Unlike in most other languages, the binding is not defined at the property level, but is a stand-alone node of its own. This is done to enforce separation of concerns and ensure clarity. You can see all your data binding in one place, all together.

Sample (assume we have a backing JFrame JavaBean with two public properties "lastName" and "firstName"):

```
JFrame(name=frame,title=Hallo):
  content:
    - JLabel(name=firstNameLabel, text="First name", labelFor=firstNameField)
    - JTextField(name=firstNameField)
    - JLabel(name=lastNameLabel, text="Last name", labelFor=lastNameField)
    - JTextField(name=lastNameField)
    - JButton(name=saveButton, text=Save)
  layout: |
    {} [grow]
    >firstNameLabel    firstNameField
    >lastNameLabel     lastNameField
    >saveButton+*
  bind:
    - this.title : "Hello, ${firstNameField.text}" #bind the frame title to show last name
    using an EL expression
    - this.firstName : firstNameField.text #bind the public Java property to the text
    field's text value, using a direct simple expression
```

Databinding requirements

In order for the binding to work between public properties, they must fire a "property change" event on the "set" and the parent class must provide the "addPropertyChangeListener" and "removePropertyChangeListener" methods. This is all part of the standard Beans Binding requirements.

A good example can be found in the Bound Properties Java tutorial:
<http://java.sun.com/docs/books/tutorial/javabeans/properties/bound.html>

Supported features

In order to integrate as best as possible with each UI toolkit, JavaBuilders rely on the best toolkit-specific library for databinding. This means that the Swing JavaBuilder uses Beans Binding (JSR 295), while the SWT JavaBuilder uses JFace DataBinding.

Not all databinding engines provide the same functionality. For example, Beans Binding does provide support for EL expressions in data binding (hence you can use them for the Swing JavaBuilder), but the JFace DataBinding engine does not (and therefore they are not supported for the SWT JavaBuilder).

Input validation

Similar to data binding, input validation is configured via a separate root level node called `validate`:

```
JFrame(name=frame,title=Binding Frame,size=packed):
  content:
    - JLabel(name=fnLbl, text="First name:")
    - JTextField(name=fName)
    - JLabel(name=lnLbl, text="Last name:")
    - JTextField(name=lName)
    - JButton(name=ok, text=OK, onAction=($validate,save,cancel))
    - JButton(name=cancel, text=Cancel, onAction=cancel)
    - MigLayout:
      [
        [grow,200px]
        >fnLbl fName
        >lnLbl lName
        >ok*=1,cancel=1      [grow,bottom]
      ]
  bind:
    - firstName: fName.text
    - lastName: lName.text
  validate:
    - fName.text: {label: First Name, mandatory: true, minLength : 5}
```

Invoking input validation

If you want to do in from the YAML file, just put `$validate` as the method name in any event handler, e.g.

```
JButton(name=saveBtn,text=Save,onAction=($validate,save,close))
```

If you want to do it from the Java then you just need to call the `validate()` method on the `BuildResult` object that was returned:

```
private BuildResult result = SwingJavaBuilder.build(this);
//validate user input
private boolean validate() {
    return result.validate();
}
```

Field label for error messages

The "label" property is used to define the name of the field that will be using in any error messages. It is localizable, so you can sent it a resource key instead, e.g.:

```
validate:
  - fName.text: {label: label.firstName, mandatory: true, minLength : 5}
```

Validator routines

The following validator routines are currently available

Validation type	Example	Comment
mandatory	mandatory: true	
minLength	minLength : 5	

maxLength	maxLength : 5	
regex	regex: "[a-zA-Z0-9]+"	Uses default validation message
	regex: "[a-zA-Z0-9]+", regexMessage: "'{0}' must be a number or letter"	Uses custom error messagee
minValue	minValue: 5	
maxValue	maxValue: 50	
dateFormat	dateFormat: yyyy/mm/dd	
emailAddress	emailAddress: true	

Full example

```
validate:
- mandatory.text: {label: Mandatory Field, mandatory: true}
- date.text: {label: Date Field, dateFormat: "yyyy/mm/dd"}
- email.text: {label: E-Mail, email: true}
- minmax.text: {label: Min/Max Length, minLength: 5, maxLength: 10}
- regex.text: {label: Regex, regex: "[a-zA-Z0-9]+"}
- regex2.text: {label: Regex, regex: "[a-zA-Z0-9]+", regexMessage: "'{0}' must be a number or letter"}
- long.text: {label: Min/Max Long, minValue: 5, maxValue: 50, mandatory: true}
```

Adding custom validators

The default validator routines not powerful enough for you? You can easily add custom validation logic to be executed together with the built-in routines via Java-side code:

```
result.getValidators().add(new IValidator() {
    public void validate(Object value, ValidationMessageList list) {
        if (!isValid) {
            list.add(new ValidationMessage("Input is not valid 'cause I say so"));
        }
    }
});
```

Executing long running methods on a background thread

A common issue in most UI toolkits is that the application locks up if a long running process is running on the EDT (Event Dispatch Thread). In this case, the recommended solution is to execute it on a background thread and if possible, provide some sort of progress indicator to the user letting them know about the current status of this process (e.g. saving large amounts of data to a database).

Method Annotation

In JavaBuilders, this is accomplished by simply annotating the long running method with a `@DoInBackground` annotation (which provides some attributes that can customize how the long running process is handled).

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface DoInBackground {
    /**
     * @return Progress message
     */
    String progressMessage() default "label.processing";

    /**
     * @return If background task is cancelable or not
     */
    boolean cancelable() default false;

    /**
     * @return Default start value for progress bar
     */
}
```

```

    int progressStart() default 1;
    /**
     * @return Default end value for progress bar
     */
    int progressEnd() default 100;
    /**
     * @return Current progress value
     */
    int progressValue() default 1;
    /**
     * @return Indicates if task should block UI with a popup or not
     */
    boolean blocking() default true;
    /**
     * @return Indicates to show indeterminate progress indicator. Overrides the progress
     start/end/value if set to true
     */
    boolean indeterminateProgress() default true;
}

```

Any method that is annotated as such must implement a signature that accepts an object of type `BackgroundEvent`, which allows the background method to communicate with the UI's progress indicator and even cancel itself, if the user requests it, e.g.:

```

@DoInBackground(cancelable=true, progressStart=1, progressEnd=100, progressValue=1,
indeterminateProgress=false)
private void save(BackgroundEvent evt) {
    System.out.println("SAVE...");
    for(int i = 0; i < 100; i++) {
        if (evt.getCancelStatus() != CancelStatus.REQUESTED) {
            try {
                Thread.currentThread().sleep(100);
                evt.setProgressValue(i + 1);
                evt.setProgressMessage(String.format("Processing %s of %s...",
                    evt.getProgressValue(), evt.getProgressEnd()));
            } catch (InterruptedException e) {}
        } else {
            evt.setCancelStatus(CancelStatus.PROCESSING);
            System.out.println("Cancelling...");
            evt.setCancelStatus(CancelStatus.COMPLETED);
            break;
        }
    }
    System.out.println("SAVE END...");
}

```

Executing multiple methods together

A typical scenario in an input dialog that occurs when a user presses the Save button is:

1. validate input
2. save the data (this can take a long time)
3. close the window

The way to handle this is to have the button execute multiple methods in sequence, e.g.:

```

JButton(text=Save,onAction=($validate,save,close))

```

On the Java side, the long running method is annotated as such:

```

@DoInBackground(indeterminateProgress=true)
private void save() { //long running process }

private void close() {
    setVisible(false);
    dispose();
}

```

The methods **after** the long running method (i.e. "close" in this example), will only execute **after** the long running method has finished, they will not run in parallel, even though they are on different threads. Hence, the sequence of events is preserved.

Domain-specific Implementations

In Swing JavaBuilder, long running methods are handled by using the standard `SwingWorker` library. A Swing progress dialog will popup up informing the user that a process is running. If the method flagged itself as cancelable, the Cancel button on the progress dialog will be enabled, allowing the user to cancel

Swing JavaBuilder - Achieving maximum productivity with minimum code via declarative UIs
the task if it runs for too long.

For the SWT JavaBuilder the plan is to support something similar or alternatively plug into the JFace Progress/Tasks API.

As you can see, JavaBuilders does not have a "one size fits all" approach and for each toolkit we plan to use the best option available on that specific platform.

Internationalization

Internationalization support in any Builder is provided at two levels: global and class-level. If any resource bundle is present (either at the global or class level), the internationalization support will automatically get activated.

Global Resource Bundles

In your main() just add the list of global application resource bundles to the configuration of your builder, e.g.:

```
SwingJavaBuilder.getConfig().addResourceBundle("Resources");
```

or:

```
ResourceBundle myResourceBundle = ....  
SwingJavaBuilder.getConfig().addResourceBundle(myResourceBundle);
```

Class-level Resource Bundles

If you need to have additional class-level resource bundles, just pass them in during the build request:

```
private ResourceBundle bundle = ResourceBundle.getBundle("MyClassBundle");  
private BuildResult result = SwingJavaBuilder.build(this, bundle);
```

The builder will look at the class-level bundles first for a key and if not found, will search through the global ones.

Usage

Once you register a resource bundle, you can pass a resource name directly to any of the properties that have been flagged as localizable, e.g.

YAML

```
JButton(name=okButton, text=button.ok)
```

Properties file

```
button.ok=OK
```

Enum property values

When building an object, if the specified property type is an Enum of any sort, the builder will automatically allow you to enter it just using the enum constant, without the actual enum name prefix.

Enums defined like constant Integers

```
//enum defined like static int constants  
enum StartPosition{ CENTER_IN_SCREEN, CENTER_IN_PARENT, MANUAL }
```



```
JXFrame.setStartPosition(StartPosition position)
```

In YAML, you can do then either:

```
JXFrame(startPosition=CENTER_IN_PARENT)
```

or the Java camel-case named equivalent:

```
JXFrame(startPosition=centerInParent)
```

Enums defined using Pascal case

If your enum is defined instead using a Pascal case syntax, e.g.

```
//enum defined like static int constants  
enum StartPosition{ CenterInScreen, CenterInParent, Manual}
```

then you can still do either the original constant value or the camel-case named equivalent:

```
JXFrame(startPosition=CenterInParent)
```

or:

```
JXFrame(startPosition=centerInParent)
```

Static int constant property values

Similar to the way Enum values are handled, the default behaviour is that when a String value is passed to an `int` property, the builder will attempt to find a corresponding `final static int` value on the Java class and use that. Both camel-case values and actual static constant names can be used, e.g.:

```
JFrame(defaultCloseOperation=EXIT_ON_CLOSE)
```

or:

```
JFrame(defaultCloseOperation=exitOnClose)
```

Using custom components

Sooner or later you will want to create a custom component instance from within your YAML file. However, the current builder does not know how to map your custom component name (e.g. "MyCustomPanel") to an actual Java class.

In order to let it know all you have to do is define an instance variable with the same type in your Java-side code and it will automatically find the corresponding class definition that way, e.g.

YAML

```
JFrame(title=frame.title,state=max,defaultCloseOperation=exitOnClose):  
  - ComponentsPanel(name=componentsPanel,tabTitle=tab.components)  
  - BorderPanel(name=borderPanel,tabTitle=tab.borders)  
  - CardLayoutPanel(name=cardLayoutPanel,tabTitle=tab.cardLayout)  
  - FlowLayoutPanel(name=flowLayoutPanel,tabTitle=tab.flowLayout)  
  - MigLayoutPanel1(name=migLayoutPanel1,tabTitle=tab.migLayout1)
```

Java

```
private ComponentsPanel componentsPanel;
private FlowLayoutPanel flowLayoutPanel;
private CardLayoutPanel cardLayoutPanel;
private MigLayoutPanel1 migLayoutPanel1;
private BorderPanel borderPanel;
```

Custom global commands

Custom global commands allows you to basically define a named reusable piece of code that you can refer to anywhere in your YAML file's event handlers.

Custom commands are prefixed with "\$" and the system ships with two pre-defined global commands:

- **\$validate** : triggers the input validation logic, if defined

```
 JButton(name=okButton, text=OK, onAction=($validate,save,finishSave))
```

- **\$confirm** : displays a standard "Are you sure?" confirmation dialog that can be invoked before any destructive action:

```
 JButton(name=deleteButton, text=Delete, onAction=($confirm,delete))
```

Adding your own custom commands

You need to implement the `ICustomCommand` interface and add it to your builder's configuration:

```
SwingJavaBuilder.getConfig().addCustomCommand("$confirm", new ICustomCommand<Boolean>() {
    public Boolean process(BuildResult result, Object source) {
        Component c = null;
        if (result.getCaller() instanceof Component) {
            c = (Component) result.getCaller();
        }
        int value = JOptionPane.showConfirmDialog(c,
            Builder.getResourceBundle().getString("question.areYouSure"),
            Builder.getResourceBundle().getString("title.confirmation"),
            JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
        if (value == JOptionPane.YES_OPTION) {
            return true;
        } else {
            return false;
        }
    }
});
```

Build events

If you need to hook up some custom pre- or post-processing every time a build is executed (e.g. to integrate a 3rd party library like [JavaCSS](#) , you can add a listener to the builder), preferably in your `main()` method , e.g.:

```
//event listeners
SwingJavaBuilder.getConfig().addBuildListener(new BuildAdapter() {
    @Override
    public void buildStarted(BuildEvent evt) {
        System.out.println(("Build started from caller: " + evt.getSource()));
    }
    @Override
    public void buildEnded(BuildEvent evt) {
        System.out.println(("Build ended for root object: " +
            evt.getResult().getRoot()));
    }
});
```

Processing the proper object

In the `buildStarted` event you should access `evt.getSource()` , which refers to the caller that

initiated the build (i.e. your Java class). However, in the `buildEnded` event it is better to access `evt.getResult().getRoot()`, which is the root object that was created from the build file.

The two are not necessarily the same (e.g. in order to create a `JPanel` from a YAML file your Java-side class does not have to extend `JPanel` at all, it is optional). This is useful in toolkits like SWT that do not allow you to extend particular component types.

Hot deployment of UI components

In order to further maximize developer productivity, all the JavaBuilders come with support for dynamically updating components while running the application. This means you can edit your YAML files and preview them in your app by just re-opening the panel/dialog being edited, without the need to restart the whole application.

In order to do this you need to pass the `"javabuilders.dev.src"` property to the Java VM on program startup and have it point to the relative path where your source code is vs. the compiled `.class` files.

In Eclipse, where the classes are in `"bin"` and the source code usually in `"src"` you need to pass this VM argument in your run configuration:

```
-Djavabuilders.dev.src=../src
```

That's it! Now the builder will read the YAML files from the source folder, instead of the bin folder, meaning you can keep editing them while the app is running and immediately see the changes as soon as you re-open the current component you were working on.

Swing Features

Now that we've seen the core JavaBuilders's features, let's explore what the Swing JavaBuilder provides on top of that for building actual Swing user interfaces.

Overview

Swing JavaBuilder

The Swing JavaBuilder is an instance of the JavaBuilders engine, pre-configured for use with the Swing UI toolkit. It is represented by the main class `org.javabuilders.swing.SwingJavaBuilder` and in most typical cases that is the only class you will be dealing with.

Java

```
public class MyFrame extends JFrame {  
    private BuildResult result = SwingJavaBuilder.build(this);  
    public MyFrame() {}  
}
```

The returned `BuildResult` obtain contains a reference to the various objects that were created during the build process, but it is often not necessary to interact with it at all (unless you are doing something more complex or custom).

General handling of component properties

In most cases there is a simple one-to-one mapping between the properties of Swing components and how they are set in the YAML file, e.g. a `JTextField.text` property in YAML is simply `JTextField(text=Some Text)`.

However, some components have been enhanced in the Swing JavaBuilder to make instantiating and using them even easier.

Actions and menus

Creating actions and menus for any application is one of the most cumbersome and time consuming tasks in Swing development. Fortunately enough, the Swing JavaBuilder delivers a whole slew of productivity enhancements in this area that makes creating menus a breeze.

Text, accelerators and mnemonics

Whether you are dealing with an `Action` or a `JMenuItem`, you can handle defining all these 3 properties in one simple text value, where the mnemonic is indicated via a "&" prefix and the accelerator is typed in manually after a "\t" tab indicator (similar to the way it is done in SWT), e.g.

```
JMenuItem(text="&Save\tCtrl+S")
```

The sample above sets the text to "Save", the mnemonic on the "S" character and the accelerator to "Ctrl+S".

Valid accelerators are:

1. Ctrl
2. Alt
3. Shift
4. Meta

followed by the appropriate character. They can be mixed together, e.g. "Ctrl+Alt+N". Due to the embedded "\t", such menu definitions have to be escaped into quoted text, as per the example above.

Actions

The regular Swing Action API has been modified separately to separate the concept of "name" vs "text" (which are the same in the Action API, but we treat them separately so that the text can be easily

internationalized, without affecting the name). It provides name, text, tooltipText, icon properties and the name of the Java method to be invoked is defined in the onAction handler.

YAML

```
Action(name=newAction, text=menu.file.new, icon=images/document-new.png, onAction=onFileNew)
```

Java

```
private void onFileNew() {
    System.out.print("onFileNew was invoked!");
}
```

Any descendant of AbstractButton (such as JMenuItem or JButton) can then refer to it in its action property, e.g.:

```
JFrame(title=frame.title, state=max, defaultCloseOperation=exitOnClose):
- Action(name=newAction, text=menu.file.new, tooltipText=menu.file.new.tooltip,
icon=images/document-new.png, onAction=onFileNew)
- Action(name=openAction, text=menu.file.open, tooltipText=menu.file.open.tooltip,
icon=images/document-open.png, onAction=onFileOpen)
- Action(name=saveAction, text=menu.file.save, tooltipText=menu.file.save.tooltip,
icon=images/document-save.png, onAction=onSave)
- Action(name=exitAction, text=menu.file.exit, icon=images/process-stop.png,
onAction={$confirm,exit})
- Action(name=option1Action, text=menu.option1, onAction=option1)
- Action(name=helpAboutAction, text=menu.help.about, onAction=onHelpAbout)
- JMenuBar:
- JMenu(name=fileMenu, text=menu.file):
- JMenuItem(action=newAction)
- JMenuItem(action=openAction)
- JSeparator()
- JMenuItem(action=saveAction)
- JSeparator()
- JMenuItem(action=exitAction)
- JMenu(name=optionsMenu, text=menu.options):
- JRadioButtonMenuItem(name=radio1Menu, action=option1Action)
- JRadioButtonMenuItem(name=radio2Menu, text=menu.option2)
- JRadioButtonMenuItem(name=radio3Menu, text=menu.option3)
- ButtonGroup: [radio1Menu, radio2Menu, radio3Menu]
- JSeparator()
- JCheckBoxMenuItem(text=menu.option1, onAction=option1)
- JCheckBoxMenuItem(text=menu.option2)
- JCheckBoxMenuItem(text=menu.option3)
- JMenu(name=helpMenu, text=menu.help):
- JMenuItem(action=helpAboutAction)
```

JMenuBar and JMenuItem

If you do not wish to use Actions, you can create menus by directly specifying the relevant properties on JMenuBar and JMenuItem instances:

```
JFrame(title=frame.title, iconImage=images/system-lock-screen.png):
- JMenuBar:
- JMenu(name=fileMenu, text=menu.file):
- JMenuItem(name=newMenu, text=menu.file.new, onAction=onFileNew)
- JMenuItem(name=openMenu, text=menu.file.open, onAction=onFileOpen)
- JSeparator()
- JMenuItem(name=exitMenu, text=menu.file.exit, onAction=exit)
- JMenu(name=optionsMenu, text=menu.options):
- JRadioButtonMenuItem(name=radio1Menu, text=menu.option1, onAction=option1)
- JRadioButtonMenuItem(name=radio2Menu, text=menu.option2)
- JRadioButtonMenuItem(name=radio3Menu, text=menu.option3)
- ButtonGroup: [radio1Menu, radio2Menu, radio3Menu]
- JSeparator()
- JCheckBoxMenuItem(text=menu.option1, onAction=option1)
- JCheckBoxMenuItem(text=menu.option2)
- JCheckBoxMenuItem(text=menu.option3)
- JMenu(name=helpMenu, text=menu.help):
- JMenuItem(name=helpAboutMenu, text=menu.help.about, onAction=onHelpAbout)
```

However, we recommend you always use Actions instead.

JPopupMenu

Popup menus can easily be added to any Swing component by simply specifying the "popupMenu" property to point to an existing JPopupMenu instance by name. The Swing JavaBuilder takes care of all the mouse event wiring to popup the menu upon right-click.

With actions

```
- Action(name=copyAction, text=menu.edit.copy, onAction=copy)
- Action(name=pasteAction, text=menu.edit.paste, onAction=paste)
- JPopupMenu(name=popup):
  - JMenuItem(action=copyAction)
  - JMenuItem(action=pasteAction)
- JTabbedPane(name=tabs, onChange=onTabChanged):
  - JPanel(name=frameYamlSource, tabTitle=tab.frameYamlSource):
    - JScrollPane(name=scroll1):
      JTextArea(name=frameSourceArea, popupMenu=popup)
```

Without actions

```
- JPopupMenu(name=popup):
  - JMenuItem(name=popupCopy, text=Copy, onAction=copy)
  - JMenuItem(name=popupPaste, text=Paste, onAction=paste)
- JTabbedPane(name=tabs, onChange=onTabChanged):
  - JPanel(name=frameYamlSource, tabTitle=tab.frameYamlSource):
    - JScrollPane(name=scroll1):
      JTextArea(name=frameSourceArea, popupMenu=popup)
```

Borders

Regular Borders

Any Swing component that allows setting of borders can do it by using a set of pre-defined constants:

- loweredBevel
- raisedBevel
- loweredEtched
- raisedEtched

```
JPanel(name=panel1, border=raisedBevel)
```

Color and Line borders

Borders can also be specified using just a line width or a Color / line width combination:

```
- JPanel(name=panel1, border=3)
- JPanel(name=panel1, border=olive 3)
- JPanel(name=panel1, border=ff00ee 3)
```

Titled Border

A titled border is a special case, since it has a text associated with it. In this case, there is a special property that will automatically create a TitledBorder and put the proper text in it, namely `groupTitle` :

```
JPanel(name=groupBox1, groupTitle=Customer Data):
  content:
    - JLabel(name=nameLabel, text="Customer name:")
    - JText(name=nameField)
```

Note:

`groupTitle` is internationalizable, so you can pass a resource key to it, instead of a hard-coded String.

Button Group

In order to create a ButtonGroup you just need to define it as a collection and pass it the names of buttons that define a group. This works for both regular radio buttons as well as radio button menu items.

Radio buttons

```
- JPanel(name=controls):
  - JRadioButton(name=rb1, text=Radio button 1)
  - JRadioButton(name=rb2, text=Radio button 2, selected=true)
  - ButtonGroup: [rb1, rb2]
```

Radio button menu items

```
- JMenu(name=optionsMenu, text=menu.options):  
  - JRadioButtonMenuItem(name=radio1Menu, text=menu.option1, onAction=option1)  
  - JRadioButtonMenuItem(name=radio2Menu, text=menu.option2)  
  - JRadioButtonMenuItem(name=radio3Menu, text=menu.option3)  
  - ButtonGroup: [radio1Menu, radio2Menu, radio3Menu]
```

Colors

Colors can be specified using a standard HTML/CSS style syntax. Valid values are:

Hex Color

```
JTextArea(name=textArea, background=ff00ee)
```

Short hex color (e.g. f0e gets interpreted as ff00ee)

```
JTextArea(name=textArea, background=f0e)
```

HTML color name (case-insensitive)

```
JTextArea(name=textArea, background=olive)
```

Note:

HTML color names: http://www.w3schools.com/html/html_colornames.asp

Fonts

Fonts can be specified using a CSS-like syntax: "bold|italic size name":

```
JButton(font=italic)  
JButton(font=italic bold)  
JButton(font=italic 14pt)  
JButton(font=Arial)  
JButton(font=italic bold 14pt Arial)
```

Icons and images

Any Swing API that expects an Icon or Image can be expressed by simply putting in the image path, relative to the caller class that initiated the build process.

YAML

```
JMenuItem(text=menu.save, icon=images/document-save.png)
```

Alternatively, if you initialized the builder with a `ResourceBundle` to activate internationalization, you can pass a resource key instead. The builder will look for the path to the image via the key in the bundle instead, e.g.

YAML

```
JMenuItem(text=menu.save, icon=images.saveDocument)
```

Properties file

```
images.saveDocument=/myapp/resources/images/document-save.png
```

JComboBox

Databinding

In order to bind a `List` to a `JComboBox`, you need to bind it to its `model` property, e.g.

```
bind:
- jComboBox.model: this.books
```

JFrame

`JFrame` support in the Swing JavaBuilder adds custom processing for the following properties:

size

Can be in "width_x_height" format (e.g. 800x400) or "packed" to indicate the `JFrame.pack()` method should be called at the end (after all the child components have been added), e.g.

```
JFrame(size=800x400)
JFrame(size=packed)
```

state

Allows setting the extended state of a frame, valid values are:

- `max`
- `maxh`
- `maxv`
- `icon`

```
JFrame(state=max)
```

JList

Databinding

In order to bind a `List` to a `JList`, you need to bind it to its `model` property, e.g.

```
bind:
- jList.model: this.books
```

JScrollPane

Used to wrap components in a scrollable pane. Since it only has one child underneath, it is entered not as a YAML list, but a single item:

```
JScrollPane(name=scrollPanel, verticalScrollBarPolicy=asNeeded,
horizontalScrollBarPolicy=asNeeded):
  JTextArea(name=textArea)
```

You can also use the shorter `vScrollBar` and `hScrollBar` aliases:

```
JScrollPane(name=scrollPanel, vScrollBar=asNeeded, hScrollBar=asNeeded):
  JTextArea(name=textArea)
```

Valid values for both properties are:

- `always`
- `asNeeded`
- `never`

JSplitPane

In order to use a JSplitPane just list the child components underneath it. The first two will be automatically added as the left/right (or top/bottom) panes, e.g.:

```
JPanel:
- JSplitPane(name=split1,orientation=verticalSplit):
  - JCustomPanel1(name=panel1)
  - JCustomPanel2(name=panel2)
```

The orientation property's verticalSplit or horizontalSplit values define the type of split.

JTabbedPane

In order to create tab pages, just list the controls you want as tabs underneath the JTabbedPane node. In order to specify the tab title, tooltip and icon use the following properties:

- tabTitle (localizable)
- tabToolTip (localizable)
- tabIcon
- tabEnabled

Those are used only if a component is listed underneath a JTabbedPane and are ignored if used anywhere else.

```
- JTabbedPane(name=tabs):
  - JPanel(tabTitle=tab.frameYamlSource, tabIcon=images/tab1.png):
```

JTable

Custom Table Models

You can integrate custom table models into your JTables. First, you must register your custom model (usually in the main()), so that the Swing JavaBuilder engine is aware of it, e.g.:

```
SwingJavaBuilder.getConfig().addType(MyCustomTableModel.class);
```

Then you can just refer to it directly:

```
JPanel:
- JScrollPane(name=scroll2):
  JTable(name=table1):
    - MyCustomTableModel(name=model)
```

Note: Your custom table does not actually need to have name property. If it does not exist, the Swing JavaBuilder will handle it as a virtual property. A named instance of the model (that you can manipulate from the Java code) will be created, e.g.:

```
private MyCustomTableModel model; //reference will be set during build process
```

Table Columns

JTable provides an easy way to create table columns, by just specifying a list of TableColumn objects underneath it, e.g.:

```
JPanel:
- JScrollPane(name=scroll2):
  JTable(name=table1):
    - TableColumn(name=col1,resizable=true, headerValue=Column 1)
    - TableColumn(name=col2,resizable=true, headerValue=Column 2)
    - TableColumn(name=col3,resizable=false, headerValue=Column 3)
```

When processing the list of table columns, the builder will evaluate the columns that are there already. If it can match based on the `identifier` or `headerValue` then it will use that existing columns, otherwise it will create a new one and add it to the `JTable`.

Cell Editor

Adding cell editors to a column is very easy. You can either define an explicit `TableCellEditor` implementation:

```
JTable(name=table1):  
  - TableColumn(name=col1,resizable=true, headerValue=Column 1):  
    - MyCustomCellEditor(name=colEditor)
```

or you can define an explicit `JCheckBox`, `JComboBox` or `JTextField` underneath it. In this case the builder will automatically wrap it with a `DefaultCellEditor` wrapper:

```
JTable(name=table1):  
  - TableColumn(name=col1,resizable=true, headerValue=Column 1):  
    - JComboBox(name=col1Box)  
  - TableColumn(name=col2,resizable=true, headerValue=Column 2):  
    - JCheckBox(name=col2Box)  
  - TableColumn(name=col3,resizable=false, headerValue=Column 3):  
    - JTextField(name=col3Field)
```

Cell Renderer

Similarly, you can define a `TableCellRenderer` underneath a column:

```
JTable(name=table1):  
  - TableColumn(name=col1,resizable=true, headerValue=Column 1):  
    - MyCustomRenderer(name=colRenderer)
```

If you want to define a column header renderer, just add a `forHeader=true` property:

```
JTable(name=table1):  
  - TableColumn(name=col1,resizable=true, headerValue=Column 1):  
    - MyCustomRenderer(name=colRenderer, forHeader=true)
```

Event handlers

Event handlers

Here's a complete list of event handlers by component type. Next to them is also the event-specific class that can get passed to your Java method if you require it (just remember to make it part of the method's signature), e.g.

```
private void someButtonClicked() {  
    //...one valid signature...  
}  
  
private void someButtonClicked(ActionEvent evt) {  
    //...another valid signature ...  
}  
  
private void someButtonClicked(JButton source) {  
    //...another valid signature ...  
}  
  
private void someButtonClicked(JButton source, Action evt) {  
    //...yet another valid signature ...  
}
```

Action

Name	Event Class
onAction	ActionEvent

(Abstract)Button

onAction	ActionEvent
----------	-------------

Component

onFocus	FocusEvent
onFocusLost	FocusEvent
onKeyPressed	KeyEvent
onKeyReleased	KeyEvent
onKeyTyped	KeyEvent
onMouseClicked	MouseEvent
onMouseDoubleClicked	MouseEvent
onMouseDragged	MouseEvent
onMouseEntered	MouseEvent
onMouseExited	MouseEvent
onMouseMoved	MouseEvent
onMousePressed	MouseEvent
onMouseReleased	MouseEvent
onMouseRightClicked	MouseEvent
onMouseWheelMoved	MouseEvent

JComboBox

onAction	ActionEvent
----------	-------------

JFrame

onStateChanged	WindowEvent
onWindowActivated	WindowEvent
onWindowClosed	WindowEvent
onWindowClosing	WindowEvent
onWindowDeactivated	WindowEvent
onWindowDeiconified	WindowEvent
onWindowFocus	WindowEvent
onWindowFocusLost	WindowEvent
onWindowIconified	WindowEvent
onWindowOpened	WindowEvent

JTabbedPane

onChange	ChangeEvent
----------	-------------

JTable

onSelection	ListSelectionEvent
-------------	--------------------

JTextField

onAction	ActionEvent
----------	-------------

JTree

onSelection	TreeSelectionEvent
-------------	--------------------

JTree

onSelection	TreeSelectionEvent
-------------	--------------------

Window

onStateChanged	WindowEvent
onWindowActivated	WindowEvent
onWindowClosed	WindowEvent
onWindowClosing	WindowEvent
onWindowDeactivated	WindowEvent
onWindowDeiconified	WindowEvent
onWindowFocus	WindowEvent
onWindowFocusLost	WindowEvent
onWindowIconified	WindowEvent
onWindowOpened	WindowEvent

Swing Layout Management

Layout management is one of the biggest pain points in any UI development. The Swing JavaBuilder solves it by using a simple DSL (Domain Specific Language) that runs on top of the brilliant MigLayout layout manager.

After using MigLayout you will never go back to any other JDK layout manager, it makes them all obsolete.

MigLayout DSL

What is MigLayout?

MigLayout is a revolutionary layout manager for Swing and SWT, written by Mikael Grev and released under the open source BSD license. It revolutionizes layout management by making it much more dynamic and thus greatly reducing the number of lines of code one has to write, even for very complex layouts.

MigLayout is available for download from <http://miglayout.com>. We recommend to read the Cheat Sheet and introduction to MigLayout available on that website. Once you try MigLayout you can never go back to archaic layout managers such as GridBagLayout or GroupLayout. It even makes the formidable JGoodies Forms layout manager obsolete.

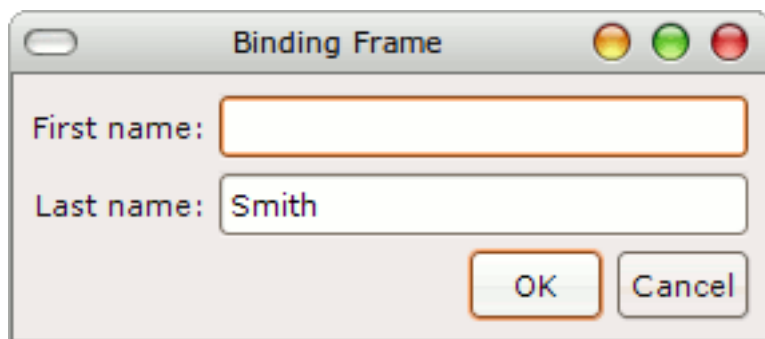
Visual MigLayout DSL syntax (a GUI builder in pure text)

The visual layout DSL is basically a way to have a GUI builder, but in a pure text format. In short, it allows you to define controls' layout in a text file (by using their names) and from their relative alignments and number of rows, the builder will attempt to automatically figure out how many rows/columns there are, which control goes into which cell, whether it should be left/top/right/center aligned, how many cells should it span, etc.

The layout DSL translates the constraints into standard MigLayout constraints, hence this is basically nothing more than a visual text-based interface to the full power of MigLayout.

Quick Example

Let's say we need to create a simple dialog with 3 rows: a table/text field in the first two rows, and OK/Cancel buttons (right aligned) in the last row.



```
MigLayout: |
[[insets 8]
 [pref]      [grow]
>firstNameLabel  firstName
>lastNameLabel  lastName
>okButton+*,cancelButton    [growy, bottom]
{okButton: tag OK, cancelButton: tag Cancel}
```

From this you can probably see right away that we have 3 rows (as in 3 lines of text), the labels are in

the same vertical column, the text fields are in the same vertical column (which is flagged to "grow", a standard MigLayout constraint).

General format

```
MigLayout: |
  [{global layout constraints}]                #optional
  [column constraints] [another column's constraints] #optional
  control1 control2 [row constraint - optional]
  control3 control4
  {control1: specific MigLayout constraint (e.g. baseline) } #optional
```

Alignment

Goes before the control name, e.g. ">fieldNameLabel". If none are presents it defaults to "top, left".

<	horizontally left aligned (can be omitted, it is the default value)
<	horizontally left aligned (can be omitted, it is the default value)
>	horizontally right aligned
	horizontally centered
-	vertically centered
/	vertically bottom aligned
^	vertically top aligned (usually omitted, use only when needed to override the default, e.g. "baseline" in MigLayout)

Cell Spanning

In "+X+Y" format (X= horizontal cells to span, Y= vertical cells to span), e.g. "okButton+2".

Examples:

- +* : horizontally span rest of row (e.g. "okButton+*")
- +2 : horizontally span 2 cells
- +2+4: horizontally span 2 cells, vertically span 4 cells
- +2+*: horizontally span 2 cells, vertically span the the rest of the column

Cell Splitting

In "controlName1,controlName2" format (i.e. control names separated with a comma). Used when you want to place multiple controls into the same cell. All the general cell constraints (alignment, spanning, etc.) are applied to the first control, e.g. "okButton+*,cancelButton".

Size Groups

In "=X" format (X = size group number), e.g. "okButton=1 cancelButton=1"

Allows to specify which controls should have the same preferred size. Useful especially when you want different command buttons to have the same size (e.g. OK and Cancel). There is also support for horizontal and vertical size groups (i.e. those that apply only to common width and/or height, instead of both), but it's not quite there yet. It is defined by appending an 'x' or 'y' after the size group, e.g.:

```
okButton=1x cancelButton=1x`
```

String Literal Controls

In order to further simplify the creation of user interfaces, the DSL allows you to enter string literals (embedded in double quotes) instead of control names. Such entries will automatically be interpreted as

labels (e.g. JLabel for Swing, Label for SWT, etc.) and an underlying control will be created without the need to manually specify it in the YAML.

Using string literal controls

```
JFrame(name=frame, title=frame.title, size=packed, defaultCloseOperation=exitOnClose):  
- JTextField(name=fName)  
- JTextField(name=lName)  
- JTextField(name=email)  
- JButton(name=save, text=button.save, onAction=($validate,save,done))  
- JButton(name=cancel, text=button.cancel, onAction=($confirm,cancel))  
- MigLayout: |  
  [pref] | [grow,100] [pref] | [grow,100]  
  "label.firstName" fName "label.lastName" lName  
  "label.email" email+*  
  >save+=1,cancel=1
```

or the more verbose, traditional way:

Using explicit label definitions

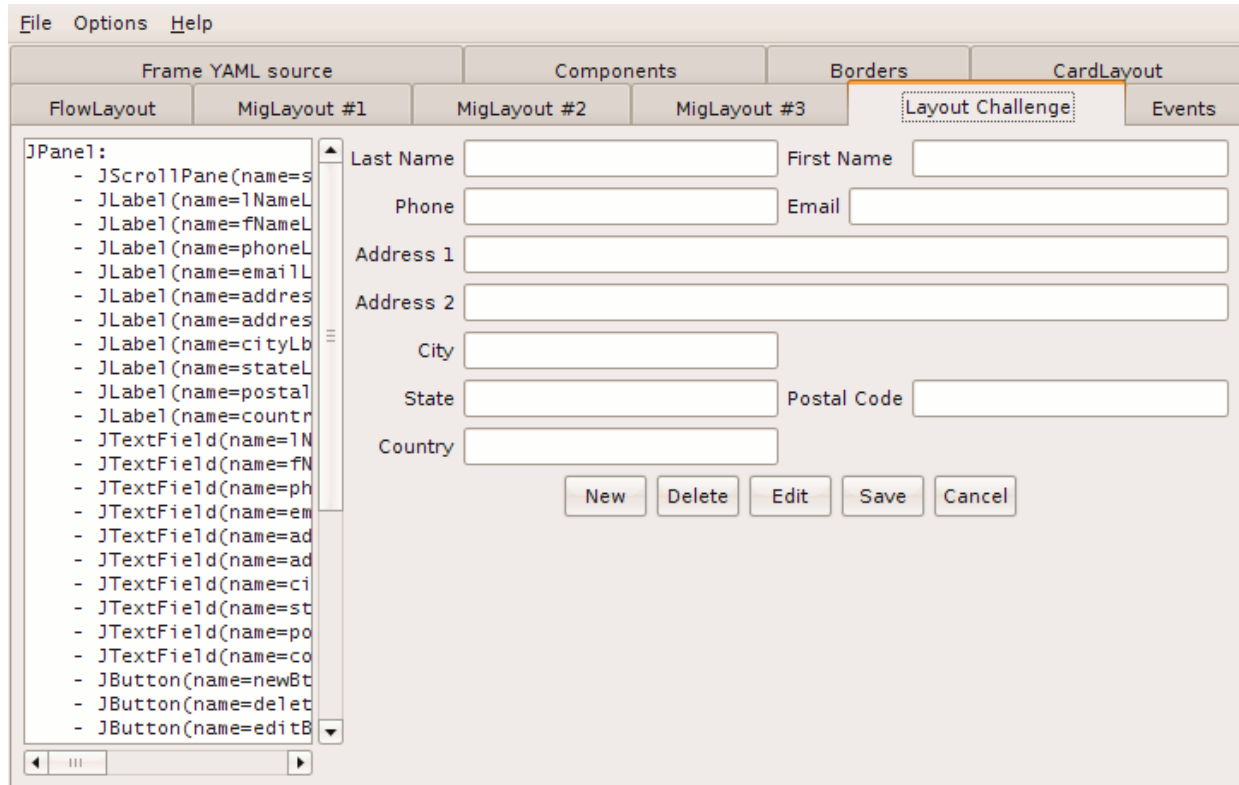
```
JFrame(name=frame, title=frame.title, size=packed, defaultCloseOperation=exitOnClose):  
- JLabel(name=fNameLbl, text=label.firstName)  
- JLabel(name=lNameLbl, text=label.lastName)  
- JLabel(name=emailLbl, text=label.email)  
- JTextField(name=fName)  
- JTextField(name=lName)  
- JTextField(name=email)  
- JButton(name=save, text=button.save, onAction=($validate,save,done))  
- JButton(name=cancel, text=button.cancel, onAction=($confirm,cancel))  
- MigLayout: |  
  [pref] | [grow,100] [pref] | [grow,100]  
  fNameLbl fName lNameLbl lName  
  emailLbl email+*  
  >save+=1,cancel=1
```

Complex Example

From John O'Connors Layout Manager Challenge:

http://weblogs.java.net/blog/joconner/archive/2006/10/layout_manager.html

The screenshot shows a Java Swing application titled "Address Book Demo". On the left is a scrollable list of names: Bunny, Bugs; Cat, Sylvester; Coyote, Wile E.; Devil, Tasmanian; Duck, Daffy; Fudd, Elmer; Le Pew, Pepé; and Martian, Marvin (which is highlighted in blue). To the right of the list is a form for editing an address book entry. The form contains the following fields: Last Name (Martian), First Name (Marvin), Phone (805-123-4567), Email (marvin@wb.com), Address 1 (1001001010101 Martian Way), Address 2 (Suite 10111011), City (Ventura), State (CA), Postal Code (93001), and Country (empty). At the bottom of the form are five buttons: New, Delete, Edit, Save, and Cancel. The "Delete" button is currently highlighted with a blue border.



```

JPanel:
- JScrollPane(name=scroll1): JTextArea(name=source,font=Monospaced,editable=false)
- JTextField(name=lName)
- JTextField(name=fName)
- JTextField(name=phone)
- JTextField(name=email)
- JTextField(name=address1)
- JTextField(name=address2)
- JTextField(name=city)
- JTextField(name=state)
- JTextField(name=postal)
- JTextField(name=country)
- JButton(name=newBtn, text=New)
- JButton(name=deleteBtn, text=Delete)
- JButton(name=editBtn, text=Edit)
- JButton(name=saveBtn, text=Save)
- JButton(name=cancelBtn, text=Cancel)
- MigLayout:
  [200,grow] [right] [200,grow] [200,grow]
  scroll1+1+* "Last name:" lName "First Name" fName
  "Phone:" phone "Email:" +2,email
  "Address 1:" address1+*
  "Address 2:" address2+*
  "City:" city
  "State:" state "Postal Code:" postal
  "Country:" country
  ^|newBtn+*=1,^deleteBtn=1,^editBtn=1,^saveBtn=1,^cancelBtn=1 [grow]

```

MigLayout

If for whatever reason you do not want to use the MigLayout DSL, you can still use regular MigLayout properties and syntax, e.g.:

```

JFrame(title=My Frame):
  content:
    - JLabel(name=firstNameLabel,text=First Name)
    - JTextField(name=firstName)
    - JLabel(name=lastNameLabel,text=Last Name)
    - JTextField(name=lastName)
    - JButton(name=okButton)
    - MigLayout:
      layoutConstraints: wrap 2 #general layout constraints
      columnConstraints: [] [grow] [] #general column constraints
      rowConstraints: [] [] [] #general row constraints
      constraints:

```



```
- firstNameLabel: right
- firstName: 200px, sg 1
- lastNameLabel: right
- lastName: 200px, sg 1
- okButton: span, right, tag ok
```

But we recommend you always use the DSL syntax instead, it's much more powerful and easier to use after the initial learning curve.

CardLayout

CardLayout support is provided by adding a CardLayout node at the end of the list of child components, e.g.

```
JPanel:
- JPanel(name=panel1)
- JPanel(name=panel2)
- CardLayout(name=cards): [panel1,panel2]
```

By default the card name is the same as the name of the control that was added as a card. Using the "name" property you can get a handle to the created instance of CardLayout in your Java-side code, e.g.

```
private CardLayout cards;
```

FlowLayout

In order to use FlowLayout, just create a FlowLayout node at the end of the list of child components. No need to specify which ones to add, they all get added automatically, e.g.:

```
JPanel:
- JPanel(name=panel1,groupTitle=Flow layout components):
  - JLabel(text=Label 1)
  - JButton(text=Button 1)
  - JLabel(text=Label 2)
  - JButton(text=Button 2)
  - JLabel(text=Label 3)
  - JButton(text=Button 4)
  - JLabel(text=Label 5)
  - JButton(text=Button 5)
  - FlowLayout(alignment=left,hgap=30,vgap=30,alignOnBaseline=true)
```

Extending the JavaBuilders engine

Overview

The core JavaBuilders engine is domain-agnostic, i.e. there is no logic in it specific to any particular toolkit such as Swing or SWT. Each of domain-specific builders (such as the Swing JavaBuilder or the SWT JavaBuilder) are just thin proxies for the common `Builder` APIs which pass along a pre-configured instance of a `BuilderConfig` object, which contains all the component types and custom handlers for each UI toolkit.

This builder configuration object is usually exposed via the static `getConfig()` method on the builder, e.g. `SwingJavaBuilder.getConfig()`.

By manipulating its properties you can change the default configuration, register new object types, customized handlers for particular controls or particular properties of a control.

Registering new component types

All you need to do is call the `BuilderConfig.addType(Class clazz)` method, presumably from your application's `main()`:

```
SwingJavaBuilder.getConfig().addType(MyCustomComponent.class);
```

and then you can start referring to it directly in the YAML file:

```
MyCustomClass(property1=value1,property2=value2, etc...)
```

You can also add it with a specific alias to avoid name collision (by default it takes the simple class name):

```
SwingJavaBuilder.getConfig().addType("CustomClassAlias",MyCustomClass.class);
```

```
CustomClassAlias(property1=value1,property2=value2, etc...)
```

Customizing object creation : `ITypeHandler`

If you need to write your own custom creation code for a class instance (e.g. for a control that has a constructor that expects parameters during initialization), you need to implement an instance of `ITypeHandler`, usually by extending `AbstractTypeHandler`. It needs to be then registered for the class-specific `TypeDefinition` object within the `BuilderConfig` instance.

Customizing initialization logic: `ITypeHandlerAfterCreationProcessor`

If your object does not need special constructor logic, but just some post-creation initialization, then you need to implement the simple `IAfterCreationProcessor` interface and register it with your type's `TypeDefinition`. It's logic will be executed after the object is created, but before any children get processed.

Customizing post-processing of children nodes; `ITypeHandlerFinishProcessor`

If you need to inject some logic after a parent's child nodes have been all processed, you need to implement the `ITypeHandlerFinishProcessor` interface and add it to the appropriate `TypeDefinition` object.

Appendix

Layout Samples

Layout Samples