Table of Contents

Version: 1.0 / 2019-07-01

Introduction

This collection of various tutorials is aimed at providing you with a broad overview over the concepts and mechanics of the JVx application framework.

Of Technologies and Factories

Let’s talk about the UI layer, the implementations, and the factory that powers it all.

The Basics

For everyone who does not know, JVx allows you to write code once and run it on different GUI frameworks without changing your code. This is achieved by hiding the concrete GUI implementations behind our own classes, the UI classes, and providing “bindings” for different GUI frameworks behind the scenes. Such a “single sourcing” approach has many advantages, one of which is that migrating to a new GUI framework requires only the change of a single line: the one that controls which factory is used.

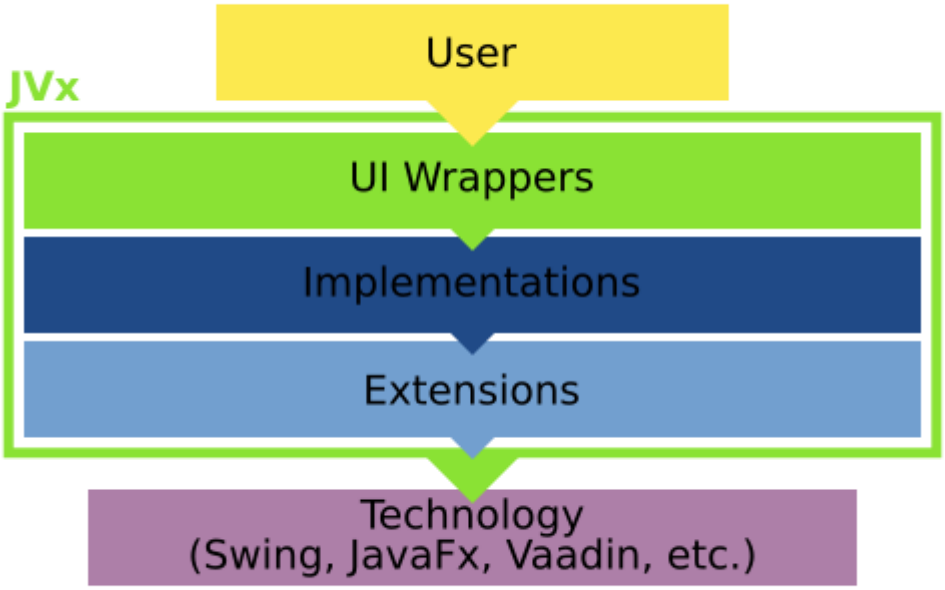
The Patterns

The factory pattern is an important pattern in object-oriented programming It empowers us to delegate the creation of objects to another object that is not known at design and/or compile time. That allows us to use objects which have not been created by us but merely “provided” to us by an unknown-to-us source.

The bridge pattern, on the other hand, describes a technique which wraps implementations in another implementation and forwards all or most functionality to that wrapped implementation. This allows us to mix and match functionality without the need to have it in all implementations at once.

Like an Onion

JVx is separated into different layers with the UI layer being at the top and of the most concern to users.



Technology

Obviously, the first one in the chain is the so-called “technology” layer. It represents the UI technology – for example Swing, JavaFX or Vaadin – that is used to power the JVx application.

To put it more simply:

```
public class JButton {}
```

Extension

Next comes the extension layer. Components from the technology are extended to support needed features of JVx. This includes creating bindings for the databook, additional style options, and changing of behavior, if necessary. From time to time, this also includes creating components from scratch if the provided ones do not meet the needs, or there simply are none with the required functionality. For the most part, we do our best that these layers can be used without JVx, meaning that they represent a solitary extension to the technology. A very good example is our JavaFX implementation, which compiles into two separate jars, the first being the complete JVx/JavaFX stack, the second being stand-alone JavaFX extensions that can be used in any application and without JVx.

Theoretically, one can skip this layer and directly jump to the implementation layer, but, so far, it has proven necessary (for cleanliness of the code, object structure, and sanity reasons) to create a separate extension layer.

```
public class JExtendedButton extends JButton {}
```

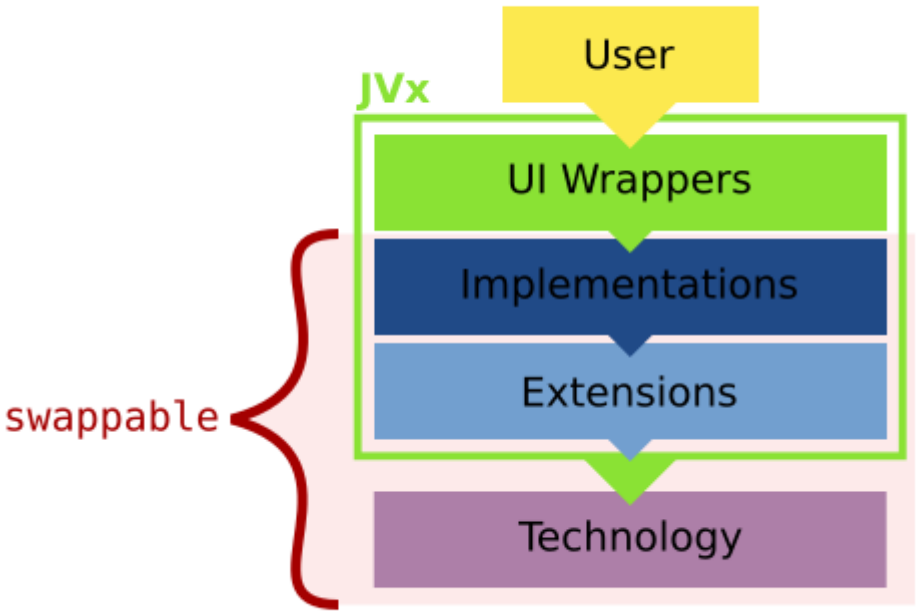
Implementation

After that comes the implementation layer. These implementations of the JVx interfaces are the actual objects returned by the factory. This is some sort of “glue” layer: it binds the technology or extended components against the interfaces which are provided by JVx.

```
public class SwingButton implements IButton {}
```

UI

Last, but definitely not least, is the UI layer, which wraps the implementations. It is completely implementation-independent, which means that one can swap out the stack underneath:



This is achieved because the UI layer is not extending the implementation layer but wrapping instances provided by the factory. It is oblivious to what technology is actually underneath it.

```
public class UIButton implements IButton {}

SwingButton resource = SwingFactory.createButton()
```

Why is the UI Layer Necessary?

It isn’t, not at all. The implementations could be used directly without any problems, but having yet another layer has two key benefits:

- It allows easier usage.

- It allows to add technology-independent features.

By wrapping it one more time, we gain a lot of freedom which we would not have otherwise when it comes to features and coding. The user does not need to call the factory directly and, instead, just needs to create a new object:

```
IButton button = new UIButton();
```

Internally, of course, the factory is called and an implementation instance is created, but that is an implementation detail. If we would use the implementation layer directly, our code would need to know about the implementations, which doesn't follow the single-sourcing principle:

```
IButton button = new SwingButton();
```

It also would be possible to directly use the factory, but that makes it quite tedious to type:

```
IButton button = UIFactoryManager.getFactory().createButton();
```

Both can be avoided by using another layer that the factory calls for us:

```
public class UIButton implements IButton
{
    private IButton resource;

    public UIButton()
    {
        resource = UIFactoryManager.getFactory().createButton();
    }

    public void someInterfaceMethod()
    {
        resource.someInterfaceMethod();
    }
}
```

Additionally, this layer allows us to implement features that can be technology-independent. Our naming scheme, which we created during stress testing of a Vaadin application, is a very good example of that. The names of the components are derived in the UI layer without any knowledge of the underlying technology or implementation.

Also, it provides us (and everyone else, of course) with a layer which allows to rapidly and easily build compound components out of already existing ones, like this:

```
public class LabeledButton extends UIPanel
{
    private IButton button = null;
    private ILabel label = null;

    public LabeledButton ()
    {
        super();
```

```
        initializeUI();
    }

    private void initializeUI()
    {
        button = new UIButton();
        label = new UILabel();

        setLayout(new UIBorderLayout());
        add(label, UIBorderLayout.LEFT);
        add(button, UIBorderLayout.CENTER);
    }
}
```

Of course, that is not even close to sophisticated, or even a good example for that matter. However, it shows that one can build new components out of already existing ones without having to deal with the technology or implementation at all, creating truly cross-technology controls.

The Factory

The heart piece of the UI layer is the factory that is creating the implemented classes. It's a rather simple system, a singleton which is set to the technology-specific implementation and can be retrieved later:

```
// At the start of the application.
UIFactoryManager.setFactoryInstance(new SwingFactory());
// Or alternatively:
UIFactory.getFactoryInstance(SwingFactory.class());

// Later inside the UI wrappers.
IButton button = UIFactory.getFactory().createButton();
```

The complexity of the implementation of the factory is technology dependent, but for the most part it is devoid of any "interesting" logic:

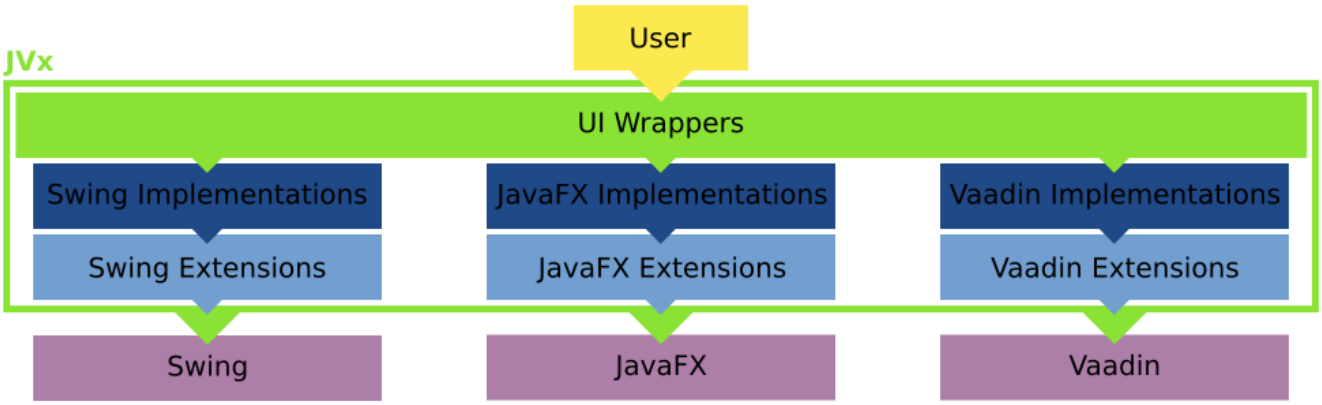
```
public class SwingFactory implements IFactory
{
    @Override
    public IButton createButton()
    {
        SwingButton button = new SwingButton();
        button.setFactory(this);

        return button;
    }
}
```

It "just returns new objects" from the implementation layer. That's about it when it comes to the

factory, it is as simple as that.

Piecing It Together

With all this in mind, we know now that JVx has swappable implementations underneath its UI layer for each technology it utilizes:



Changing between them can be as easy as setting a different factory. I say “can” because that is only true for Swing, JavaFX, and similar technologies. Vaadin, obviously, requires some more setup work. Theoretically, one could embed a complete application server and launch it when the factory for Vaadin is created, allowing the application to be basically stand alone and started as easily as a Swing application. That is possible.

What Else?

That is how JVx works in regards to the UI layer. It depends on “technology-specific stacks”, which can be swapped out and implemented for pretty much every GUI framework out there. We currently provide support for Swing, JavaFX, and Vaadin, but we also had implementations for GWT and Qt. Additionally, we support a “headless” implementation, which uses lightweight objects that can be serialized and send over the wire without much effort.

Adding a New Technology

Adding support for a new technology is as straightforward as one can imagine: simply create the extensions/implementations layers and implement the factory for that technology. Giving a complete manual would be out for scope for this document, but the most simple approach to adding a new stack to JVx is to start with stubbing out the `IFactory` and implementing `IWindow`. Once that one window shows up, it’s just implementing one interface after another in a quite straightforward manner. In the end, your application can switch to yet another GUI framework without the need to change your code.

Conclusion

Even though the stack of JVx is more complicated compared with other GUI or application frameworks, this complexity is set off by the benefits it brings. One can change the used GUI technology without much effort and, most importantly, without touching the application logic at all.

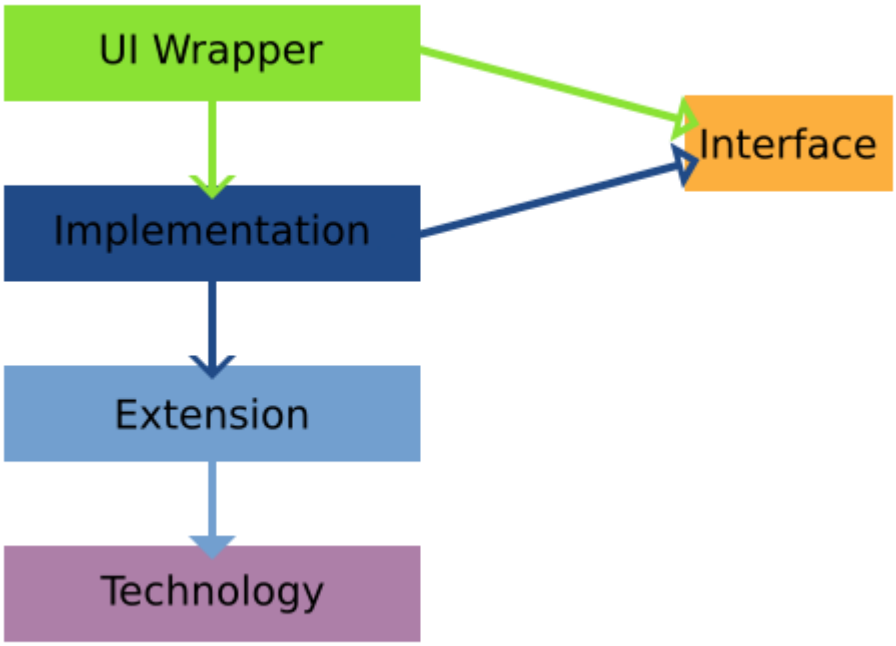
Resource and UI Resource

Let’s talk about resources and UI resources and why they sound similar yet are not the same.

The Basics

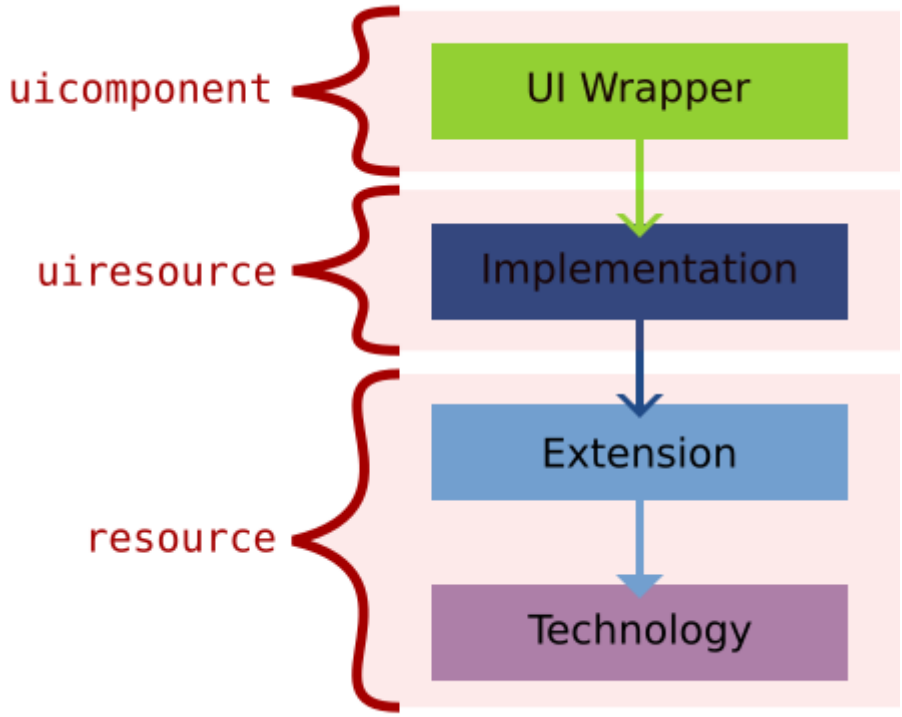
We’ve encapsulated by a wrapper class. A “UI resource”, on the other hand, is an encapsulated concrete implementation of one of the interfaces on the UI layer.

Let’s do a short overview of how the JVx architecture looks like in regards to the GUI stack:



The UI wrappers are the main UI classes that are used to create the GUI (e.g., `UIButton`). These are wrapping the implementations (e.g., `SwingButton`), which themselves are wrapping the extension/technology (e.g., a `JVxButton`/`JButton`). Only the UI and implementation classes implementing the interface are required for the component (e.g., `IButton`). That also means that the implementation is dependent on the extension/technology component, but the UI can use any object which implements the interface.

Now, with that knowledge, we can start defining what is what:



The resource itself, accessed by calling `<uiwrapper>.getResource()`, is the extension/technology component. The UI resource can be accessed by calling `<uiwrapper>.getUIResource()`. The UI component can be accessed by calling `<uiwrapper>.getUIComponent()` and is usually the UI wrapper class itself. If we use our previous Swing example, the resource would be a `JVxButton/JButton`, the UI resource would be the `SwingButton` and the UI component would be the `UIButton`.

As one can see, access to all objects which comprise GUI possible at all times. We, of course, have the UI component, we can access the implementation component, and we can access the extension/technology component. Theoretically, we could also swap them at runtime, but in JVx, this is limited to the construction of the object to greatly reduce the potential for error and complexity of the framework code.

Creating Custom Components

We will use an example from the part about creating custom components, which we will come to later. The `BeepComponent` is a simple `UIComponent` extension that contains a label and two buttons inside itself.

```
public class BeepComponent extends UIComponent<IPanel>
{
   public BeepComponent()
   {
      super(new UIPanel());

      UIButton highBeepButton = new UIButton("High Beep");
```

```
      highBeepButton.eventAction().addListener(Beeper::playHighBeep);

      UIButton lowBeepButton = new UIButton("Low Beep");
      highBeepButton.eventAction().addListener(Beeper::playLowBeep);

      UIFormLayout layout = new UIFormLayout();

      uiResource.setLayout(layout);
      uiResource.add(new UILabel("Beep"), layout.getConstraints(0, 0, -1,
0));
      uiResource.add(highBeepButton, layout.getConstraints(0, 1));
      uiResource.add(lowBeepButton, layout.getConstraints(1, 1));
   }
}
```

We are setting a new UI resource (a `UIPanel`) in the constructor (at line #5), which is to be used by the `UI component`. In this case, it is not an implementation, but another UI component. However, that doesn't matter because the UI resource must only implement the expected interface. At line #15 we start using that custom UI resource.

Because UI component is an abstract component designed for exactly this usage, the example might not be the most exciting one, but it clearly illustrates the mechanics.

Bolting on Functionality

Also, from the part about creating custom components, we can reuse the `PostfixedLabel` as example:

```
private UILabel testLabel = new UILabel()
{
    public UILabel()
    {
        super(new PostfixedLabel("", "-trial"));
    }
};
```

Now `testLabel` will be using the `PostfixedLabel` internally but with no indication to the user of the object that this is the case. This allows us to extend the functionality of a component completely transparently, especially in combination with functions that return a `UI component` and similar.

An Important Note About the Component Hierarchy

If we create a simple component extensions, like the `BeepComponent` above, it is important to note that there is one other layer of indirection in regards to the hierarchy on the technology layer. If we create a simple frame with the `BeepComponent` in it, one might expect the following hierarchy:

```
         UI                    Technology
```

```
---------------------    ----------------------
 UIFrame                  Frame
   \-UIPanel                \-Panel
       \-BeepComponent          \-BeepComponent
                                    \-Panel
                                        |-Label
                                        |-Button
                                        \-Button
```

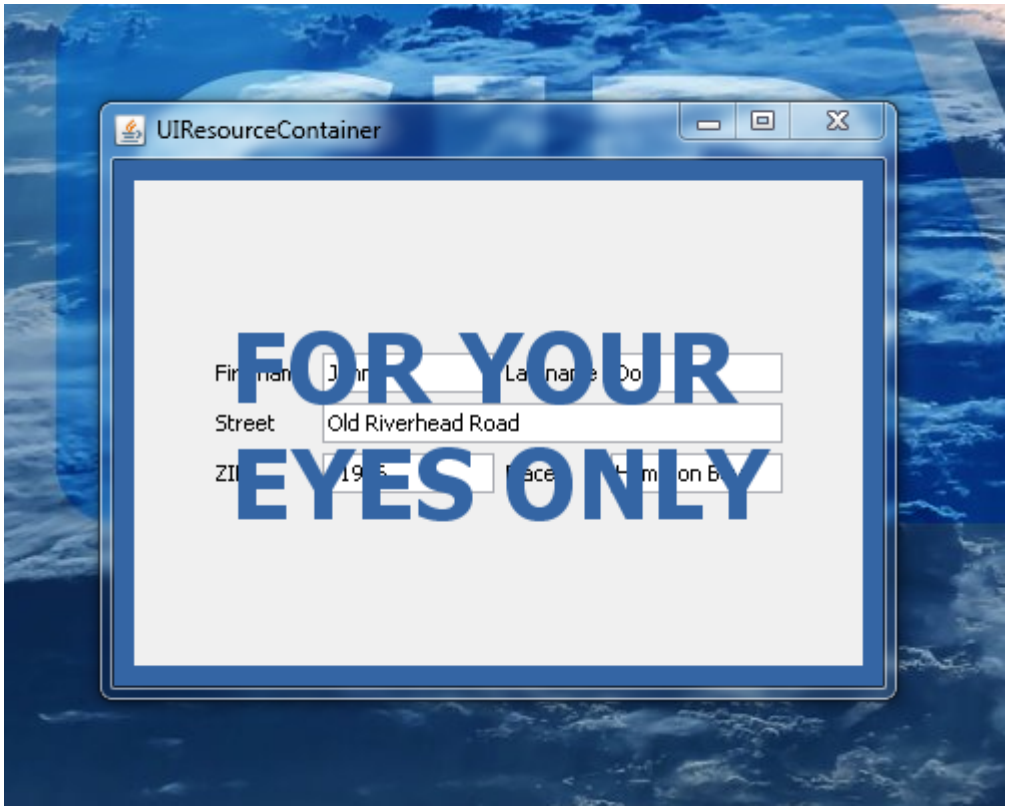
with the BeepComponent added and its subcomponents as its children. However, the actual hierarchy looks like this:

```
         UI                    Technology
---------------------    ----------------------
 UIFrame                  Frame
   \-UIPanel                \-Panel
       \-BeepComponent          \-Panel
                                    |-Label
                                    |-Button
                                    \-Button
```

That is because such extended components are not “passed” to the technology; they only exist on the UI layer because they do not have a technology component which could be used. That is done by adding the `UI component` to the UI parent, but for adding the actual technology component, the set UI resource is used.

The Special Case of Containers

Another special case is containers. For example, we could create a panel that displays an overlay in certain situations, and we will need to use that throughout the whole application.



That means we do not want to build it every time anew, so one option would be to use a factory method to “wrap” the content. Something like this:

```
UIFormLayout panelLayout = new UIFormLayout();
panelLayout.setHorizontalAlignment(UIFormLayout.ALIGN_CENTER);
panelLayout.setVerticalAlignment(UIFormLayout.ALIGN_CENTER);

UIPanel panel = new UIPanel();
panel.setLayout(panelLayout);
panel.add(new UILabel("Firstname"), panelLayout.getConstraints(0, 0));
panel.add(new UITextField("John"), panelLayout.getConstraints(1, 0));
panel.add(new UILabel("Lastname"), panelLayout.getConstraints(2, 0));
panel.add(new UITextField("Doe"), panelLayout.getConstraints(3, 0));
panel.add(new UILabel("Street"), panelLayout.getConstraints(0, 1));
panel.add(new UITextField("Old R. Road"), panelLayout.getConstraints(1, 1,
3, 1));
panel.add(new UILabel("ZIP"), panelLayout.getConstraints(0, 2));
panel.add(new UITextField("11946"), panelLayout.getConstraints(1, 2));
panel.add(new UILabel("Place"), panelLayout.getConstraints(2, 2));
panel.add(new UITextField("Hampton Bays"), panelLayout.getConstraints(3,
2));

parentContainer.add(OverlayPanelFactory.wrap(panel), UIBorderLayout.CENTER);
```

And the wrap method itself:

```
public static final UIPanel wrap(IComponent pContent)
{
    UILabel overlayLabel = new UILabel("FOR YOUR<br>EYES ONLY");
```

```
    overlayLabel.setBackground(null);
    overlayLabel.setFont(UIFont.getDefaultFont().deriveFont(UIFont.BOLD,
48));
    overlayLabel.setForeground(UIColor.createColor("#3465a4"));
    overlayLabel.setHorizontalAlignment(UILabel.ALIGN_CENTER);

    UIFormLayout layout = new UIFormLayout();

    UIPanel panel = new UIPanel();

    panel.setLayout(layout);
    panel.setBackground(UIColor.createColor("#3465a4"));
    panel.add(overlayLabel, layout.getConstraints(0, 0, -1, -1));
    panel.add(pContent, layout.getConstraints(0, 0, -1, -1));

    return panel;
}
```

This is easy enough, but let's say we'd like to add logic to that wrapper. At that point, it becomes more complicated. We can't use the same technique as the custom component from above because, in that case, the "overlaying panel" would simply not be displayed. However, there is a similar mechanism for containers: setting the UI resource container.

The UI resource container is another special mechanism that works similar to setting the UI resource, but it works the other way round. While setting the UI resource "hides" components from the technology in UI layer, setting the UI resource container hides components from the UI layer while they are added in the technology. As it is a little complicated, here is our example using this technique again:

```
public static class OverlayedPanel extends UIPanel
{
    public OverlayedPanel()
    {
        super();

        UILabel overlayLabel = new UILabel("FOR YOUR<br>EYES ONLY");
        overlayLabel.setBackground(null);
        overlayLabel.setFont(UIFont.getDefaultFont().deriveFont(UIFont.BOLD,
48));
        overlayLabel.setForeground(UIColor.createColor("#3465a4"));
        overlayLabel.setHorizontalAlignment(UILabel.ALIGN_CENTER);

        UIPanel innerPanel = new UIPanel();

        UIFormLayout layout = new UIFormLayout();

        setLayout(layout);
        setBackground(UIColor.createColor("#3465a4"));
        add(overlayLabel, layout.getConstraints(0, 0, -1, -1));
        add(innerPanel, layout.getConstraints(0, 0, -1, -1));
```

```
        setUIResourceContainer(innerPanel);
    }
}
```

What we've done is extended a UI panel (line #1), setting it up and adding children, and then we've declared one of its children as the UI resource container (line #22). So all methods that are specific to UI container (adding children, setting a layout, etc.) are now forwarding to the inner panel and manipulating the contents of the overlaid panel directly.

And here is how it is used:

```
UIFormLayout panelLayout = new UIFormLayout();
panelLayout.setHorizontalAlignment(UIFormLayout.ALIGN_CENTER);
panelLayout.setVerticalAlignment(UIFormLayout.ALIGN_CENTER);

UIPanel panel = new OverlayedPanel();
panel.setLayout(panelLayout);
panel.add(new UILabel("Firstname"), panelLayout.getConstraints(0, 0));
panel.add(new UITextField("John"), panelLayout.getConstraints(1, 0));
panel.add(new UILabel("Lastname"), panelLayout.getConstraints(2, 0));
panel.add(new UITextField("Doe"), panelLayout.getConstraints(3, 0));
panel.add(new UILabel("Street"), panelLayout.getConstraints(0, 1));
panel.add(new UITextField("Old R. Road"), panelLayout.getConstraints(1, 1,
3, 1));
panel.add(new UILabel("ZIP"), panelLayout.getConstraints(0, 2));
panel.add(new UITextField("11946"), panelLayout.getConstraints(1, 2));
panel.add(new UILabel("Place"), panelLayout.getConstraints(2, 2));
panel.add(new UITextField("Hampton Bays"), panelLayout.getConstraints(3,
2));

parentContainer.add(panel, UIBorderLayout.CENTER);
```

Notice that we can use it as any other panel (line #5) and simply add it to the parent (line #18). For a user of the API, it is transparent as to whether there are more components or not. This is also visible in the created component hierarchy:

```
         UI                    Technology
---------------------    ----------------------
 UIPanel                  Panel
   \-OverlayedPanel         \-Panel
       |-UILabel                |-Label
       |-UITextField                \-Panel
       |-UILabel                    |-Label
       |-UITextField                |-TextField
       |-UILabel                    |-Label
       |-UITextField                |-TextField
       |-UILabel                    |-Label
       |-UITextField                |-TextField
       |-UILabel                    |-Label
       \-UITextField                |-TextField
                                    |-Label
```

```
\-TextField
```

This makes it very easy to have containers which add additional components without the actual GUI noticing or caring.

Conclusion

Because of the way the JVx framework is designed, it is easy to access all layers of the GUI framework and facilitate the usage of these layers to create custom components and allow easy access to the wrapped components, no matter on what layer or of what kind they are.

Launchers and Applications

Let’s talk about launchers, and how they are used to start JVx applications.

Starting an Application

From a technical point of view, there are two prerequisites which must be fulfilled before a JVx application can run:

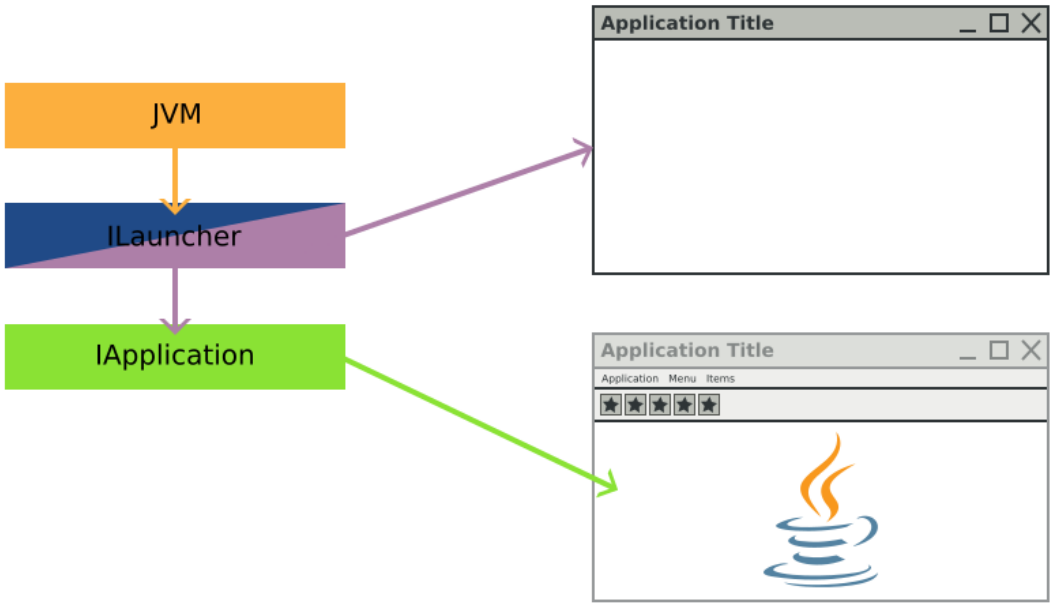
- The JVM must have started.
- The technology specific system must have started.

Then, and only then, the JVx application can run. Depending on the implementation that is used, that can be as easy as instancing the factory (Swing, JavaFX), but it can also mean that a servlet server has to start (Vaadin). Because we do not wish to encumber our applications with technology-specific code, we have to entrust all this to an encapsulated entity, meaning the implementations of `ILauncher` and `IApplication`.

Following the Chain

The steps for getting an application to start are as follows:

- The first thing that must run is, obviously, the JVM. Without it, we won’t have much luck starting anything!
- The launcher must be created, and it must start the technology.
- The launcher then creates the application, which the user is seeing.



So we need two classes, the `ILauncher` implementation that knows how to start the technology and the `IApplication` implementation. That we already knew, so let's try to put this into code. For simplicity reasons (and because I don't want to write a complete factory from scratch for this example), we will reuse the Swing implementation and write a new launcher and application for it.

Entry Point

The `main` class that we will use as example is very straightforward:

```
public class Main
{
    public static void main(String[] pArgs)
    {
        // All we have to do here is kickoff the creation of the launcher.
        // The launcher will do everything that is required to start for us.
        //
        // In a real world scenario and/or application there might be more
        // setup or groundwork required, for example processing the
arguments,
        // but we don't need any of that here.
        new SwingLauncher();
    }
}
```

All we have to do there is start the launcher itself. As the comment suggests, there might be work required for a "real" application startup. For this example, however, it is all we need to do. Of course, we could also directly embed this little function into the launcher implementation itself to save us one

class.

The Launcher

The `ILauncher` implementation, on the other hand, contains quite a bit of logic but nothing unmanageable:

```
public class SwingLauncher extends SwingFrame
                           implements ILauncher
{
    // We have to extend from SwingFrame because there is no factory
    // instantiated at that point, so we can't use UI components.

    private IApplication application;

    public SwingLauncher()
    {
        super();

        try
        {
            SwingUtilities.invokeAndWait(this::startup);
        }
        catch (InvocationTargetException | InterruptedException e)
        {
            e.printStackTrace();
        }
    }

    @Override
    public void dispose()
    {
        try
        {
            // We must notify the application that we are being disposed.
            application.notifyDestroy();
        }
        catch (SecurityException e)
        {
            e.printStackTrace();
        }

        super.dispose();

        // We have to make sure that the application is exiting when
        // the frame is disposed of.
        System.exit(0);
    }
```

```
    private void startup()
    {
        // We create a new SwingFactory and it is directly registered as
global
        // instance, that means it will be used by all components which are
        // created from now on.
        UIFactoryManager.getFactoryInstance(SwingFactory.class);

        // Also we set it as our factory instance.
        setFactory(UIFactoryManager.getFactory());

        // Because the IApplication implementation we use is based upon
        // UI components (which is advisable) we have to wrap this launcher
        // in an UILauncher.
        UILauncher uiLauncher = new UILauncher(this);

        // Now we create the main application.
        // Note that the ExampleApplication is already based upon
        // UI components.
        application = new ExampleApplication(uiLauncher);

        // Then we add the application as content to the launcher.
        uiLauncher.add(application);

        // Perform some setup work and start everything.
        uiLauncher.pack();
        uiLauncher.setVisible(true);

        // We also have to notify the application itself.
        application.notifyVisible();
    }

    // SNIP
}
```

In short, the launcher is kicking off the Swing thread by invoking the startup method on the main Swing thread. This startup method will instantiate the factory and then create the application. From there, we only need to set it to visible and then our application has started.

The launcher extends from `SwingFrame`. That is required because there hasn't been a factory created yet that could be used by UI components to create themselves. If we'd try to use an UI component before creating/setting a factory, we would see the constructor of the component fail with a `NullPointerException`.

The method `startup()` is invoked on the main Swing thread, which also happens to be the main UI thread for JVx in this application. Once we are on the main UI thread, we can create the application, add it, and then set everything to visible.

The Application

The `IApplication` implementation is quite short because we extend `com.sibvisions.rad.application.Application`, an `IApplication` implementation created with UI components.

```
public class ExampleApplication extends Application
{
    public ExampleApplication(UILauncher pParamUILauncher)
    {
        super(pParamUILauncher);
    }

    @Override
    protected IConnection createConnection() throws Exception
    {
        // Not required for this example.
        return null;
    }

    @Override
    protected String getApplicationName()
    {
        return "Example Application";
    }
}
```

Because the launcher has previously started the technology and created the factory, we can now use UI components, which means we are already independent of the underlying technology. So, the `IApplication` implementation can already be used with different technologies and is completely independent.

Notes on the Launcher

As you might have noticed, in our example the launcher is a (window) frame. That makes sense for nearly every desktop GUI toolkit, as they all depend upon a window as the main method to display their applications. But the launcher could also be simpler: for example, just a call to start the GUI thread. Or it could be something completely different: for example, an incoming HTTP request.

Also, don’t forget that the launcher is providing additional functionality to the application, like saving file handles, reading and writing the configuration, and similar platform and toolkit-dependent operations. See the launcher for Swing for further details.

Conclusion

This example demonstrates how a simple launcher is implemented and why it is necessary to have a

launcher in the first place. Compared with the “JVx are of course a lot more complex than these examples, that is because they implement all the required functionality and also take care of a lot of boiler plate operations. It is taking care of all technology specific code and allows to keep your application free from knowing about the platform it runs on.

Databooks

Let’s talk about databooks, which allow access to data without any effort.

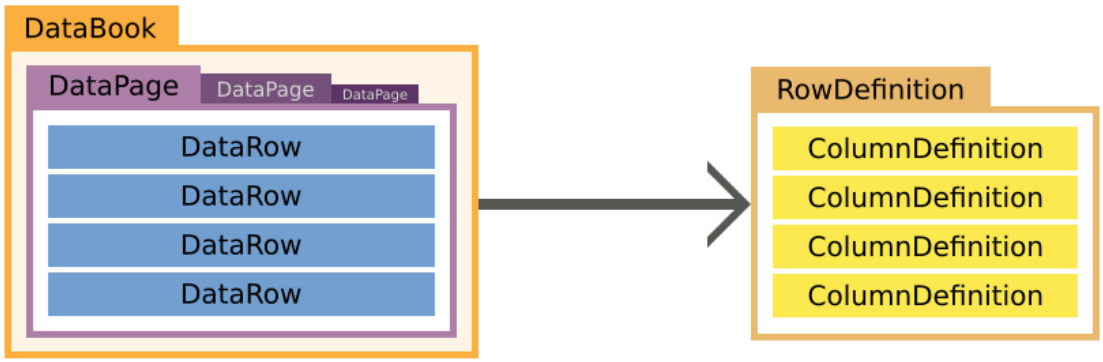
What Is It?

Databooks are an active model that allow you to directly query and manipulate the data. Contrary to many other systems, JVx does not map the data into objects, but allows you to directly access it in a table-like fashion exposing columns, rows, and values.

One could say that it is like a three dimensional array with these dimensions:

- DataPages
- DataRows
- Columns/Values

with DataPages containing DataRows, which in turn contain the values and everything referencing the RowDefinition, which further outlines how a row looks like.



Row Definition

The row definition defines what columns are available in the row and stores some additional information about them, like the names of the primary key columns. You can think of the row definition as the headers of a table.

Its creation and usage is rather simple, and, if you're working with RemoteDataBooks there is no need to create one at all, as it is automatically created when the databook is opened. A row definition holds and manages column definitions, which define the columns.

```
RowDefinition rowDefinition = new RowDefinition();
rowDefinition.addColumnDefinition(columnDefinitionA);
rowDefinition.addColumnDefinition(columnDefinitionB);
rowDefinition.addColumnDefinition(columnDefinitionC);

dataBook.setRowDefinition(rowDefinition);
```

Column Definition

The column definition defines and provides all necessary information about the column, like its datatype, its size, and whether it is nullable or not. You can think of it as one column in a table.

```
ColumnDefinition columnDefinition = new ColumnDefinition("NAME", new
StringDataType());
columnDefinition.setNullable(false);
```

Metadata

Most of the column definition is additional information about the column, like if it is nullable, the label of the column, default values, allowed values, and similar information.

Data Type

Of course, we must define what type the value in the column has. This is done by setting a data type on the column definition. The data type defines what kind of values the column holds, like if it is a string, a number, or something else. We provide the most often used data types out of the box:

- BigDecimal
- BinaryData
- Boolean
- Long
- Object
- String
- Timestamp

It is possible to add new data types by simply implementing `IDataType`.

Data Row

The data row represents a single row of data; it holds/references its own row definition and, of course, provides access to the values of the row. Accessing the data row can be done either by column index

or column name, and the methods either return or accept objects. Let's look at a simple usage example:

```
DataRow dataRow = new MemDataRow(rowDefinition);

String value = (String)dataRow.getValue("COLUMN_A");

dataRow.setValue("COLUMN_A", "New Value");
```

Data Page

The data page is basically a list of data rows. It also holds its own row definition, which is shared with all the contained data rows.

The main usage of data pages is to allow paging in a master/detail relationship. If the master selects a different row, the detail databook selects the related data page.

Databook

The databook is the main model of JVx, it provides direct access to its current data page and data row by extending from IDataRow and IDataPage.

By default, the databook holds one data page and only has multiple data pages if it is the detail in a master/detail relationship.

Usage Example

Here is a simple example of a `MemDataBook`, an `IDataBook` implementation that only operates in memory:

```
// Create a new instance.
IDataBook dataBook = new MemDataBook();
// Set the name.
dataBook.setName("test");
// Add some columns.
dataBook.getRowDefinition().addColumnDefinition(new ColumnDefinition("ID",
new LongDataType()));
dataBook.getRowDefinition().addColumnDefinition(new
ColumnDefinition("COLUMN_STRING", new StringDataType()));
// Open it, so that it can be used.
dataBook.open();

// Insert a new row.
dataBook.insert(false);
dataBook.setValue("ID", Long.valueof(0));
```

```
dataBook.setValue("COLUMN_STRING", "VALUE");

// Insert a new row.
dataBook.insert(false);
dataBook.setValue("ID", Long.valueof(1));
dataBook.setValue("COLUMN_STRING", "VALUE_A");

// Save the currently selected row.
// Note that the first one has been saved implicitely
// when the selection changed to the new row.
dataBook.saveSelectedRow();

// Change the first row.
dataBook.setSelectedRow(0);
dataBook.setValue("COLUMN_STRING", "VALUE_NEW");
dataBook.saveSelectedRow();

// Delete the second row.
dataBook.setSelectedRow(1);
dataBook.delete();
```

Accessing the Data With Strings

One of the major advantages of the databook concept is that there is no need to create new classes to represent each table, view, or query result. One can always use the databook directly and easily, and model changes don't necessitate changes on the client side. The downside to this approach is that we lose compile time checks because we access the data dynamically. However, this can be mitigated by using EPlug, an Eclipse plugin which provides compile time checks and many more features.

No Primitives, Objects Only

We do not provide overloads to fetch primitives. This is because there are mainly three types of data inside a database:

- Numbers
- Text
- Binary Data

Text and binary data are both objects (arrays of primitives are objects after all) and numbers are either primitives or objects. Most of the time, if we deal with numbers inside a database, we want them to be of arbitrary precision, which means we must represent them as `BigDecimal`. Supporting `double` or `float` in these cases would be dangerous because one might write a `float` into the database , which might or might not end up with the correct value in the database. To completely eliminate such problems, we only support objects, which means that one is "limited" to the usage of number extensions like `BigLong` and `BigDecimal`, which do not suffer from such problems.

Where Are the Data Pages?

What is not clear from this example is how and when data pages are used. As a matter of fact, most of the time there is no need to think about data pages because they are managed directly by the databook, and, if used like this, there is only one data page. Multiple data pages will be used if there is a master/detail relationship defined, in which case the databook selects the correct data page automatically.

Master/Detail

Master/detail is something that occurs in nearly every data model. It simply means that there is one master data set that is referenced by one or multiple detail data sets. Or to express it in SQL:

```
SELECT
  *
FROM
  MASTER m
  LEFT JOIN DETAIL d ON m.ID=d.MASTER_ID;
```

We can, of course, express a master/detail relationship when using databooks. For that, we just create a `ReferenceDefinition` and assign it to the detail databook:

```
// Create the master.
IDataBook masterDataBook = new MemDataBook();
masterDataBook.setName("master");
masterDataBook.getRowDefinition().addColumnDefinition(new
ColumnDefinition("ID", new LongDataType()));
masterDataBook.open();

// Create the detail.
IDataBook detailDataBook = new MemDataBook();
detailDataBook.setName("detail");
detailDataBook.getRowDefinition().addColumnDefinition(new
ColumnDefinition("ID", new LongDataType()));
detailDataBook.getRowDefinition().addColumnDefinition(new
ColumnDefinition("MASTER_ID", new LongDataType()));
// Set it as detail of the master.
detailDataBook.setReferenceDefinition(new ReferenceDefinition(new Streing[]
{"MASTER_ID"}, masterDataBook, new String[] {"ID"});
detailDataBook.open();
```

Let’s assume the following data for illustrative purposes:

```
MASTER              DETAIL
======        =================
  ID            ID   | MASTER_ID
------        ------|----------
     1             1|         1
```

```
     2              2|          1
     3              3|          2
                    4|          2
                    5|          2
                    6|          3
                    7|          3
                    8|          3
```

Now, if we select the second row in the `masterDataBook`, the `detailDataBook` will just contain the rows with the corresponding `MASTER_ID`, so 3, 4, and 5.

```
MASTER              DETAIL
======         =================
  ID             ID   | MASTER_ID
------         ------|----------
     1              3|          2
S    2              4|          2
     3              5|          2
```

The `detailDataBook` is automatically adjusted according to the selection in the `masterDatabook`. Of course, this can have an arbitrary depth too.

Conclusion

The databook is the backbone of JVx: it provides a clean and easy way to access and manipulate data. At the same time, it is flexible and can be customized to specific needs with ease.

Application Basics

Let’s talk about the basics: how a JVx application starts, how it works, and how the connection strings together the client and server side.

Multitier Architecture

JVx is designed to be Multitier by default. It allows a clean and easy separation of processes and makes it easy to build, maintain and extend applications by separating the client, server and data storage.

Launchers

The following method is a simplified way to launch a JVx application. Normally, you’d use the technology specific launcher to launch the application. These launchers do know exactly what is

required to set it up and start the technology and the application. However, covering the launchers is out of scope for this post, so we will review them and their mechanics in a follow-up.

The Simplest JVx Application: Just the GUI

But first, we will start without anything. The most simple application you can create with JVx is an application which opens a single window and only works with in-memory data (if at all). This can be easily achieved by “just starting” the application.

The JVx GUI is a simple layer on top of the Technology which implements the actual functionality. So if we want to have a GUI we’ll need to initialize the factory before doing anything else:

```
UIFactoryManager.getFactoryInstance(SwingFactory.class);
```

With this little code, we have initialized everything we need to create a simple Swing application. Now we can start to create and populate a window with something:

```
UIFrame frame = new UIFrame();
frame.setLayout(new UIBorderLayout());
frame.addComponent(new UILabel("Hello World!"));

frame.pack();
frame.setVisible(true);

frame.eventWindowClosed().addListener(() -> System.exit(0));
```

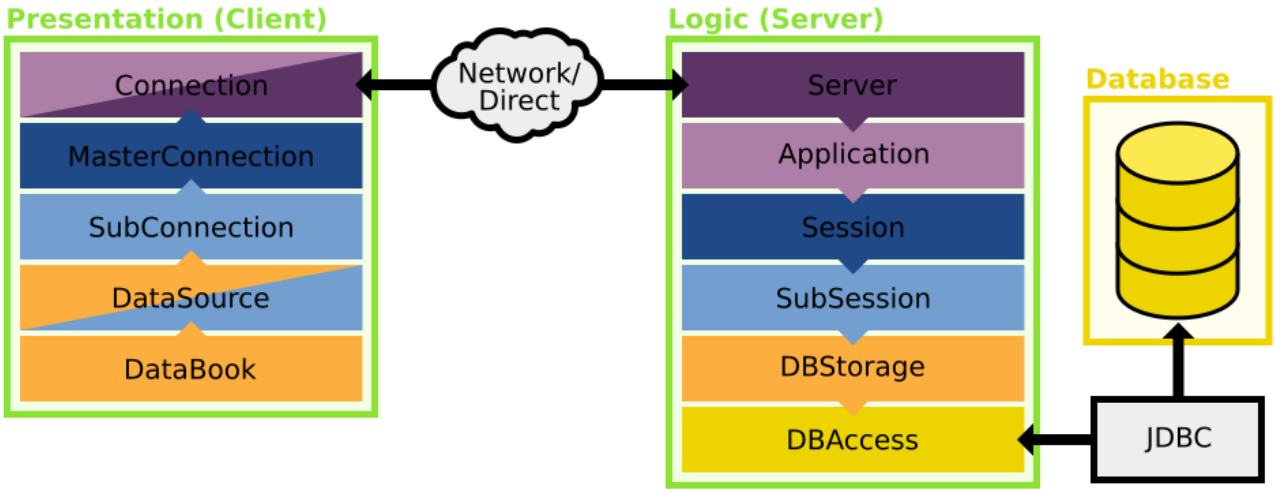
We can start to create and manipulate the GUI. In this case, we are building a simple window with a label inside. Lastly, we make sure that the JVM will exit when the window is closed.

A very good example and showcase for that is the JVx Kitchensink.

That’s it! That is the most simple way to start a JVx application. We can use all controls, and we can use `MemDataBooks` without any problem or limitation. Best of all, we can simply switch to another technology by using another factory.

Anatomy of a Remote JVx Application

Of course, JVx wouldn’t be that useful if it would just provide static GUI components. Now, to explain what else is required for a remote JVx application, I have to go far afield, so let’s head down the rabbit hole.



What you are seeing here is a rough sketch of how the architecture of JVx looks like. Let's walk through the image step by step. We will look at each successive layer and work our way from the database on the server to the databook on the client.

DBAccess, Accessing a Database

Accessing a database is easy when using `DBAccess`. All we must do is to set the JDBC URL of the server and connect to it:

```
DBAccess dbAccess = DBAccess.getDBAccess(
        "jdbc:postgresql://server:5432/schema",
        "username",
        "password");
dbAccess.open();
```

As a note, the instance returned by `getDBAccess(...)` is the database-specific `DBAccess` extension, which knows how to handle its database.

We can, of course, use `DBAccess` to directly access the database:

```
dbAccess.executeStatement("insert into SOME_TABLE values (?, ?);",
        BigDecimal.valueOf(1),
        "Some Value");

List<Bean> data = dbAccess.executeQuery("select * from SOME_TABLE");
```

...or manipulate the database, query information about the database, execute procedures, or anything else!

DBStorage, Preparing the Database Access for Databooks

The downside of using DBAccess is that everything must be database-specific. To become database-neutral, we must use `DBStorage`. `DBStorage` does not care which database it is connected to and

can operate on any of them:

```
DBStorage storage = new DBStorage();
storage.setDBAccess(dbAccess);
storage.setWritebackTable("SOME_TABLE");
storage.open();
```

We can use this to insert, update, delete and fetch data. Additionally, the `DBStorage` does retrieve and manage the metadata of the table we've set, which means that we can query all column names, what type they are, we can even access the indexes and the default values. Short, the `DBStorage` leaves little to be desired when it comes to operating on a database.

If we query data from the `DBStorage`, we receive a list of rows. The rows are either represented as `Object` array, `IBean`, or a POJO, and we can easily manipulate the data, like this:

```
for (IBean row : storage.fetchBean(null, null, 0, -1))
{
    row.put("SOME_COLUMN", "new value");
    storage.update(row);
}
```

As one can see, it looks quite familiar to the DataBook, which isn't a coincidence. The `DBStorage` "powers" the databooks on the server side, a databook will get its data from and will send its modified data to the `DBStorage`.

I've been using the `DBStorage` here as an example, but actually the storage is not dependent on a database. `IStorage` can be implemented to provide any sort of data provider, like reading from an XML or JSON file, scraping data from a website, fetching data from a different process, or reading it directly from a hardware sensor.

Life Cycle Objects, the Business Objects With All the Logic

Life Cycle Objects, or LCOs, are the server side business objects which contain and provide the business logic. They are created and destroyed as is requested by the client side and are used to provide specific functionality to the client, like providing functionality specific to one screen or workflow. This is done by RPC, Remote Procedure Calls, which means that the client is directly calling the methods defined in the LCOs, which includes getting the Storages for the DataBooks.

There is also a security aspect to these, as you can permit one client access to a certain LCO but lock out everyone else, which means that only that client can use the functionality provided by the LCO.

But let's not get ahead of our selves, there are three important "layers" of LCOs which we will look at.

Application

The LCO for the application represents the application itself and provides functionality on the application layer. It is created once for the lifetime of the application and this instance is shared by all sessions.

```
public class Application extends GenericBean
{
}
```

Session

The LCO for the session represents one session, which most of the time also equals one client connection. It provides functionality which should be session-local, like providing the database connection which can be used.

```
public class Session extends Application
{
    protected DBAccess getDBAccess() throws Exception
    {
        // Code for initialization and caching of DBAccess goes here.
    }
}
```

Sub-Session, AKA Screen

The sub-session, also known as screen, LCO is the last in the chain. It provides functionality specific to a certain part of the application, like a single screen, and provides the storages required to power the databooks and other functionality.

```
public class MySubSession extends Session
{
    public DBStorage getTablename() throws Exception
    {
        // Code for initialization and caching of DBStorage goes here.
    }
}
```

Server, Serving It Up

There really isn't much to say about the server; it accepts connections and hands out sessions. Of course, it is not that easy, but for this guide we do not need to go into any further detail.

Connection, Connecting to a Server

The connection that strings together the client and the server is used for, obviously, the communication between them. It can be anything from a simple direct connection that strings two objects together to an HTTP connection that talks with a server on the other side of the planet.

By default, we provide different `IConnection` implementations, the `DirectServerConnection`, `DirectObjectConnection`, the `HttpConnection` and the `VMConnection`. The

`DirectServerConnection` is a simple `IConnection` implementation that forwards method calls to known objects without a layer of indirection and is used when the client and server reside inside the same JVM. The `HttpConnection` communicates with the server over an HTTP connection and is used whenever the client and server are not inside the same JVM. The `DirectObjectConnection` and `VMConnection` are used for unit tests.

As an example, we will use the `DirectServerConnection`, which serves as server and connection. It is used if the server and client reside in the same JVM.

```
IConnection connection = new DirectServerConnection();
// The connection will be automatically opened by the MasterConnection.
```

Master- and Sub-Connections, Client-Side Life Cycle Management

The `MasterConnection` is the main connection that is used to access the server and its functionality. When a `MasterConnection` is established, a session LCO on the server is created.

```
MasterConnection masterConnection = new MasterConnection(connection);
masterConnection.open();
```

A `SubConnection` is a sub-connection of the `MasterConnection` and allows us to access specific functionality encapsulated in an LCO. When a `SubConnection` is established, the requested/specified LCO on the server is created and can be accessed through the `SubConnection`

```
SubConnection subConnection =
masterConnection.createSubConnection("MySubSession");
subConnection.open();
```

The `SubConnection` can now access the functionality provided by the application, the session, and the LCO that was specified.

```
subConnection.callAction("doSomethingOnTheServer");
```

DataSource, Preparing the Connection for the Databook

To provide data to the databooks, we can use the connection we've described earlier. However, the databook does not directly know about the connection; it expects an `IDataSource`, which is used as an intermediate:

```
IDataSource dataSource = new RemoteDataSource(subConnection);
dataSource.open();
```

Of course, the `RemoteDataSource` is just one possible implementation of `IDataSource` that can be used to provide data to the databook.

Databook, Accessing Data

And now we are at the other end of the chain,: at the databook on the client side. We just need to tell our databook what data source to use, and we are done.

```
RemoteDataBook dataBook = new RemoteDataBook();
dataBook.setDataSource(dataSource);
dataBook.setName("storagename");
dataBook.open();
```

The name of the databook is used to access the DBstorage object in the LCO provided by the data source. The mechanism for that is a simple search for a getter with the set name.

Interactive Demo

There is an interactive demo on our blog that allows you to explore the connections between the client- and server side. The complement classes are always highlighted, and you can click on the names of the objects to receive additional information.

The JVx Application: Manual Example

Now that we have seen all layers that make up the architecture of JVx, let's put all of that into code:

```
public class JVxLocalMain
{
    public static void main(String[] pArgs) throws Throwable
    {
        // ########################## Server ############################

        // ------------------------ DBAccess ------------------------

        // The DBAccess gives us access to the database.
        DBAccess dbAccess = DBAccess.getDBAccess(
                "jdbc:h2:mem:database",
                "",
                "");
        dbAccess.open();

        // We'll insert some data for this example.
        dbAccess.executeStatement("create table if not exists TEST("
                + "ID int primary key auto_increment,"
                + "NAME varchar(128) default '' not null);");
        dbAccess.executeStatement("insert into TEST values (1, 'Name A');");
        dbAccess.executeStatement("insert into TEST values (2, 'Name B');");
        dbAccess.executeStatement("insert into TEST values (3, 'Name C');");
```

```
        // ------------------------ DBStorage ------------------------

        // Our sole storage.
        DBStorage testStorage= new DBStorage();
        testStorage.setDBAccess(dbAccess);
        testStorage.setWritebackTable("TEST");
        testStorage.open();

        // --------------- LCO / Session / Application --------------

        // We are skipping the LCO, Session and Application in this example.

        // ################### Network / Connection ###################

        // For this example we are initializing a DirectObjectConnection,
which
        // does not require a server.
        // It is designed to be used mainly for unit testing.
        DirectObjectConnection connection = new DirectObjectConnection();
        connection.put("test", testStorage);

        // ########################## Client ##########################

        // --------------------- MasterConnection ---------------------

        MasterConnection masterConnection = new
MasterConnection(connection);
        masterConnection.open();

        // ---------------------- SubConnection -----------------------

        // We are skipping the SubConnection in this example.

        // ------------------------ DataSource ------------------------

        IDataSource dataSource = new RemoteDataSource(masterConnection);
        dataSource.open();

        // ------------------------ DataBook --------------------------

        RemoteDataBook dataBook = new RemoteDataBook();
        dataBook.setDataSource(dataSource);
        dataBook.setName("test");
        dataBook.open();

        // You can use the DataBook here.

        // Perform cleanup of all opened objects here.
    }
}
```

With this little example, we have a wholly working JVx application. We provide ways to create most of this out of the box and read most of it from configuration files, so there really is little code to be written. See the JVx FirstApp as a perfect example of this.As you can see, there is rarely any need to write such code; all you have to do is create a new application and start it.

Additionally, we could combine this long example with the simple one from before to initialize and create a GUI that could use our `RemoteDataBook`, like this:

```
// Insert after the RemoteDataBook has been created.

// Set the UI factory which should be used, in this case it is
// the SwingFactory.
UIFactoryManager.getFactoryInstance(SwingFactory.class);

UIFrame frame = new UIFrame();
frame.setLayout(new UIBorderLayout());
frame.add(new UITable(dataBook));

frame.pack();
frame.setVisible(true);

frame.eventWindowClosed().addListener(() -> System.exit(0));
```

Abstractions at Every Step

As you can see, you always have full control over the framework and can always tailor it to your needs. There is always the possibility to provide a custom implementation:

- Accessing an unsupported database can be achieved by extending `DBAccess`.
- Having a different service/way of providing data can be implemented on top of `IStorage`.
- Supporting a different connection can be implemented on top of `IConnection`.
- And a completely different way of providing data can be implemented on top of `IDataSource`.

You can swap out every layer and provide custom and customized implementations, which work exactly as you require.

Just Like That

Just like that, we've walked through the whole stack of a JVx application, from the database, which holds the data, all the way to the client GUI. Of course, there is much more going on in a full-blown JVx application. For example, I've spared you here the details of the configuration, server, network, and providing actual LCOs. However, all in all, this should get you going.

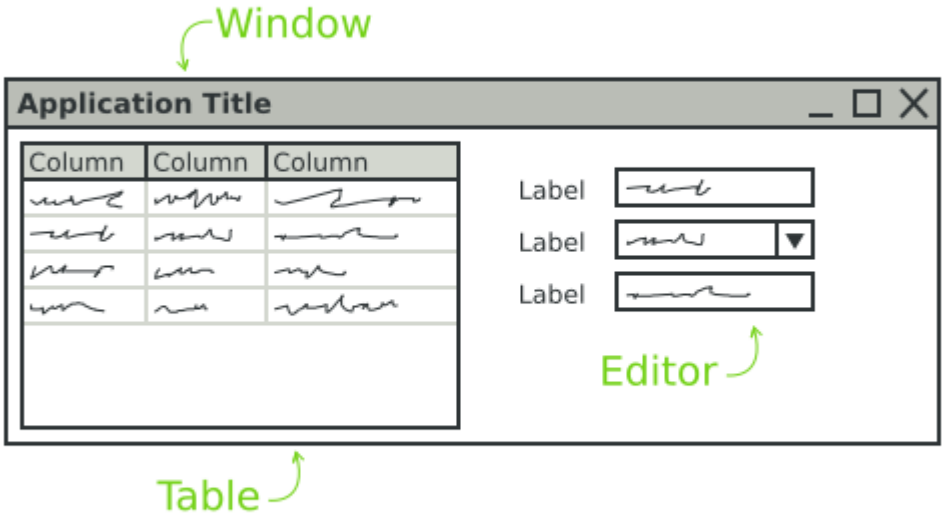
CellEditors

Let’s talk about CellEditors and how they are decoupled from the surrounding GUI.

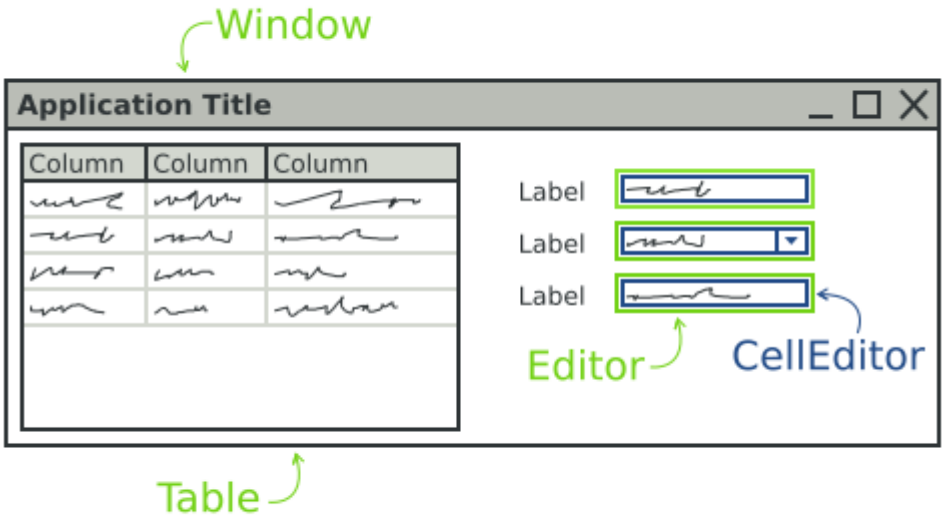
What Are They?

While we’ve already covered large parts of how the GUI layer of technologies and factories and JVx work, the CellEditors have been left completely untouched and unmentioned. One might believe that they can be easily explained together with the editors. However, they are a topic of their own – and a complex one at that.

The difference between editors (the `UIEditor` for the most part) and CellEditors is that the editors only provide the high-level GUI control, while the CellEditors provide the actual functionality. Let’s take a look at a simple screen.



We see a window with a table on the left and some editors on the right, simple enough. Now these components we are seeing are `UIEditors`, not CellEditors. The CellEditors themselves are only added as child components to the editors, so the editors are basically just panels which contain the actual CellEditor.



Technically, every `UIEditor` is just another panel that gets the CellEditor added. The CellEditors themselves follow the same pattern as all GUI components in JVx: there is the base interface, an eventual extension of technology components, the implementation, and, finally, the UI object. They are, however, rarely directly used in building the GUI but mostly only referenced when building the model.

Why Do They Exist?

If you want to make GUI editor components, I know of two possible ways off the top of my head to achieve that: you create dedicated editor components for the datatypes that are available, for example a NumberEditor, TextEditor, and so forth, or you create one editor component that acts as a mere container and allows you to plug in any wanted behavior for the type you're editing.

We've opted for the second option because it means that the GUI is actually decoupled from the datatypes (and in extension the data) of the model. If we'd have separate components for each datatype, changing the datatype of a single column would mean that you'd have to touch all editors associated with that column and change that code, maybe with rippling effects on the rest of the GUI. With the CellEditors, one can change the datatype of a column and not worry about the GUI that is associated with that column. The CellEditor is changed on the model once and that change is automatically picked up by all editors. This also means that one can define and change defaults very easily and globally.

Of course, one can also set the preferred or wanted CellEditor directly on the editor instead of using the one defined in the model should the need arise.

And the Table?

The same applies to the table. Theoretically, every cell of the table can be viewed as a single editor, for this context at least. So a single cell behaves the same as an editor when it comes to how the CellEditors are handled.

How Many Are There?

JVx comes with a variety of CellEditors out of the box:

- Boolean
- Choice
- Date/Time
- List
- Number
- Text
  - HTML
  - Multiline
  - Password
  - Standard

With these, nearly all needs can be covered. If there is need for a new one, it can be created and added like any other UI component.

Using CellEditors

As said previously, which CellEditor is used is defined primarily with the model. For example:

```
private void initiliazeModel() throws ModelException
{
    dataBook = new MemDataBook();

    ICellEditor cellEditor = new UITextCellEditor();
    IDataType dataType = new StringDataType(cellEditor);
    ColumnDefinition column = new ColumnDefinition("COLUMN", dataType);

    RowDefinition rowDefinition = dataBook.getRowDefinition();
    rowDefinition.addColumnDefinition(column);

    dataBook.open();
}

private void initializeUI() throws ModelException
{
    editor = new UIEditor(dataBook, "COLUMN");
```

```
    add(editor);
}
```

We can see that every column has a datatype and every datatype has a CellEditor. That allows the model to provide the actual editing functionality without changing the GUI code. The editor, when `notifyRepaint()` is called, will fetch the CellEditor from the datatype and use it. Additionally, there is a technology-dependent default mechanism that allows this system to work even when the UI classes are not used.

Let’s do a step-by-step explanation of what happens:

- The model is created.
- The GUI is created.
- The model invokes `notifyRepaint()` on all bound controls.
- The editor gets the CellEditor from the model and adds it to itself.

Instance Sharing

If we revisit the example code from above, we will notice that the CellEditor instance is set on the model and must then be used by the editor. That means that a single CellEditor instance is used for all bound editors. We all know that sharing instances in such a way can be fun, but, in this case, it is not a problem because CellEditors are only “factories” for the actual editing components.

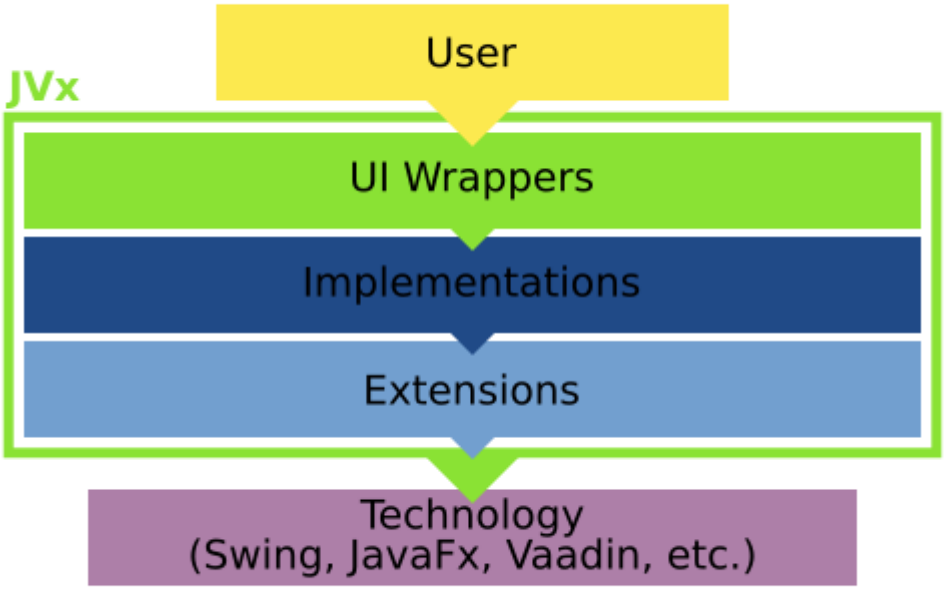
The `ICellEditor` interface actually only specifies two methods: whether or not it is a direct cell editor, and the factory method for creating an `ICellEditorHandler`. The CellEditorHandler is the manager of the instance of the component that is going to be embedded into the editor.

- `notifyRepaint()` is called on the editor.
- The Editor gets the CellEditorHandler from the CellEditor.
- The Editor gets the component from the CellEditorHandler and embeds it.

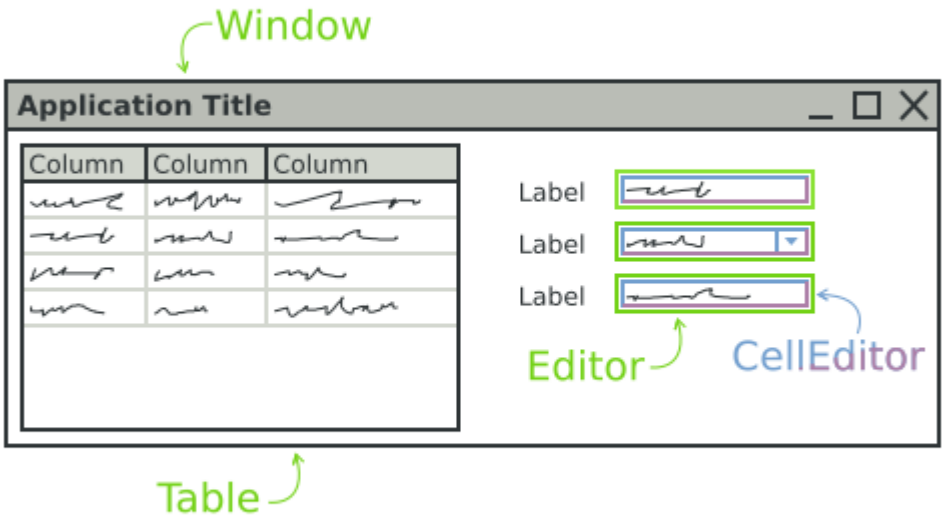
This mechanism makes sure that no component instances end up shared between different GUI components.

A Closer Look at the CellEditorHandler

If we take a good look at the CellEditorHandler interface, we see that it contains everything that is required for setting up a component to be able to edit data coming from a data row. One method is especially important: the `getCellEditorComponent()` function. It returns the actual technology component that is to be embedded into the editor. That means that, even though there are implementations for the CellEditors on the UI layer, the actual components that will provide the functionality for editing the data are implemented on the technology layer. A short refresher:



Revisiting our simple screen from above, we'd actually need to represent it as something like this:



Because the embedded components in the editor are actually on the technology layer.

CellRenderers

There is another small topic we need to discuss, CellRenderers. They follow nearly the same schematics as CellEditors but are used to display values directly; for example, values in a table cell. The table is also the primary component that uses them to display the cell values until the editing is started. For simplicity's sake, most CellEditors implement `ICellRenderer` directly and provide management of the created component. That is because the reuse of components for barely displaying values is easier and doesn't have as much potential for error.

Conclusion

CellEditors provide an easy way to edit data, and, more importantly, they are decoupled from the GUI code in which they are used in a way that allows the model to change, even dynamically. This enables programmers to create and edit screens and models quickly without the need to check if the GUI and the model fit together because they always will.

Custom Components

Let’s talk about custom components, and how to create them.

The GUI of JVx

We’ve previously covered how the GUI of JVx works, and now we will have a look at how we can add custom components to the GUI.

In the terminology of JVx, there are two different kinds of custom components:

- UI based
- Technology based

We will look at both, of course.

Custom Components at the UI Layer

The simplest way to create custom components is to extend and use already existing UI classes like `UIPanel` or `UIComponent`. These custom components will be technology-independent because they use technology-independent components. There is no need to know about the underlying technology. You can think of them as a “remix” of already existing components.

The upside is that you never have to deal with the underlying technology, but the downside is that you can only use already existing components (custom drawing is not possible).

Let’s look at a very simple example. Te will extend the `UILabel` to always display a certain postfix along with the set text:

```
public class PostfixedLabel extends UILabel
{
    private String postfix = null;

    // We must store the original text so that we can return
    // it if requested. Otherwise we could only return the text
    // with the appended postfix, which works unless the postfix
    // changes.
    private String text = null;

    public PostfixedLabel()
    {
        super();
    }

    public PostfixedLabel(String pText)
    {
        super(pText);
    }

    public PostfixedLabel(String pText, String pPostfix)
    {
        super(pText);

        setPostfix(pPostfix);
    }

    @Override
    public String getText()
    {
        return text;
    }

    @Override
    public void setText(String pText)
    {
        text = pText;

        if (!StringUtil.isEmpty(postfix) &amp;&amp;
!StringUtil.isEmpty(pText))
        {
            // We translate the text and the postfix now separately,
            // the underlying label will obviously try to translate
            // the concatenated version.
            super.setText(translate(pText) + translate(postfix));
        }
        else
        {
            super.setText(pText);
        }
    }
```

```
    public String getPostfix()
    {
        return postfix;
    }

    public void setPostfix(String pPostfix)
    {
        postfix = pPostfix;

        // If the postfix changed, we must update the text.
        setText(text);
    }
}
```

It will be treated just like another label, but every time a text is set, the postfix is appended to it.

Another example: we want a special type of component one that always does the same but will be used in many different areas of the application. It should contain a label and two buttons. The best approach for a custom component that should not inherit any specific behavior is to extend `UIComponent`:

```
public class BeepComponent extends UIComponent<UIPanel>
{
    public BeepComponent()
    {
        super(new UIPanel());

        UIButton highBeepButton = new UIButton("High Beep");
        highBeepButton.eventAction().addListener(Beeper::playHighBeep);

        UIButton lowBeepButton = new UIButton("Low Beep");
        highBeepButton.eventAction().addListener(Beeper::playLowBeep);

        UIFormLayout layout = new UIFormLayout();

        uiResource.setLayout(layout);
        uiResource.add(new UILabel("Press for beeping..."),
layout.getConstraints(0, 0, -1, 0));
        uiResource.add(highBeepButton, layout.getConstraints(0, 1));
        uiResource.add(lowBeepButton, layout.getConstraints(1, 1));
    }
}
```

So we extend `UIComponent` and set a new `UIPanel` as UIResource on it, which we can use later and which is the base for our new component. After that, we added a label and two buttons which will play beep sounds if pressed. This component does not expose any specific behavior as it extends `UIComponent`. It only inherits the most basic properties, like background color and font settings, yet it can easily be placed anywhere in the application and will perform its duty.

Custom Controls at the Technology Layer

The more complex option is to create a custom component at the technology layer. That means we have to go through a multi-step process to create and use the component:

1. Create an interface for the functionality you'd like to expose
2. Extend the technology component (if needed)
3. Implement the necessary interfaces for JVx
4. Extend the factory to return the new component
5. Create a UIComponent for the new component
6. Use the new factory

I will walk you through this process step by step.

The upside is that we can use any component that is available to us in the technology. The downside is that it is quite a bit of work to build the correct chain, ideally for every technology.

Creating an Interface

The first step is to think about what functionality the component should expose. We will use a progress bar as an example. We don't want anything fancy for now, a simple progress bar on which we set a percent value should be more than enough:

```
/**
 * The platform and technology independent definition for a progress bar.
 */
public interface IProgressBar extends IComponent
{
    /**
     * Gets the current value, in percent.
     *
     * @return the current value. Should be between {@code 0} and {@code
100}.
     */
    public int getValue();

    /**
     * Sets the current value, in percent.
     *
     * @param pValue the value. Should be between {@code 0} and {@code 100}.
     */
    public void setValue(int pValue);
}
```

It might not be the most sophisticated example (especially in regards to documentation), but it will do for now. This interface will be the foundation for our custom component.

Extending the Component, if Needed

We will be using Swing and the `JProgressBar` for this example, so the next step is to check if we must add additional functionality to the technology component. In our case we don't, as we do not demand any behavior that is not provided by `JProgressBar`, but, for the sake of the tutorial, we will still create an extension of `JProgressBar`.

```
public class ExtendedProgressBar extends JProgressBar
{
    public ExtendedProgressBar(int pMin, int pMax)
    {
        super(pMin, pMax);
    }
}
```

Within this class, we could now implement additional behavior independent of JVx. For example, we provide many extended components for Swing, JavaFX, and Vaadin with additional features but without depending on JVx. The extension layer is the perfect place to extend already existing components with functionality that will be used by, but is not dependent on, JVx.

Creating the Implementation

The next step is to create an implementation class that allows us to bind our newly extended `JProgressBar` to the JVx interfaces. Luckily, there is the complete Swing implementation infrastructure we can use:

```
public class SwingProgressBar extends SwingComponent
                              implements IProgressBar
{
    public SwingProgressBar()
    {
        // We can hardcode the min and max values here, because
        // we do not support anything else.
        super(new ExtendedProgressBar(0, 100));
    }

    @Override
    public int getValue()
    {
        return resource.getValue();
    }

    @Override
    public void setValue(int pValue)
    {
        resource.setValue(pValue);
    }
}
```

That's it already. Again, in this case it is quite simple because we do not expect a lot of behavior. The implementation layer is the place to "glue" the component to the JVx interface, implementing missing functionality that is depending on JVx and "translating" and forwarding values and properties.

Extending the Factory

Now, we must extend the factory to be aware of our new custom component, which is equally as simple as our previous steps. First, we extend the interface:

```
public interface IProgressBarFactory extends IFactory
{
    public IProgressBar createProgressBar();
}
```

And then we extend the SwingFactory:

```
public class ProgressBarSwingFactory extends SwingFactory
                                     implements IProgressBarFactory
{
    @Override
    public IProgressBar createProgressBar()
    {
        SwingProgressBar progressBar = new SwingProgressBar();
        progressBar.setFactory(this);
        return progressBar;
    }
}
```

Again, it is that easy.

Creating the UIComponent

So that we can use our new and shiny progress bar easily, and without having to call the factory directly, we wrap it one last time in a new `UIComponent`:

```
public class UIProgressBar extends UIComponent<IProgressBar>
                           implements IProgressBar
{
    public UIProgressBar()
    {
        // We'll assume that, whoever uses this component,
        // is also using the correct factory.
super(((IProgressBarFactory)UIFactoryManager.getFactory()).createProgressBar
());
    }

    @Override
    public int getValue()
```

```
    {
        return uiResource.getValue();
    }

    @Override
    public void setValue(int pValue)
    {
        uiResource.setValue(pValue);
    }
}
```

Nearly done, we can almost use our new and shiny component in our project.

Using the Custom Factory

Of course, we have to tell JVx that we want to use our factory and not the default one. Depending on the technology used, this is done at different places.

Swing and JavaFX

Add the factory setting to the application.xml of the application:

```
<Launcher.uifactory>your.package.with.custom.components.SwingProgressBarFactory</Launcher.uifactory>
```

Vaadin

Add the following setting to the web.xml under the WebUI Servlet configuration:

```
<init-param>
    <param-name>Launcher.uifactory</param-name>
    <param-value>your.package.with.custom.components.VaadinProgressBarFactory</param-value>
</init-param>
```

Using Our New Component

Now we are done. From here we can use our custom component like any other.

```
UIProgressBar progressBar = new UIProgressBar();
progressBar.setValue(65);

// Skip
```

```
add(progressBar, constraints);
```

Wrapping Custom Components With UICustomComponent

There is a third way to have technology-dependent custom components in JVx. You can wrap them within a `UICustomComponent`:

```
JProgressBar progressBar = new JProgressBar(0, 100);
progressBar.setValue(100);

UICustomComponent customProgressBar = new UICustomComponent(progressBar);

// Skip

add(customProgressBar, constraints);
```

This has the upside of being fast and easy, but the downside is that your code has to know about the currently used technology and it is no longer easily portable.

Conclusion

As you can see, there are multiple ways of extending the default set of components that are provided by JVx, depending on the use case and what custom components are required. It is very easy to extend JVx with all the components one requires.

FormLayout

Let’s talk about the FormLayout and why the anchor system makes it much more flexible than a simple grid.

Basics

JVx comes with five layouts out of the box:

- null/none/manual
- BorderLayout
- FlowLayout
- GridLayout
- FormLayout

Of these five, the first four are easily explained. Only the FormLayout needs some more information because it might not be as easy to grasp off the bat as the others.

The FormLayout uses a dependent anchor system. An anchor in this context is a position inside the layout that is calculated from parent anchors and either the size of the component or a fixed value. So we can say there are two different types of anchors inside the FormLayout that we are concerned about:

- Autosize anchors, its position is calculated from the component assigned to it.
- Fixed anchors, its position is fixed.

Additionally, there are three special cases of fixed anchors:

- Border anchors, which surround the FormLayout at its border.
- Margin anchors, which are inset from the border by the defined value.
- Gap anchors, which are added to create a gap between components.

When it comes to calculating the position of an anchor, the position of the parent anchor is determined, and then the value of the current anchor is added (which is either the size of a component or a fixed value). Simplified and in pseudo-code it can expressed like this:

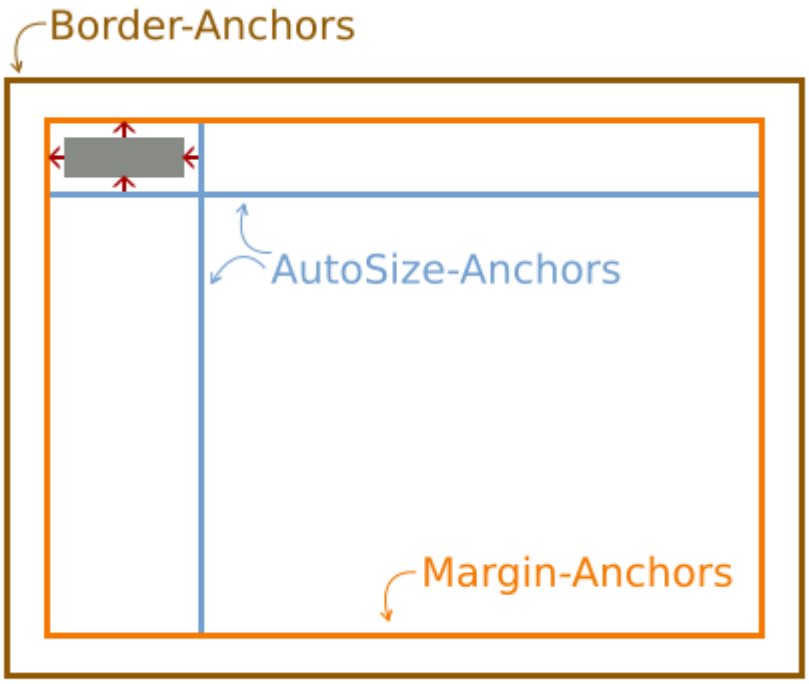
```
public int getPosition(Anchor pAnchor)
{
    int parentPosition = 0;

    if (pAnchor.getParent() != null)
    {
        parentPosition = getPosition(pAnchor.getParent());
    }

    if (pAnchor.isAutoSize())
    {
        return parentPosition + pAnchor.getComponent().getWidth();
    }
    else
    {
        return parentPosition + pAnchor.getValue();
    }
}
```

With this knowledge, we are nearly done with completely understanding the FormLayout.

Creating Constraints

Now, the second important part after the basics is knowing how the constraints are created. For example this:

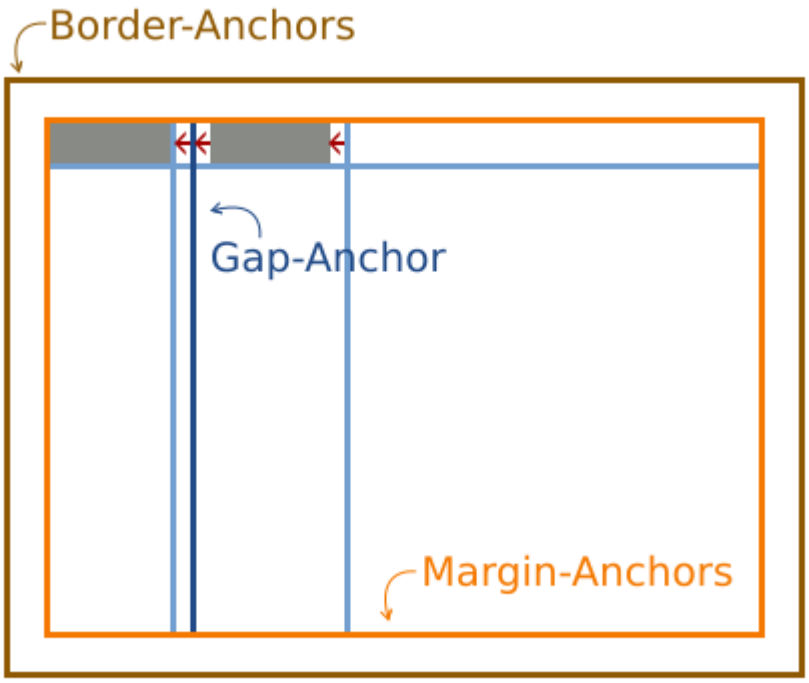
```
panel.add(component, layout.getConstraints(0, 0));
```



With the coordinates of `0,0`, no new anchors are created. Instead, the component is attached to the top and left margin anchor. Two new autosize anchors (horizontal and vertical) are created and attached to the component.

We now add a second component in the same row:

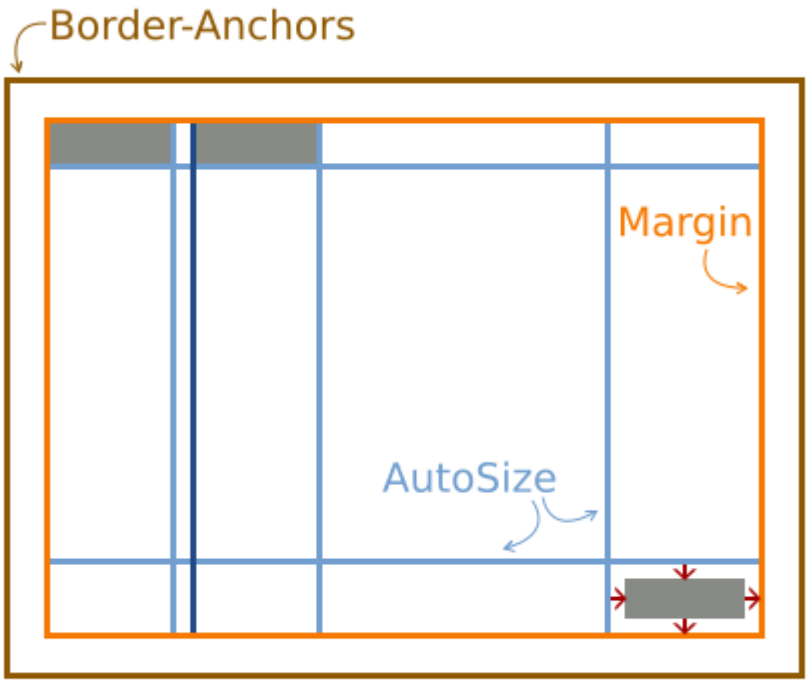
```
panel.add(component, layout.getConstraints(0, 0));
panel.add(component, layout.getConstraints(1, 0));
```



Because we are still on row `0`, the component is attached to the top margin anchor and the previous autosize anchor for this row. Then, a new gap anchor will be created, which is attached to the trailing autosize anchor of the previous component.

We can, of course, also add items to the right and bottom:

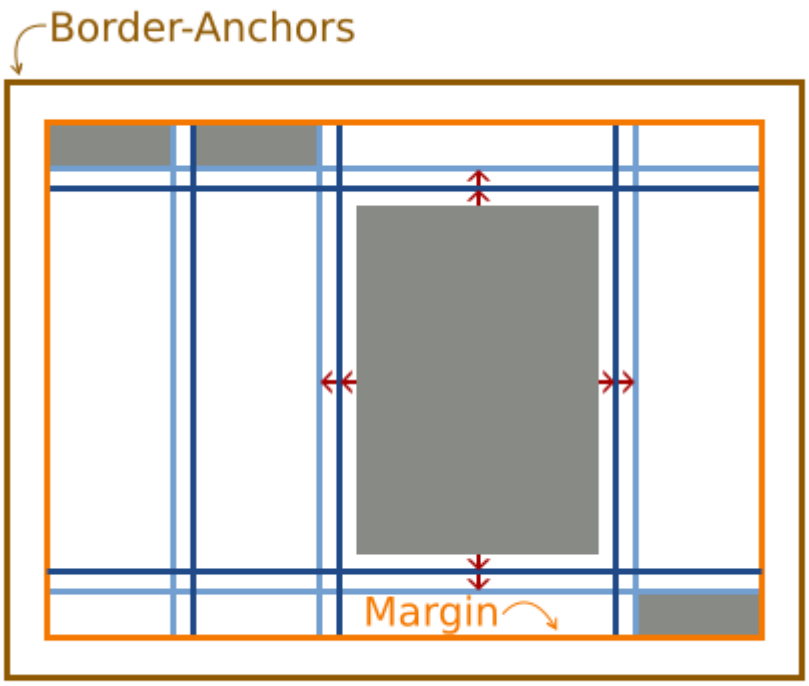
```
panel.add(component, layout.getConstraints(0, 0));
panel.add(component, layout.getConstraints(1, 0));
panel.add(component, layout.getConstraints(-1, -1));
```



What happens is the same as when adding a component at the coordinates `0,0`, except that the reference is the lower right corner. The component is attached to the bottom and right margin anchors with trialing autosize anchors.

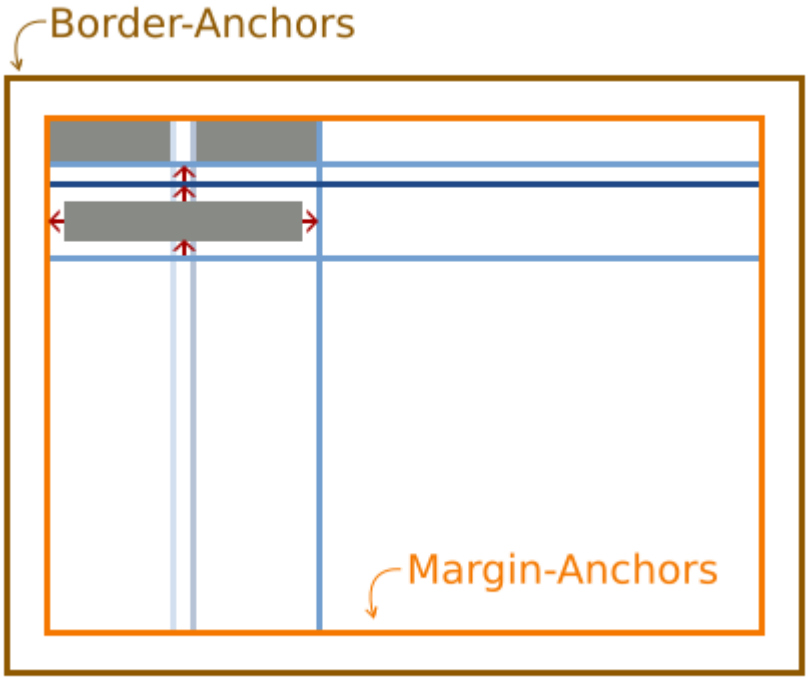
Last but not least, we can add components that span anchors:

```
panel.add(component, layout.getConstraints(0, 0));
panel.add(component, layout.getConstraints(1, 0));
panel.add(component, layout.getConstraints(-1, -1));
panel.add(component, layout.getConstraints(2, 1, -2, -2));
```



Again, the same logic as previously applies, with the notable exception that new gap anchors are created for all four sides. This includes variants that span anchors:

```
panel.add(component, layout.getConstraints(0, 0));
panel.add(component, layout.getConstraints(1, 0));
panel.add(component, layout.getConstraints(0, 1, 2, 1));
```



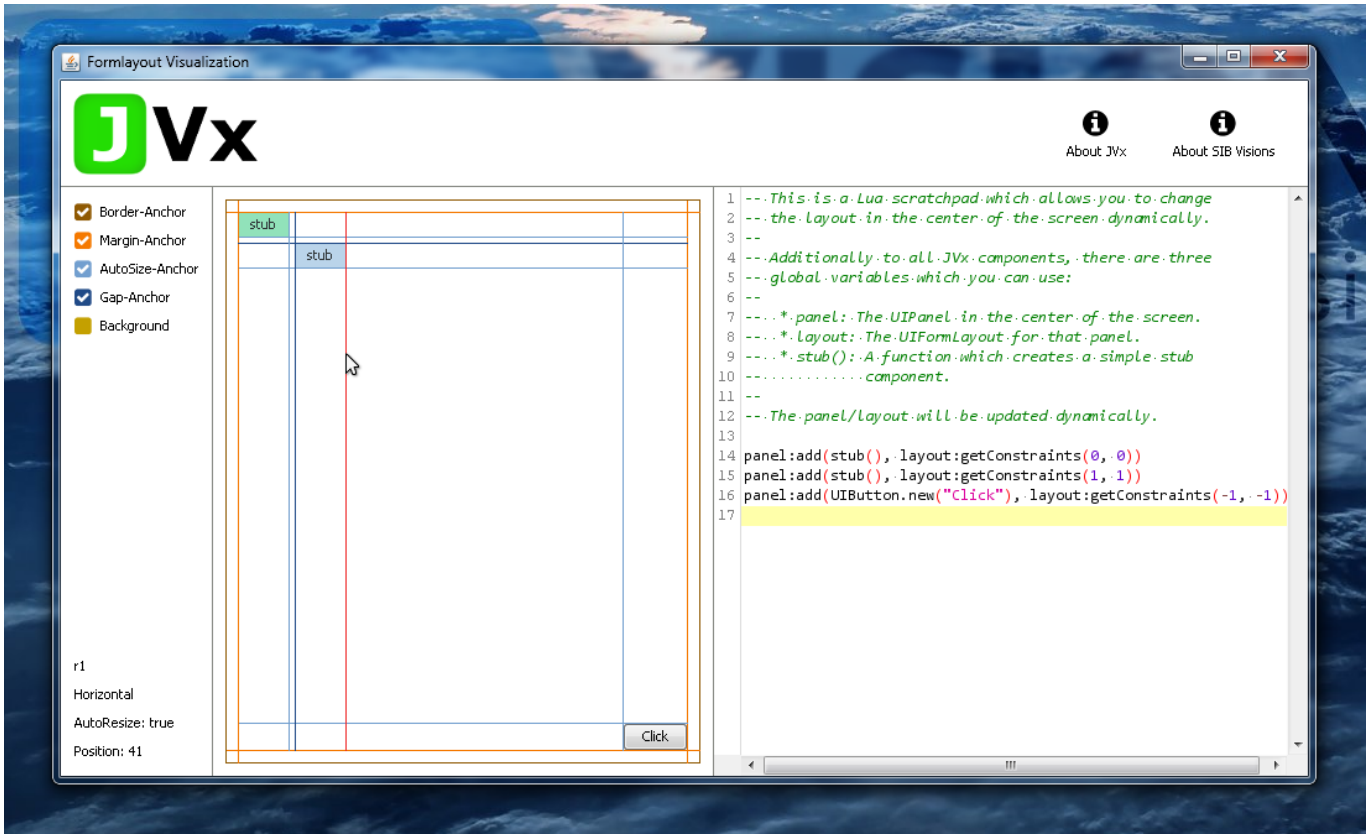
The component is horizontally attached to the left margin anchor and additionally to the autosize

anchor of the second column. The autosize- and gap anchors of the first column are not ignored, but they are not relevant to this case.

At this point it is important to note that spanning and stretched components are disregarded for the preferred size calculation of the layout. Therefore, whenever you span or stretch a component, it is not taken into account when the preferred size of the layout is calculated, which can lead to unexpected results.

Interactive Demo

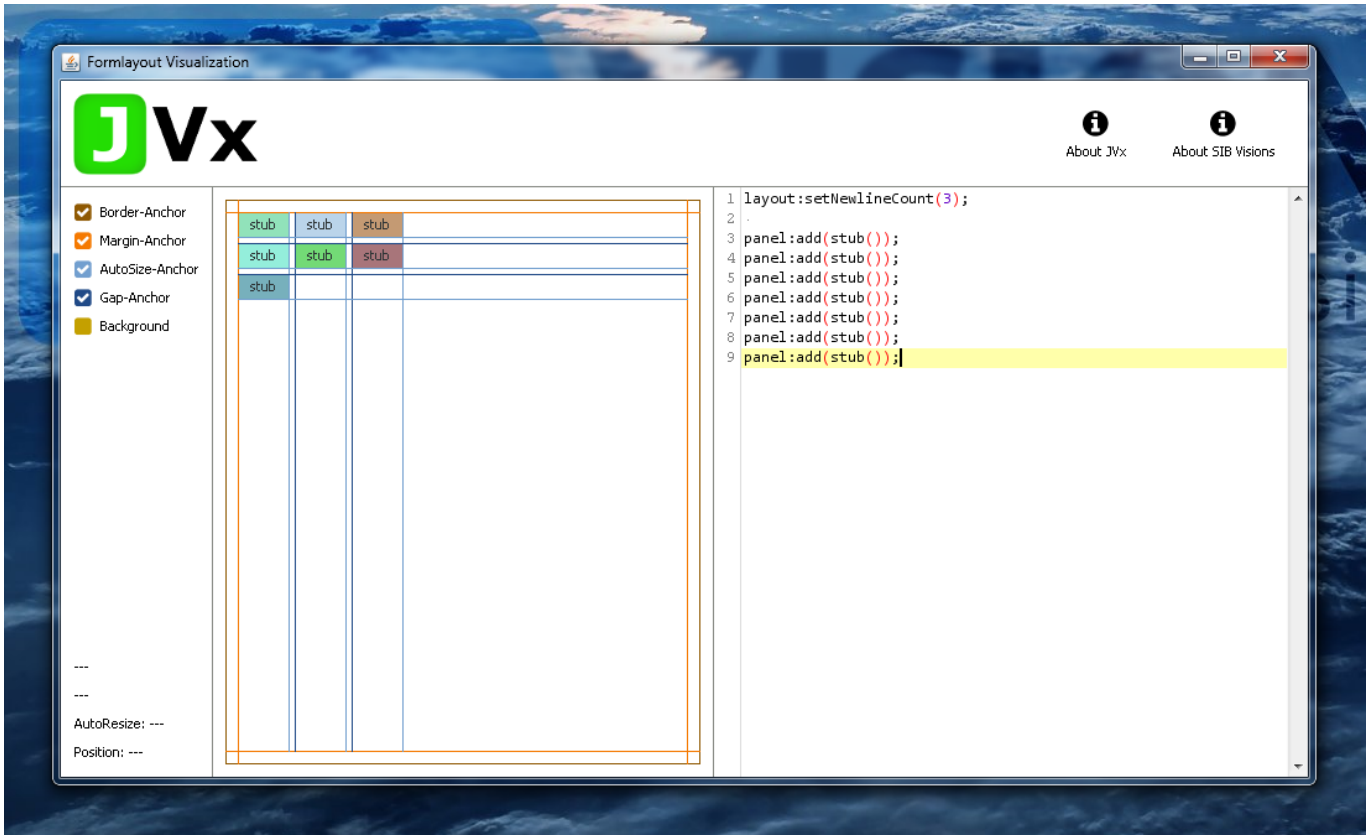
Sometimes, however, it might not be obvious what anchors are created and how they are used. For this, we have created a simple interactive demonstration application that allows you to inspect the created anchors of a layout: the JVx FormLayout Visualization.



On the left is the possibility to show and hide anchors together with the information about the currently highlighted anchor. On the right is an Lua scripting area, which allows you to quickly and easily rebuild and test layouts. It utilizes the JVx-Lua bridge from a previous blog post and so any changes to the code are directly applied.

The Simplest Usage: Flow-Like

Enough of the internals, let’s talk use-cases. The most simple use-case for the FormLayout can be a container which flows its contents in a line until a certain number of items is reach, at which point it breaks into a new line.



It does not require any interaction from us except adding components. In this case, when three components have been added, the next one will be added to the next line and so on. This is quite useful when all you want to do is display components in a fixed horizontal grid.

Java

```
layout.setNewlineCount(3);

panel.add(component);
panel.add(component);
panel.add(component);
panel.add(component);
panel.add(component);
panel.add(component);
panel.add(component);
```

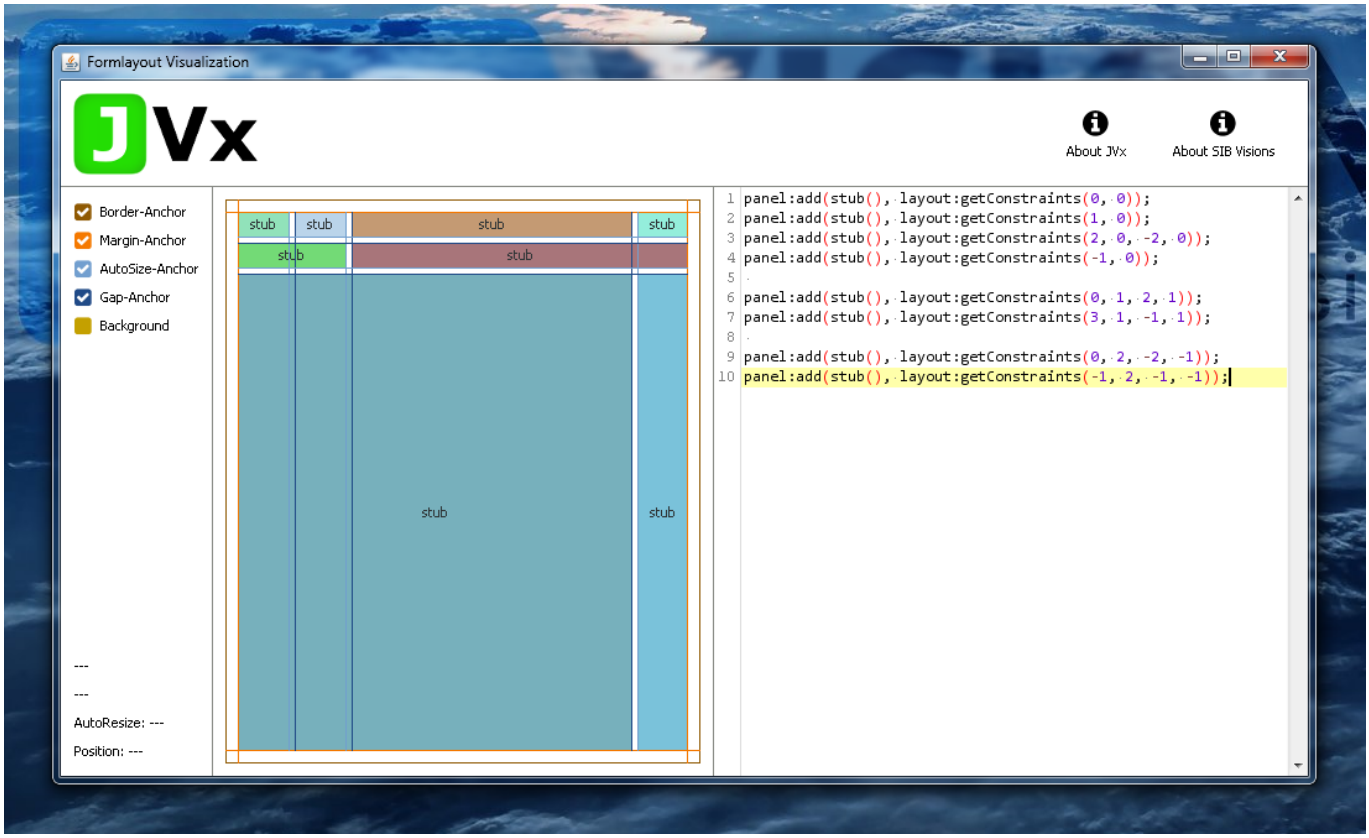
Lua (Demo Application)

```
layout:setNewlineCount(3);

panel:add(stub());
panel:add(stub());
panel:add(stub());
panel:add(stub());
panel:add(stub());
panel:add(stub());
```

```
panel:add(stub());
```

The Most Obvious Usage: Grid-Like

The FormLayout can also be used to align components in a grid and actually layout them in a grid-like fashion.



The main difference is that columns and rows are sized according to the components and not given a fixed-width portion of the panel.

Java

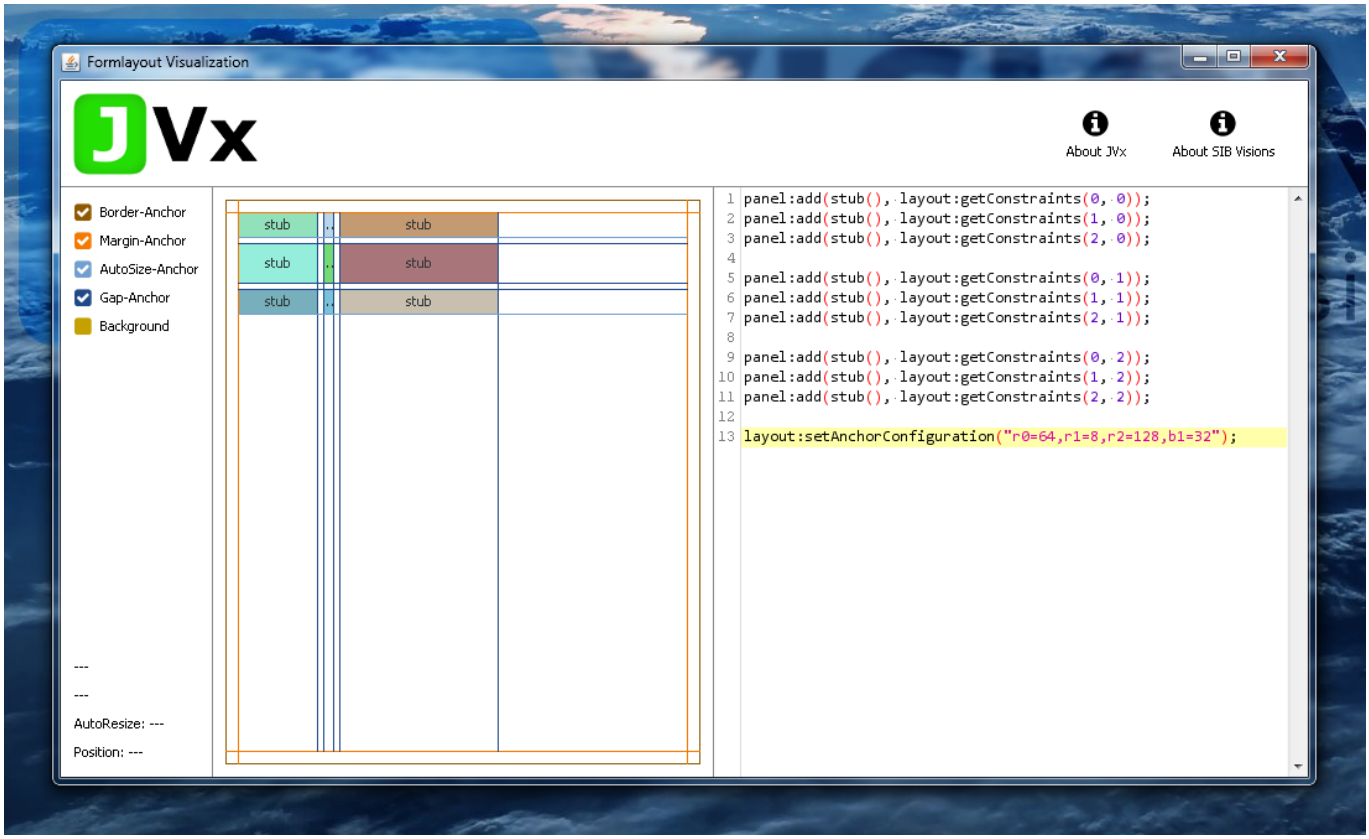
```
panel.add(component, layout.getConstraints(0, 0));
panel.add(component, layout.getConstraints(1, 0));
panel.add(component, layout.getConstraints(2, 0, -2, 0));
panel.add(component, layout.getConstraints(-1, 0));

panel.add(component, layout.getConstraints(0, 1, 2, 1));
panel.add(component, layout.getConstraints(3, 1, -1, 1));

panel.add(component, layout.getConstraints(0, 2, -2, -1));
panel.add(component, layout.getConstraints(-1, 2, -1, -1));
```

Lua (Demo Application)

```
panel:add(stub(), layout:getConstraints(0, 0));
panel:add(stub(), layout:getConstraints(1, 0));
panel:add(stub(), layout:getConstraints(2, 0, -2, 0));
panel:add(stub(), layout:getConstraints(-1, 0));

panel:add(stub(), layout:getConstraints(0, 1, 2, 1));
panel:add(stub(), layout:getConstraints(3, 1, -1, 1));

panel:add(stub(), layout:getConstraints(0, 2, -2, -1));
panel:add(stub(), layout:getConstraints(-1, 2, -1, -1));
```

The More Advanced Usage: Anchor Configuration

Additionally, the FormLayout offers the possibility to manually set the anchor position (e.g., when it is necessary to give the first elements a certain size). Together with the ability to span components, this allows us to create complex and rich layouts.



Java

```
panel.add(component, layout.getConstraints(0, 0));
panel.add(component, layout.getConstraints(1, 0));
panel.add(component, layout.getConstraints(2, 0));
```

```
panel.add(component, layout.getConstraints(0, 1));
panel.add(component, layout.getConstraints(1, 1));
panel.add(component, layout.getConstraints(2, 1));

panel.add(component, layout.getConstraints(0, 2));
panel.add(component, layout.getConstraints(1, 2));
panel.add(component, layout.getConstraints(2, 2));

layout.setAnchorConfiguration("r0=64,r1=8,r2=128,b1=32");
```

Lua (Demo Application)

```
panel:add(stub(), layout:getConstraints(0, 0));
panel:add(stub(), layout:getConstraints(1, 0));
panel:add(stub(), layout:getConstraints(2, 0));

panel:add(stub(), layout:getConstraints(0, 1));
panel:add(stub(), layout:getConstraints(1, 1));
panel:add(stub(), layout:getConstraints(2, 1));

panel:add(stub(), layout:getConstraints(0, 2));
panel:add(stub(), layout:getConstraints(1, 2));
panel:add(stub(), layout:getConstraints(2, 2));

layout:setAnchorConfiguration("r0=64,r1=8,r2=128,b1=32");
```

Conclusion

The JVx FormLayout allows us to quickly and easily create complex, good looking, and functioning layouts that are still flexible enough for the cases when a component is swapped, removed, or added. It can be used in many different circumstances and is still easy enough to use to make sure that even beginners are able to create a basic layout within seconds.

Events

Let’s talk about events and event handling in JVx.

What Are Events…

Events are an important mechanism no matter to what programming language or framework you turn to. They allow us to react on certain actions and “defer” actions until something triggers them. Such triggers can be anything: a certain condition is hit in another thread, the user clicked a button, or another action has finally finished. Long story short, you are notified that something happened and

you can now do something further.

…And Why Do I Need to Handle Them?

Well, you can't skip events, they are a cornerstone of JVx. Theoretically, you could use JVx without any of its events, but you would not only miss out on a lot of functionality, but also be unable to do anything useful. But don't worry, understanding the event system is easy and using it even easier.

Terminology

For JVx the following terminology applies: an event is a property of an object. You can register listeners on that event that will get invoked if the event is dispatched (fired). Every event consists of the `EventHandler`, which allows ypi to register, remove, and manage the listeners and also dispatches the events, meaning it invokes the listeners and notifies them that the event occurred. There is no single underlying listener interface.

Within the JVx framework, every event property of an object starts with the prefix "event" to make it easily searchable and identifiable. But enough dry talk, let's get started.

Attaching Listeners

We will now look at all the ways on how to attach a listener to an event.

Class

The easiest way to get notified of events is to attach a class (that is implementing the listener interface) to an event as listener, like this:

```
public class MainFrame extends UIFrame
{
    public MainFrame()
    {
        super();

        UIButton button = new UIButton("Click me!");
        button.eventAction().addListener(new ActionListener());

        setLayout(new UIBorderLayout());
        add(button, UIBorderLayout.CENTER);
    }
}

private static final class ActionListener implements IActionListener
{
```

```
    public void action(UIActionEvent pActionEvent) throws Throwable
    {
        System.out.println("Button clicked!");
    }
}
```

Inlined Class

Of course, we can inline this listener class:

```
public class MainFrame extends UIFrame
{
    public MainFrame()
    {
        super();

        UIButton button = new UIButton("Click me!");
        button.eventAction().addListener(new IActionListener()
        {
            public void action(UIActionEvent pActionEvent) throws Throwable
            {
                System.out.println("Button clicked!");
            }
        });

        setLayout(new UIBorderLayout());
        add(button, UIBorderLayout.CENTER);
    }
}
```

JVx Style

So far, so good. However, in JVx we have support to attach listeners based on reflection, like this:

```
public class MainFrame extends UIFrame
{
    public MainFrame()
    {
        super();

        UIButton button = new UIButton("Click me!");
        button.eventAction().addListener(this, "doButtonClick");

        setLayout(new UIBorderLayout());
        add(button, UIBorderLayout.CENTER);
    }

    public void doButtonClick(UIActionEvent pActionEvent) throws Throwable
```

```
    {
        System.out.println("Button clicked");
    }
}
```

What is happening here is that, internally, a listener is created that references the given object and the named method. This allows us to easily add and remove listeners from events and keeps the classes clean by allowing us to have all related event listeners in one place without additional class definitions.

Lambdas

Yet there is more. We can, of course, attach lambdas to the events as listeners too:

```
public class MainFrame extends UIFrame
{
    public MainFrame()
    {
        super();

        UIButton button = new UIButton("Click me!");
        button.eventAction().addListener((pActionEvent) ->
System.out.println("Button clicked"));

        setLayout(new UIBorderLayout());
        add(button, UIBorderLayout.CENTER);
    }
}
```

Method References

Lastly, thanks to the new capabilities of Java 1.8, we can also use method references:

```
public class MainFrame extends UIFrame
{
    public MainFrame()
    {
        super();

        UIButton button = new UIButton("Click me!");
        button.eventAction().addListener(this::doButtonClick);

        setLayout(new UIBorderLayout());
        add(button, UIBorderLayout.CENTER);
    }

    private void doButtonClick(UIActionEvent pActionEvent) throws Throwable
    {
```

```
        System.out.println("Button clicked");
    }
}
```

Parameters or No Parameters? To Throw or Not to Throw?

By default we actually support two different classes of listeners, the specified event/listener interface itself and `jvx.rad.util.IRunnable`. This means you can also attach methods that do not have any parameters, like this:

```
public class MainFrame extends UIFrame
{
    public MainFrame()
    {
        super();

        UIButton button = new UIButton("Click me!");
        button.eventAction().addListener(this::doButtonClickNoParameters);
        button.eventAction().addListener(this::doButtonClickWithParameters);

        setLayout(new UIBorderLayout());
        add(button, UIBorderLayout.CENTER);
    }

    private void doButtonClickNoParameters() throws Throwable
    {
        System.out.println("Button clicked");
    }

    private void doButtonClickWithParameters(UIActionEvent pActionEvent)
throws Throwable
    {
        System.out.println("Button clicked");
    }
}
```

Additionally, all listeners and `IRunnable` itself support throwing `Throwable`, which is then handled inside the `EventHandler`. As you can see, you are very flexible when it comes to what methods you can attach and use as listeners.

Creating Your Own Events

You can, of course, create your own `EventHandlers` and listeners to create your own events. All you need are two classes, an extension of EventHandler and a listener interface.

```
public class CustomEvent extends EventHandler
{
```

```
    public CustomEvent()
    {
        super(ICustomListener.class);
    }
}

public interface ICustomListener
{
    public void somethingHappened(String pName);
}
```

And that's it, from here on you can use it:

```
CustomEvent event = new CustomEvent();
event.addListener((pName) -> System.out.println(pName + " 1"));
event.addListener((pName) -> System.out.println(pName + " 2"));
event.addListener((pName) -> System.out.println(pName + " 3"));

event.dispatchEvent("Adam");
```

Additional Methods

You can also use an interface for listeners that has multiple methods, specifying in the constructor which method to invoke:

```
public class CustomEvent extends EventHandler
{
    public CustomEvent()
    {
        super(ICustomListener.class, "somethingOtherHappened");
    }
}

public interface ICustomListener
{
    public void somethingHappened(String pName);
    public void somethingOtherHappened(String pName, BigDecimal pValue);
    public void nothingHappened();
}
```

Now every time the event is dispatched, the `somethingOtherHappened` method will be invoked. However, don't use this. The upside of having a "simple" listener interface with just one method is that it allows us to use lambdas with it. A listener interface with multiple methods won't allow this.

In JVx we reduced our listener interfaces to just one method (in a backward compatible way) to make sure all events can be used with lambdas.

Fire Away!

That’s it for this short reference sheet. This is how our event system can and should be used. Of course, there is much more to it under the hood (for example, listeners being wrapped in proxy classes, reflection used for invoking methods, etc.). If you feel adventurous, be my guest and have a good look at the internals of `EventHandler`. It is quite an interesting read!