

Creating a custom control using Java

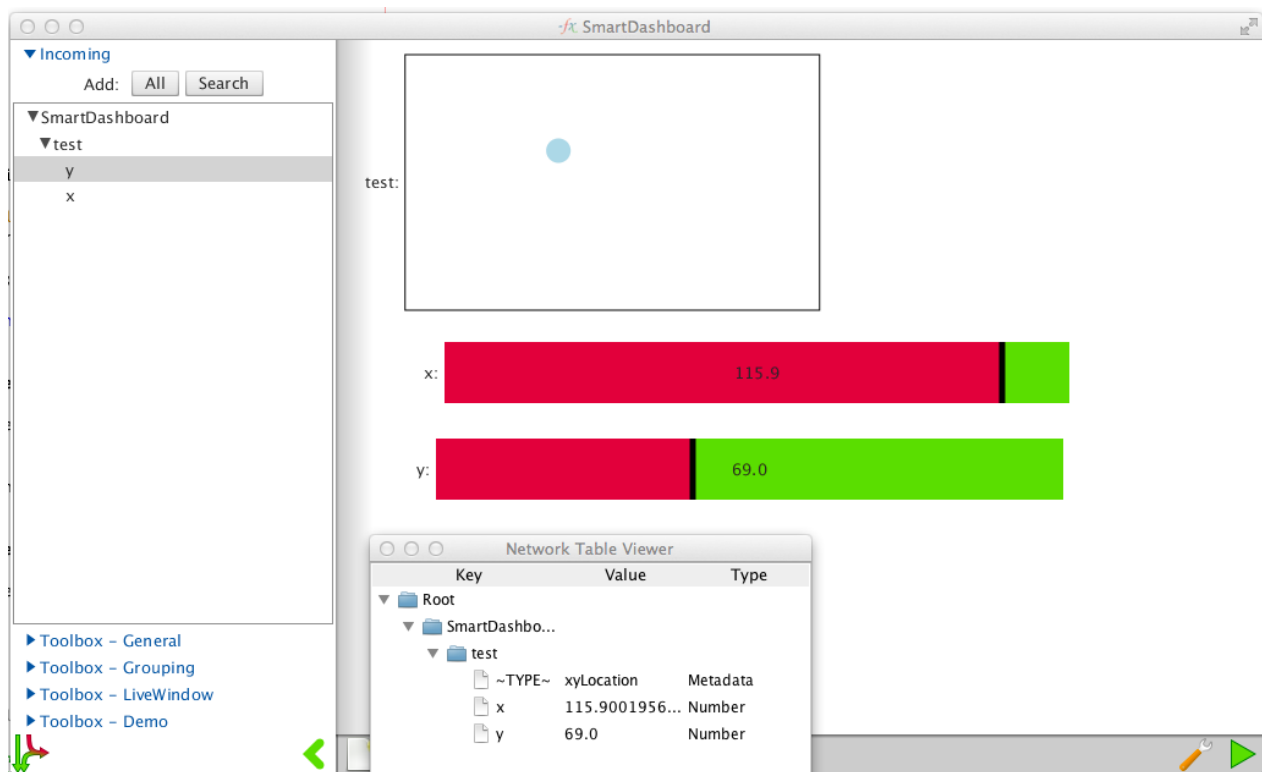
Creating a custom control using Java

sfx comes with a palette of built-in controls that feature a wide range of use cases. But sometimes you would like to further customize your robot dashboard with controls that you create yourself. There are two strategies for creating custom controls, either:

1. FXML - a XML-based markup language for describing your own controls using a declarative language without needing programming
2. Java-based controls can have more complex requirements and behaviors

In this lesson we'll look at creating Java-based controls. For FXML controls see the [FXML tutorial](#).

Creating a X-Y location indicator using Java

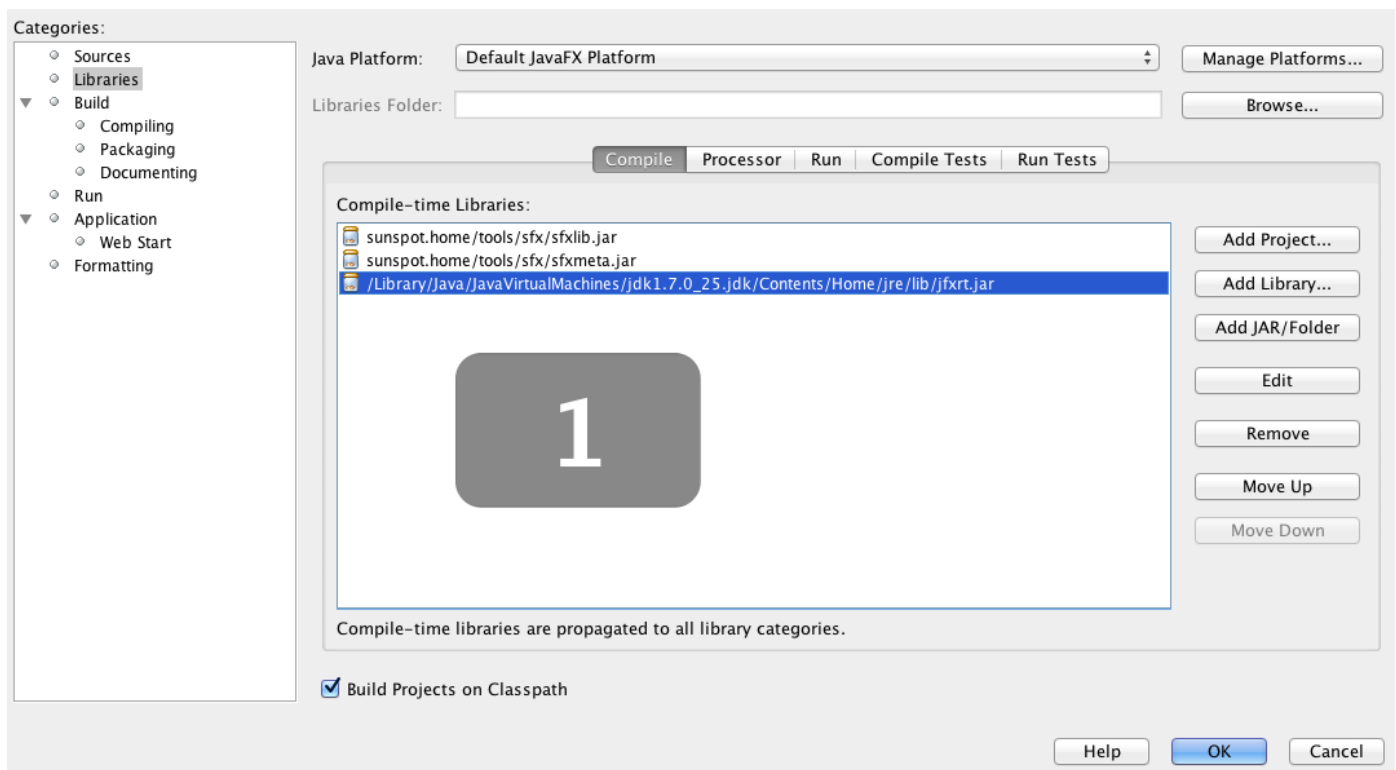


Suppose you need to display some object in 2D space, like a vision target from the camera, robot position on the field, or Joystick position so that field operators can easily see the location. As this has multiple variables, it is much easier to do this with Java-based controls.

Creating a custom control using Java

In this simple example, we will be adding an [Ellipse](#) to a data-enabled [AnchorPane](#) and moving it based on an object with x and y properties.

Create a Netbeans project



In Netbeans, create a new Java Class Library. Once you create the project, right click it and go to properties. Inside the properties, select "Libraries" and add sfxlib.jar and sfxmeta.jar (they will appear in the same directory as sfx.jar after one run). Also add jfxrt.jar, which is system dependent, but is normally found in \$JAVA_HOME/jre/lib/jfxrt.jar

Adding the Control class

Now add a Java source file for your new class (this example will call it xyLocation.java in package com.example). For our example we will extend DataAnchorPane as it is both data-enabled and supports positioning children via x and y coordinates. Since Java classes can have annotations, we can place what FXML files require in manifests in annotations only. Add the following annotation to the class:

```
@Category("Tutorial")
```

Creating a custom control using Java

This marks the class as being in the toolbox category "Tutorial"

```
@Designable(value = "X-Y Location", description = "A control to show x/y position in a range")
```

This describes this class as being designable in SFX, showing it with the given name and description

```
@GroupType("xyLocation")
```

This says that the control designs all groups of type xyLocation. This is implemented in NetworkTables by giving a table a sub-key of ~TYPE~ with value xyLocation

```
@DashFXProperties("Sealed: true, Save Children: false")
```

This adds any arbitrary manifest attributes to the class. These say to treat this as an atomic object, even though we are extending a pane that supports designable children.

As we are extending from a data-enabled class (DataAnchorPane), we can simply call `getObservable()` on ourselves and not worry too much about it. As we are displaying an object, we do need to enable the default name-prepend via `setDataMode(DataPaneMode.Nested)`. This makes all calls to `getObservable("x")` to retrieve the values under `this.getName() + "/x"` instead of just "x".

All controls are given a data source to follow when they are registered with the DataCore, which then calls `registered()`. We need to override this so we can get our keys from the provider at this time.

```
@Override
public void registered(DataCoreProvider provider)
{
    super.registered(provider);
    unwatch();
    // if we are being registered, then we can finally get the x and y variable
    // otherwise just unwatch as we are being unregistered
    if (provider != null)
    {
        xValue = getObservable("x");
    }
}
```

Creating a custom control using Java

```
        yValue = getObservable("y");
        rewatch();
    }
}
```

In order to enable more complex actions later, we will add listeners to the SmartValues

```
private void rewatch()
{
    xValue.addListener(xchange);
    yValue.addListener(ychange);
}
private void unwatch()
{
    // this function un-binds all the variable
    if (xValue != null)
        xValue.removeListener(xchange);
    if (yValue != null)
        yValue.removeListener(ychange);
}
```

xchange and ychange are defined as follows but can easily be extended for multiple other features and/or calculations

```
private ChangeListener ychange = new ChangeListener<Object>() {
    @Override
    public void changed(ObservableValue<? extends Object> ov, Object
t, Object t1)
    {
        ellipse.setCenterY(yValue.getData().asNumber() + 10); //
offset by radius
    }
},
xchange = new ChangeListener<Object>() {
    @Override
    public void changed(ObservableValue<? extends Object> ov, Object
t, Object t1)
    {
```

Creating a custom control using Java

```
        ellipse.setCenterX(xValue.getData().asNumber() + 10); //
offset by radius
    }
};
```

This simply directly sets the position from the values with a constant offset of the radius. Note that this does not scale nor have any limits, so the ellipse can move off the canvas. The units are JavaFX DPI-independent pixels (roughly 1 px at normal dpi with no transforms)

The ellipse is very simple and defined as such:

```
// we are displaying results by moving the ellipse. initialize it here
ellipse = new Ellipse(10, 10, 10, 10);
ellipse.setFill(Color.LIGHTBLUE);
this.getChildren().add(ellipse); // we inherited from DAP so just add it
to ourselves
```

Note that currently there is a small bug in DataPane in that nested mode does not update the correct keys when the name changes. As such, it is currently required to re-bind on each name change, however will not be once this bug is fixed

```
nameProperty().addListener(new ChangeListener<String>()
{
    @Override
    public void changed(ObservableValue<? extends String> ov, String
t, String t1)
    {
        unwatch();
        try
        {
            xValue = getObservable("x");
            yValue = getObservable("y");
            rewatch();
        }
        catch(NullPointerException n)
        {
            //fail, ignore, as we must not be registered yet
        }
    }
});
```

Creating a custom control using Java

```
    }  
  });
```

Creating the manifest to register the control with SFX

To add your control to the dashboard, you need to package it in a plugin. For FXML controls you can simply put it in a folder or pack it in a jar. Either way, we need a manifest file that describes what the plugin contains. Manifests are written in YAML and can contain multiple controls and other options. For our needs, we will start with this file (please generate your own UUID. uuidgenerator.net is where the provided UUID was generated)

```
API: 0.1  
Name: Tutorial plugin  
Description: Contains all the plugins from the tutorial  
Version: 1.0.0  
# Please generate your own unique UUID and replace it below  
Plugin ID: b673b0fe-716a-40ce-b446-70e34aafc509  
Controls:  
-  
  Class: com.example.xyLocation
```

This says:

- we are using plugin API version 0.1 (the current version)
- the name of the plugin is "Tutorial plugin" which can be identified by the UUID
- it has one control, the xyLocation control, which has more information in its annotations

Save the manifest as manifest.yml in the root of the src/ folder. NOTE: YAML expects spaces for indentation, not tabs.

Now build the project with netbeans, and copy the jar to sfx/plugins/. Now when you start sfx, you should be able to use the control. It is recommended to launch from the terminal with the command:

```
java -jar sfx.jar
```

in case there are any errors. All plugins can be viewed under settings>plugins.