NIFTY GUI 1.3.2

The Missing Manual

Book Credits

Authors

Jens Hohmuth (void)

Martin Karing (mkaring)
(Slick2D chapter)

Wesley Shillingford (wezrule)
(Grammar and spelling changes)

Documentversion

Version	Release	Author	Changes
1.0	28.12.2011	Jens Hohmuth (void)	Initial Version
1.1	12.11.2012	Jens Hohmuth (void)	Update for Nifty 1.3.2

INTRODUCTION

Nifty GUI is a Java library to create interactive user interfaces. It is well integrated into many existing rendering systems (JME3, JME2, LWJGL, JOGL, Slick2D and even Java2D). If necessary it can be easily integrated into other rendering systems by implementing a simple Service Provider Interface (SPI).

The actual GUI is stored in XML files (using a custom XSD) or it can be created directly from Java. Java is used to respond to events generated by the GUI and to modify the GUI to reflect changes in the state of your application, changing a text label for example. Additionally there is a large set of effects available that can be used to modify the appearance of the GUI. Effects add the "nifty" part to Nifty GUI :)

Besides many standard controls like buttons, textfields, scrollbars and so on Nifty provides a lot of freedom and it can be used to create in-game HUD like displays as well. GUIs written with Nifty can be more visual stunning and exciting of what you'd usually expect from a Java GUI system.

Here are some screenshots from the Nifty example projects:

This manual will give you an in-depth view on how Nifty works and how you can use it in your own applications.

BASICS

REQUIRED FILES

Since Nifty can be integrated into several different rendering systems there exists quite a number of adapter jars besides the Nifty core module (one for each rendering system). Usually you won't need all of them and only the one for the rendering system you'd like to use.

At the very minimum Nifty consists of at least two Jar files:

1. The Nifty core module: **nifty-<version>.jar**
2. A Nifty rendering system adapter: **nifty-<system>-<version>.jar**

The following table lists all of the available renderer Jar files for Nifty.

System	jar File
Nifty LWJGL	nifty-lwjgl-renderer-1.3.jar
Nifty JME3	(Integrated into JME3)
Nifty JME2	not released yet for Nifty 1.3
Nifty Slick2D	nifty-slick-renderer-1.3.jar
Nifty JOGL	not released yet for Nifty 1.3
Nifty Java2D	not released yet for Nifty 1.3

With the Nifty Core Jar and a renderer Jar you can already create and use Nifty. But there are additional Jars available.

ADDITIONAL FILES

There is a separate Jar that contains the Nifty standard controls („nifty-default-controls-<version>.jar"). This Jar provides standard GUI components like Button, Checkbox, ListBox and so on.

Everything that this jar provides is based on the Nifty Core module. If you don't plan to use the standard controls then this project can still be useful as a demonstration on how you can combine the basic mechanism that Nifty provides into more complex controls.

The controls project is meant to be used together with an accompanying style file. We will get into Nifty styles later in this book. For the moment you can see the „nifty-style-black-<version>.jar" as the specification on how the controls will look like. To use the Nifty standard controls you'll need to add both, the „nifty-default-controls-<version>.jar" and the „nifty-style-black-<version>.jar" to your Java classpath.

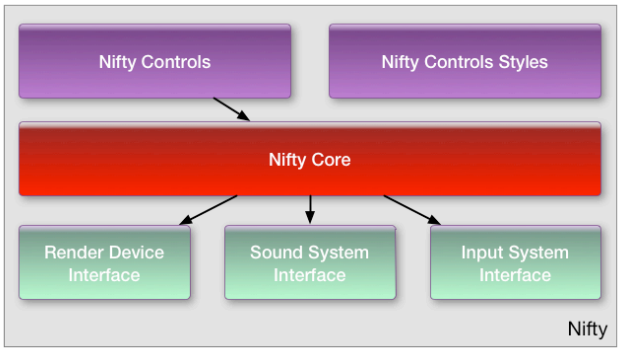
Last but not least Nifty supports sound output for playing background music or sound effects. For sound there are two additional Jar files available that use OpenAL (using LWJGL) „nifty-openal-soundsystem-<version>.jar" or Pauls Soundsystem „nifty-pauls-soundsystem-<version>.jar" for sound output.

The following table lists all of the available additional Jars for reference.

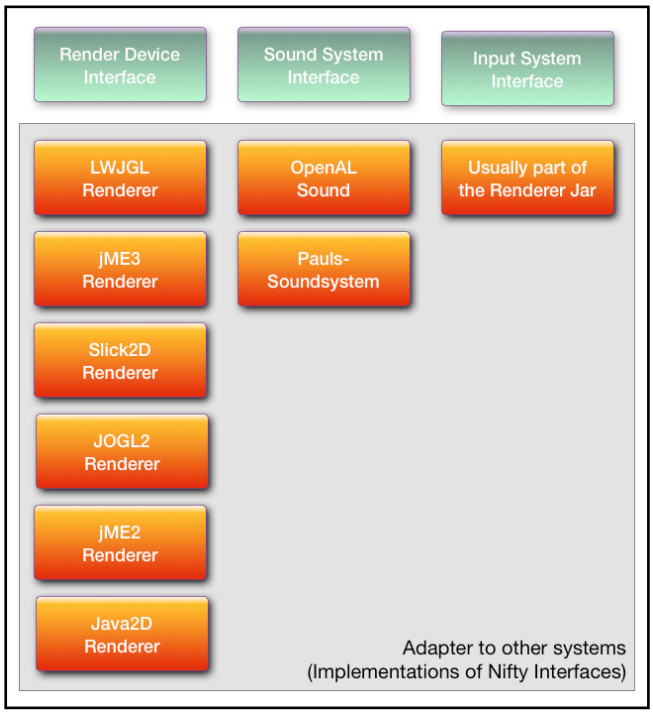
Name	Jar File
Nifty Standard Controls	nifty-default-controls-<version>.jar
Nifty Standard Controls Style	nifty-style-black-<version>.jar
Nifty OpenAL Soundsystem (LWJGL)	nifty-openal-soundsystem-<version>.jar
Nifty Pauls Soundsystem	nifty-pauls-soundsystem-<version>.jar

NIFTY SERVICE PROVIDER INTERFACE (SPI)

Nifty provides a couple of Java Interfaces that you can implement to make Nifty use whatever rendering system you want to use. This is called a Service Provider Interface (SPI). The SPI for Nifty consists of RenderDevice-, SoundSystem- and InputSystem-Java Interfaces. Here is a schematic view of all the individual components involved in Nifty.



The InputSystem is usually implemented together with a RenderDevice implementation. The next image shows all of the already available implementations of the Nifty SPI.



INITIALIZE NIFTY

To access Nifty and use it you'll first need to instantiate the de.lessvoid.nifty.Nifty class. To do so you'll need to call the constructor which looks like this:

```
  public Nifty(
      final RenderDevice newRenderDevice,
      final SoundDevice newSoundDevice,
      final InputSystem newInputSystem,
      final TimeProvider newTimeProvider);
```

As you can see the Nifty constructor requires instances of the three subsystem implementations of the SPI one for the RenderDevice, the SoundDevice and the InputSystem. Additionally it requires a de.lessvoid.nifty.tools.TimeProvider instance. The TimeProvider is just a simple class for accessing the current system time without scattering new Date() calls all over the system.

Currently most Nifty renderer Jars provide implementations for all three subsystems because they often relate to each other. Usually all you need is the „nifty-<system>-renderer-<version>.jar" for your rendering system in the Java classpath to access implementations for all subsystems.

Additionally Nifty provides Null implementations for all three Interfaces in the de.lessvoid.nifty.nulldevice package that you can use if you don't require an implementation for one of the subsystems. So for instance if you don't need any sound output in your GUI you can just use de.lessvoid.nifty.nulldevice.NullSoundDevice as the SoundDevice parameter when constructing the Nifty instance and Nifty will not output any sound.

EXAMPLE

Here is an example of creating Nifty using LWJGL and no sound output support.

```
LwjglInputSystem inputSystem = new LwjglInputSystem();
inputSystem.startup();

Nifty nifty = new Nifty(
  new LwjglRenderDevice(),
  new NullSoundDevice(),
  inputSystem,
  new TimeProvider());
```

Please note that most Nifty RenderDevice implementations assume that you have already initialized the underlying rendering system before you create Nifty using the constructor. In the example using LWJGL you'll need to initialize LWJGL before you can create the Nifty instance.

The reason is that Nifty is probably not the only part of your system that needs to render things. So Nifty lets you decide when and how you setup your rendering system and it does not try to overtake your whole system.

We'll take a look at rendering and updating Nifty next.

RENDER AND UPDATE

There are only two calls to Nifty necessary that you'll need to call regularly.

One of them is nifty.render() which will render the GUI in its current state on the screen. There is a catch however. Nifty assumes that your rendering system is in a state that is appropriate for 2d rendering and it is up to you to set it up. So in case of using LWJGL you'll need to enable 2d ortho mode prior to calling nifty.render().

Nifty.render() takes a boolean as its only parameter. You set this parameter to true if you like to clear the screen before rendering Nifty or you can set it to false if Nifty should draw the GUI without clearing the screen because maybe you've already did this on your own.

```
// get Nifty Version and output it to system.out
String niftyVersion = nifty.getVersion();
System.out.println(niftyVersion);
```

The second method you'll need to call is nifty.update(). This call will process input events and update the internal GUI state. The method will return true if Nifty reaches a state that should end the GUI processing or false if the GUI is still active and should be kept updating and rendering.

In case of using LWJGL calling Display.update() is still up to you. Here is some pseudocode for the render loop using Nifty when using LWJGL.

```
// render and update Nifty
boolean done = false;
while (!done) {
  // update Nifty
  if (nifty.update()) {
    done = true;
  }

  // render Nifty
  nifty.render(true);

  // render other stuff, call LWJGL Display.update() and so on
}
```

ELEMENTS INTRODUCTION

At it's core Nifty only supports a handful of elements. Nifty can display Text and Image elements as well as Panels, which are just rectangular areas on the screen that can optionally be visible. Usually Nifty Panels are invisible and are only used as containers for other elements to help in layout.

These three basic elements are organized or grouped into so called layers and one or more layers are grouped into a screen. You can think of a Nifty screen as a form of reference for a certain state of your GUI. There is a whole chapter dedicated to the elements and all of the attributes they provide. For now it is only important to understand that Nifty really is only about the Panel, Text and Image elements which you can position, display and interact with (click them, move them around, change them and so on).

All of the basic elements can be combined into a Nifty control. You can see this as a form of container for the basic elements and the combination of the elements can be used exactly like the

basic elements. A control can simply be a form of a template. If you need the exact same combination of elements multiple times then you can simply group them, give them a name and then reuse this combination multiple times.

Another way to see controls are the provided standard controls. There is a button control available for instance that is a combination of a panel and a text but can be simply seen and used as a single control, the „button“. There is a dedicated chapter for controls as well.

So to summarize Nifty is about the display and management of elements, where a element can be one of the build-in elements (Panel, Text, Image) or it is a combination of these build-in elements in the form of a control.

GET NIFTY VERSION STRING (NIFTY 1.3.2)

Starting with Nifty 1.3.2 the main Nifty instance has a new new `getVersion()` method that returns the version of Nifty and the time of the Nifty build.

The result is a String like: `„1.3.2 (2012-10-08 00:09:03)“`.

GUI DEFINITION

XML GUI

INTRODUCTION

One way to define all the elements that make up your GUI is to use XML-Files. This is especially useful to modify your GUI without the need to recompile any Java files. You just use the same code and change only some XML files if you need to modify the GUI.

Nifty uses XML-Schema (XSD) to define what elements and attributes are possible. This way you can use XML tools that can read the XSD information to enable things like auto completion and syntax checks when writing XML files. Using XML and XSD allows third party tools like the Nifty GUI editor in the jMonkeyEngine SDK project to parse and „understand“ the GUI definition and support you even more when designing your GUIs. However please note that currently not all possible attributes are supported or constrained in the Nifty-XSD.

The correct XML namespace for the Nifty XSD is „http://nifty-gui.sourceforge.net/nifty-1.3.xsd“ and you can download the current XSD by using the namespace URL as well.

A valid Nifty XML file looks like the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty xmlns="http://nifty-gui.sourceforge.net/nifty-1.3.xsd" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://nifty-
gui.sourceforge.net/nifty-1.3.xsd http://nifty-gui.sourceforge.net/nifty-1.3.xsd">

  <!-- Nifty XML content goes in here -->

</nifty>
```

It specifies the namespace „xmlns“ attribute as well as the „schemaLocation“ for XML tools that support the „schemaLocation“ attribute.

The next chapter will explain in detail what this „Nifty XML content“ is and how it works. For now it‘s just important to understand that the XML file will define everything that your GUI needs to display. How you can tell Nifty to actually load the XML file(s) is explained in the next section.

LOADING XML

To load a XML file you can use one of the fromXml() Methods the Nifty instance provides. There are methods available to load a file directly from the filesystem using a filename or from an InputStream. The methods allow you to specify a „screenId“ of the screen that should be started after the XML has been loaded. You can find more informations about the concepts of a Nifty screen in the next chapter.

Here are the standard methods to load a Nifty XML file.

```
// load Nifty XML file from a file or an InputStream
public void fromXml(String filename, String startScreen);
public void fromXml(String fileId, InputStream input, String startScreen);
```

When you use the method that takes an InputStream as a parameter you'll need to specify a „fileId" for the InputStream. The „fileId" is used to identify the loaded XML file in case Nifty needs to decide if a given file has already been loaded. When using the filename method the filename itself acts as the „fileId".

Sometimes it is necessary to load a XML file but without starting a screen. There are two other methods available to just load a Nifty XML file. Again, you can find more informations about the concepts of a screen in the next chapter.

```
// only load file or InputStream but don't start any screen
public void fromXmlWithoutStartScreen(String filename);
public void fromXmlWithoutStartScreen(String fileId, InputStream input);
```

As you can see from the method signature the only difference is the missing „startScreen" parameter.

There is an additional set of methods available that allow you to specify the ScreenControllers to load. The next chapter will explain what a ScreenController is and why you might want to specify them when loading a XML file.

```
// load from a file or InputStream with ScreenController instances
public void fromXml(String filename, String startScreen,
                    ScreenController ... controllers);

public void fromXml(String fileId, InputStream input, String startScreen,
                    ScreenController ... controllers);
```

All of these methods will remove any previously loaded screens and replace everything loaded with the data from the new XML file. This means that everything you would like to display must be defined in a single XML file.

If you have many screens or you want to keep them organized in separate files there are two „addXml" methods available that will just load an additional XML file. The content of the files are simply added to whatever XML data has been loaded before.

```
// add the content of an XML file to the loaded data
public void addXml(String filename);
public void addXml(InputStream stream);
```

VALIDATING XML

Nifty supports validating of XML-Files using the XSD. This way you can ensure that a given XML file is valid and does not contain any syntax errors.

XML validation is an optional step which means that all of the loadXml() and addXml() methods don't check the XML. You'll need to call a special validateXml() method to check XMLs. This is because validating XML files takes some time and if you are sure your XML files are valid (because you've written them using an XML editor that already validated the XML) validating them again would just be a waste of time.

So if you're unsure if your XML is valid you can call validateXml() which looks like this:

```
public void validateXml(String filename) throws Exception;
public void validateXml(InputStream stream) throws Exception;
```

Both methods will simply return when the XML is valid or they will throw an Exception if something is wrong with the XML. This check is always performed in respect to the XSD. The Exception will point out what is wrong.

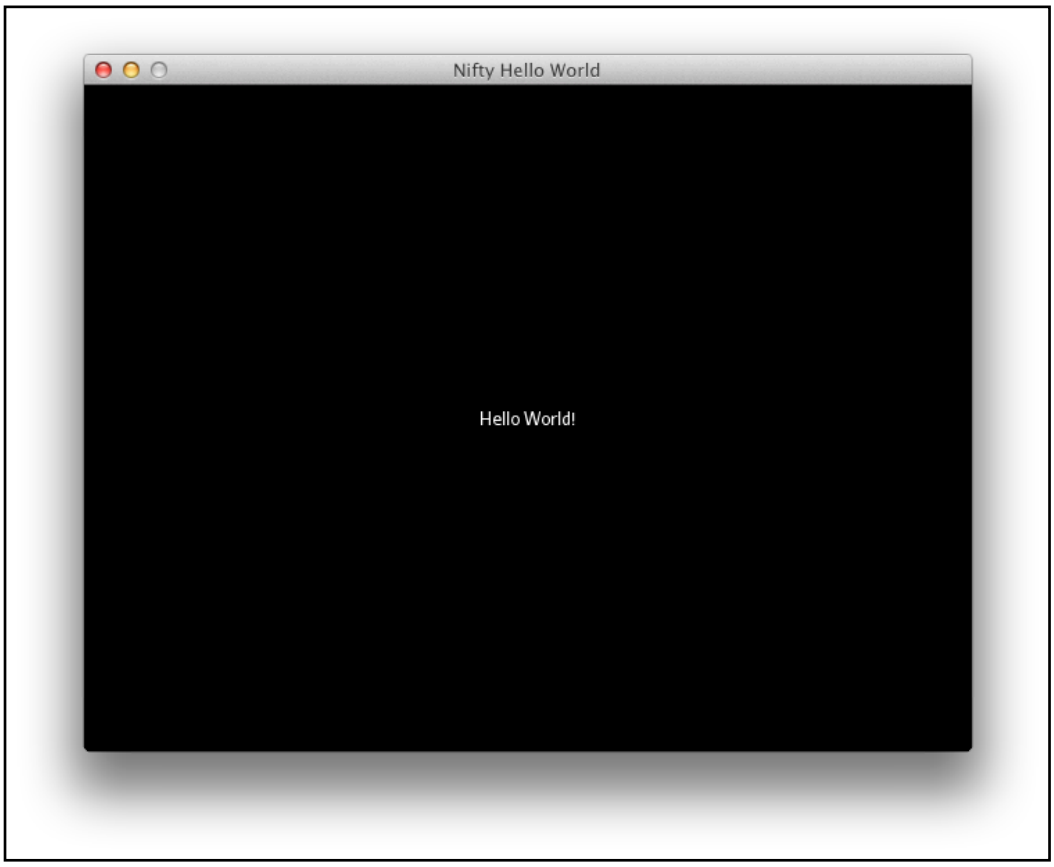
And now that you know how to load and validate XML files we'll continue with a complete Nifty XML example!

EXAMPLE

The following XML is a minimal Nifty XML file to display „Hello World“ in the middle of the screen. Again the details of what <screen>, <layer> and <text> mean is being explained in detail in the next chapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty xmlns="http://nifty-gui.sourceforge.net/nifty-1.3.xsd" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://nifty-
gui.sourceforge.net/nifty-1.3.xsd http://nifty-gui.sourceforge.net/nifty-1.3.xsd">
  <screen id="start">
    <layer childLayout="center">
      <text font="aurulent-sans-16.fnt" color="#ffff"
            text="Hello World!" />
    </layer>
  </screen>
</nifty>
```

And as the result we get a black background and a „Hello World!“ text label in the middle of the screen:



Not too bad :)

SPECIAL XML MARKUP

Every attribute of every XML element can contain the special markup „${...}“ that gets replaced with something else when the XML is loaded. The following values are supported with the „${...}“ syntax:

${id.key}
Lookup resource bundle with "id" and request "key" from it. This is explained in more detail below in the Localization section.

${ENV.key}
Lookup "key" in all of the environment variables (System.getEnv()) and replace „${ENV.key}“ with the value of the environment variable „key“.

${PROP.key}
Lookup "key" in the Nifty.setGlobalProperties(Properties) properties or if the properties are not set use System.getProperties() to lookup "key".

${CALL.method()}
Call method() at the current ScreenController and replace the value that the method() returns. When used in this way then „method()“ should return a String.

Here is an example. When we change the text in Hello Word example like so.

```
<text font="aurulent-sans-16.fnt" color="#ffff"
      text="your home directory: ${ENV.HOME}" />
```

Then „${ENV.HOME}“ will be replaced by the content of your $HOME environment variable!

If the replacement could not be performed successfully then nothing is being replaced and you'll get the original „${…}“ String back.

LOCALIZATION

Nifty localization is using standard property file based Java Resourcebundles. This simply means that you'll need to create a property file containing keys that are referenced from Nifty XML using the current locale settings of the VM.

Let's suppose you have the following files:

dialog.properties:
hello = Hello World in Default Language

dialog_de.properties:
hello = Hallo Welt in Deutsch

dialog_en.properties:
hello = hello world in english

Once you have created these files you'll need to tell Nifty where it can find them. You'll do that with the <resourceBundle> XML tag. You'll need to give the resourceBundle a name using the id property so that we can later reference this specific resourceBundle (you can have multiple different ones).

```
<resourceBundle id="dialog" filename="src/main/resources/dialog" />
```

Now that Nifty knows about your ResourceBundle you can access it with the „${id.key}“ XML markup. Here is an example to access the „hello“ key in the „dialog“ ResourceBundle we have just registered using the <resourceBundle> tag.

```
<text font="aurulent-sans-16.fnt" color="#ffff" text="${dialog.hello}" />
```

Now Nifty will use the current default locale to access the ResourceBundle with the id "dialog" and looks up the value for "hello".

If for some reason you don't want Nifty to use the default Locale you can force a specific one with the "nifty.setLocale(Locale)" method.

JAVA GUI

INTRODUCTION

XML is not the only way you can use to define Nifty GUIs. It is possible to create elements directly from Java. This is necessary when you need to create elements at runtime or when you don‘t want to be dependent on XML files at all. Everything you can do with XML is possible with Java as well.

Nifty offers two slightly different mechanism to create elements from Java and this chapter will explain both ways. What way you use is up to you in the end.

JAVA CREATOR CLASSES

This is the old way of creating elements in Nifty. For every standard element there exists a *Creator class that has simple getter and setter methods to set the attributes of the element. To actually create a new element you call the create method of the *Creator classes.

EXAMPLE

Here is an example to create a new panel in the layer with the id „baseLayer“.

To create a new element Nifty needs the Nifty instance, the screen and the parent element of the new element. The new element will be added as a new child element to the given parent element.

For this example we assume that you have the following Nifty XML and that you want to create a new panel inside the empty „baseLayer“ layer.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty xmlns="http://nifty-gui.sourceforge.net/nifty-1.3.xsd" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://nifty-
gui.sourceforge.net/nifty-1.3.xsd http://nifty-gui.sourceforge.net/nifty-1.3.xsd">
  <screen id="start">
    <layer id="baseLayer" childLayout="center">
      <!-- this layer is empty and populated from Java -->
    </layer>
  </screen>
</nifty>
```

So there is this empty layer with id=“baseLayer“. To actually create a new element inside of that layer, we‘ll first need the screen instance and the layer element. We can get both from the Nifty instance.

Please note that there is a dedicated chapter „Runtime Element Modification“ that explains how to access the screen, elements and a lot more in detail.

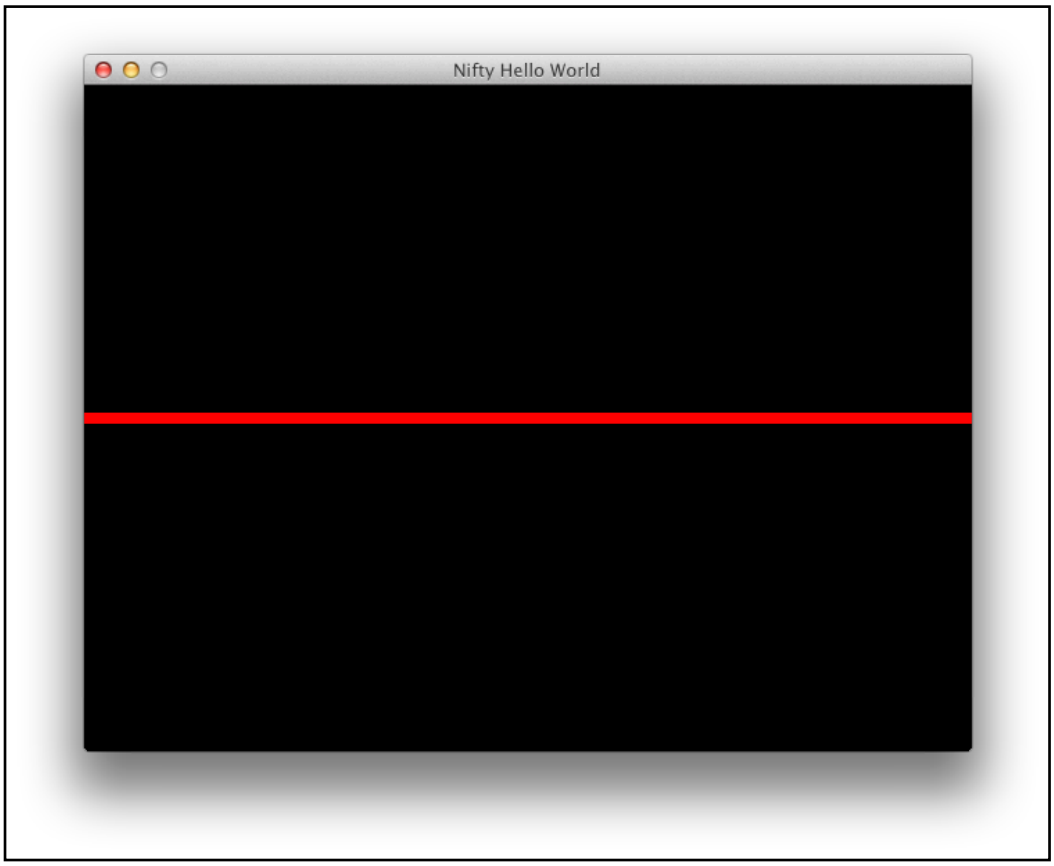
So this is the code to get the screen and the layer element:

```
Screen screen = nifty.getCurrentScreen();
Element layer = screen.findElementByName("baseLayer");
```

When we have both we can finally create the new panel using a PanelCreator instance:

```
// create a 8px height red panel
PanelCreator createPanel = new PanelCreator();
createPanel.setHeight("8px");
createPanel.setBackgroundColor("#f00f");
Element newPanel = createPanel.create(nifty, screen, layer);
```

And Nifty will create the element and we end up with this as the result:



Please note that the create() method returns the new element. This can be used as the parent element of other *Creator calls. This way you can build a whole screen with all layers and elements if necessary.

You can find all build-in *Creator classes in the de.lessvoid.nifty.controls.dynamic package. Here is a reference of all the available *Creator classes:

Classname	**Purpose**
CustomControlCreator	Create a new control instance. This is the same as the <control> tag in XML.
ImageCreator	Create a new image element.
LayerCreator	Create a new layer. Please note that you have to use screen.getRootElement() as the parent element when you call create() in this case.
PanelCreator	Create a new panel.

Classname	Purpose
PopupCreator	Create a popup element. Please note that you'll need to call registerPopup() instead of build() for the PopupCreator since you can only register new popups with Nifty instead of creating them directly. Popups have their own chapter in this book as well.
ScreenCreator	Create a new screen. Please note that the create() method of the Screen only requires the Nifty instance.
TextCreator	Create a new text element.

Besides the build-in *Creator classes the standard controls project introduces special classes for each of the standard controls that allows you to create them. You can find these classes in the de.lessvoid.nifty.controls.<controlname>.builder package.

Please note that they are called Create<ControlName>Control. Besides their name they work the same as the core *Creator classes.

JAVA BUILDER CLASSES

The Java Builder way to create elements works similar to the Creator classes but provides a somewhat nicer API. The trick is that the *Builder classes are designed in a way that feels more like a DSL (Domain Specific Language) for Nifty instead of a regular class. This is achieved by nesting anonymous inner classes with an initialize block.

Here is a short reminder what an initialize block is:

```
public class Stuff {
  {
    // you can do things in here to initialize this class
  }
}
```

And here is an anonymous inner class:

```
void someMethod() {
  new Stuff() {
    // define methods here and Java will create an anonymous inner class for it
  };
}
```

The Nifty Java Builders combine both so that we can create elements very easily.

EXAMPLE

Here is the panel we've seen before with the *Creator classes in the Java Builder version.

We'd like to add a new panel to the empty layer in the XML from above.

```
new PanelBuilder() {{
  height("8px");
  backgroundColor("#f00f");
}}.build(nifty, screen, layer);
```

So besides the duplicate {{ and }} this looks almost the same as the *Creator version but it is quite a bit shorter.

But the really interesting things are happening when we nest the Builders.

So in the next example we create the whole screen, with a layer and the panel using only Java Builders.

EXAMPLE

Create a complete screen with Java Builders only.

```
Screen screen = new ScreenBuilder("start") {{
  layer(new LayerBuilder("baseLayer") {{
    childLayoutCenter();
    panel(new PanelBuilder() {{
      height("8px");
      backgroundColor("#f00f");
    }});
  }});
}}.build(nifty);
```

And that's a very compact way to create a Nifty GUI!

You can find all the Builder classes in the de.lessvoid.nifty.builder package. Here is an overview of what you can find in that package:

Classname	**Purpose**
ControlBuilder	Create a new control instance. This is the same as the <control> tag in XML.
ControlDefinitionBuilder	Define a new control. This is the same as the <controlDefinition> tag.
EffectBuilder	Create a new effect. You can use this with the on<Effect>() methods of any Builder class.
HoverEffectBuilder	Create a new hover effect. You can use this with the onHover() method of all Builders that support onHover()
ImageBuilder	Create a new image. Use this with the image() method.
LayerBuilder	Create a new layer. Use this with the layer() method.

Classname	Purpose
PanelBuilder	Create a new panel. Can be used with the panel() method.
PopupBuilder	The PopupBuilder is used to register a new popup with Nifty (see the chapter about popups for an example)
ScreenBuilder	The ScreenBuilder adds a new screen to a Nifty instance.
StyleBuilder	Register a new style with Nifty using the StyleBuilder. This is the same as the XML <style> tag.
TextBuilder	The TextBuilder is used to build a new text element. You use it with the text() method.

ELEMENTS

SCREEN

WHAT IS A SCREEN?

The basic building block of any Nifty GUI is the concept of a screen. In XML it is defined in a <screen> element and this element acts as the root or parent element for all other GUI elements. A screen can also be used to manage individual states in an application. A typical Nifty GUI consists of several screens that are interconnected.

Every screen must be given a unique name with the id attribute. This way every screen can be identified and referenced. When loading XML-files with the fromXml() method the screen id must be specified to define which screen the GUI should start with. It is possible to switch screens from Java with the nifty.gotoScreen(String screenId) method which takes the screenId of the target screen as the parameter.

Here is a simple example of a screen definition using XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <screen id="start">
    <!-- content of the screen -->
  </screen>
</nifty>
```

and the same example using Java Builders:

```
Screen screen = new ScreenBuilder("start") {{
  // content of the screen
}}.build(nifty);
```

SCREEN CONTROLLER

The ScreenController is a Java class that implements the Nifty ScreenController interface. Every screen has a ScreenController instance attached. If you don‘t provide one then Nifty will use a simple default implementation.

Whenever something interesting is happening to a screen a method on its ScreenController instance is called. The ScreenController is also the place where Nifty will look for additional callback methods as you will see in Chapter about interacting with the GUI.

The ScreenController interface consists of three methods.

The first method gives you access to the main Nifty instance and the Screen class, the Java representation of the active screen. Nifty will call this method when it initializes the screen. The method is: bind(Nifty nifty, Screen screen).

There are two other simple methods in the ScreenController interface that are called in the screen life cycle: onStartScreen() and onEndScreen().

The ScreenController interface looks like this:

```
/**
 * ScreenController Interface all screen controllers should support.
 * @author void
 */
public interface ScreenController {
  /**
   * Bind this ScreenController to a screen. This happens
   * right before the onStartScreen STARTED and only exactly once for a screen!
   * @param nifty nifty
   * @param screen screen
   */
  void bind(Nifty nifty, Screen screen);

  /**
   * called right after the onStartScreen event ENDED.
   */
  void onStartScreen();

  /**
   * called right after the onEndScreen event ENDED.
   */
  void onEndScreen();
}
```

To connect a screen with a ScreenController you need to specify the fully qualified class name of your ScreenController in the controller attribute of the <screen> tag:

```
<screen id="start" controller="my.package.MyScreenController">
  <!-- ... -->
```

To provide Nifty with a ScreenController instance there are two ways possible:

1. Nifty creates a new instance of the given ScreenController class and registers this instance with the Screen using the bind() method.

2. You can give Nifty an existing ScreenController instance that matches the classname given in the controller attribute. In the example given you would give Nifty an instance of the my.package.MyScreenController class.

Nifty will first look for an existing instance and creates a new class only when it can't find an existing one.

To register a ScreenController instance with Nifty there are additional parameters on the fromXml() method. This way you can even add multiple different instances for use in multiple Nifty screens.

The fromXml method looks like this:

```
public void fromXml(String filename, String startScreen,
                    ScreenController ... controllers);
```

Nifty will use the className to match instances so you'll still need the controller attribute in the XML.

In case you want to use anonymous inner classes for your ScreenController like in this example:

```
class MyStuff {
 nifty.fromXml("menu.xml", "start", new ScreenController() {
    public void bind(Nifty nifty, Screen screen) {
      // ...
    }
  });
}
```

You'll need to specify the controller attribute like: „MyStuff$1".

When you use the Java Builders to create your GUI you can can directly set a ScreenController instance:

```
Screen screen = new ScreenBuilder("start") {{
  controller(new MyScreenController());
  // ...
}}.build(nifty);
```

DEFAULT FOCUS ELEMENT

Another attribute the screen element supports is the defaultFocusElement. You simply specify the element id of the element that should retrieve the keyboard focus when the screen is started. If you don't specify the defaultFocusElement then Nifty will use the very first focusable element on the screen. More about the keyboard focus is presented in a later chapter.

```
<screen id="start" defaultFocusElement="okButton">
  <!-- other stuff with a button element with the id „okButton" -->
```

It works the same using the Java Builder mechanism of course:

```
Screen screen = new ScreenBuilder("start") {{
  defaultFocusElement("okButton");
  // ...
}}.build(nifty);
```

SCREEN LEVEL KEYBOARD EVENTS

There are two other attributes available for the Screen element: inputMapping and inputMappingPre. These attributes are explained in the Nifty InputEvents and InputMapping section of the Interaction chapter later as well.

LAYER

Within a screen you can have several layers of elements. A layer is a container for other elements. You can stack layers on top of each other. So for example you can use a layer for the background and another layer on top of it to display elements. Layers are rendered in the order they appear in the screen. So for the background example you should define your background layer first and all other layers after it.

EXAMPLE

Here is an example screen that consists of two layers:

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <screen id="start">
    <layer id="background">
      <!-- background layer content in here -->
    </layer>
    <layer>
      <!-- content for this layer in here -->
    </layer>
  </screen>
</nifty>
```

Using the Java Builder pattern it would look like this:

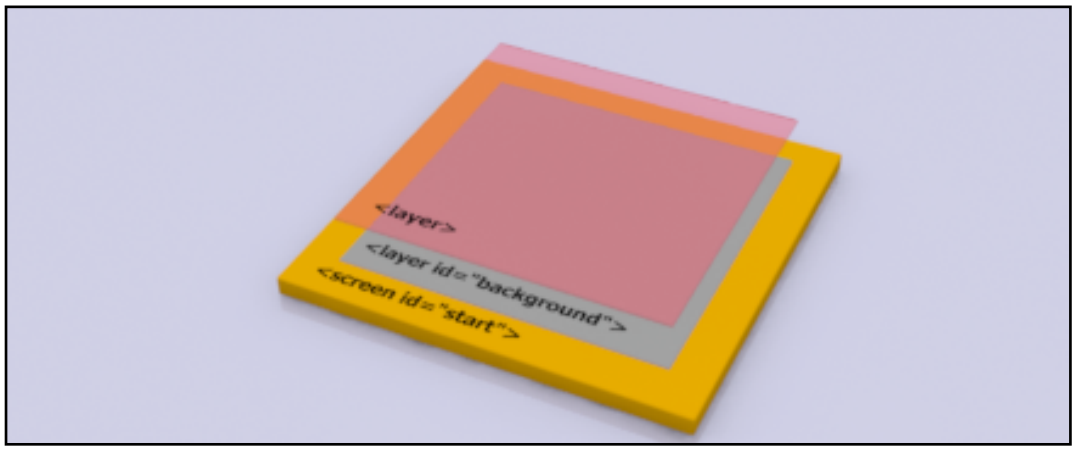
```
Screen screen = new ScreenBuilder("start") {{
  layer(new LayerBuilder("background") {{
    // background layer content in here
  }});
  layer(new LayerBuilder() {{
    // content for this layer in here
  }});
}}.build(nifty);
```

As you can see the background layer is defined first and therefore will be rendered first as well.

By default layers are transparent which means that you would actually see nothing rendered if you would try this example as is.

Opposite to screen definitions the id attribute of the layer element is optional. You can give layers a name in form of the id attribute if you later need to reference a layer from Java. For instance if it's necessary to dynamically hide or show a layer you can access the layer from Java using its id and toggle it's visibility.

The following picture shows a visualization of what is going on in the example:



As you can see from the picture the layer with the id="background" will be rendered below the second layer that does not have an id.

A layer supports all of the general element attributes that are explained later in this chapter.

PANEL

A Panel is a (usually) invisible helper element that can contain other elements. Panels are normally used to help layout other elements. The next chapter will discuss how layout in Nifty works.

Besides layout you could use panels for design purposes since they can be given a backgroundColor. So if you need a solid colored rectangle you could use a Panel for this as well.

The Panel element name is <panel> in XML and there is a PanelBuilder available for the Java Builder pattern.

Like the layer element panels support general element attributes as explained a bit later.

Panels support the backgroundImage attribute so that you can set a background image for the panel. There are additional properties available to influence the way the backgroundImage is applied. This is the filter and the imageMode attribute. Both work the same as for the image element and are explained below when we discuss the image element.

EXAMPLE

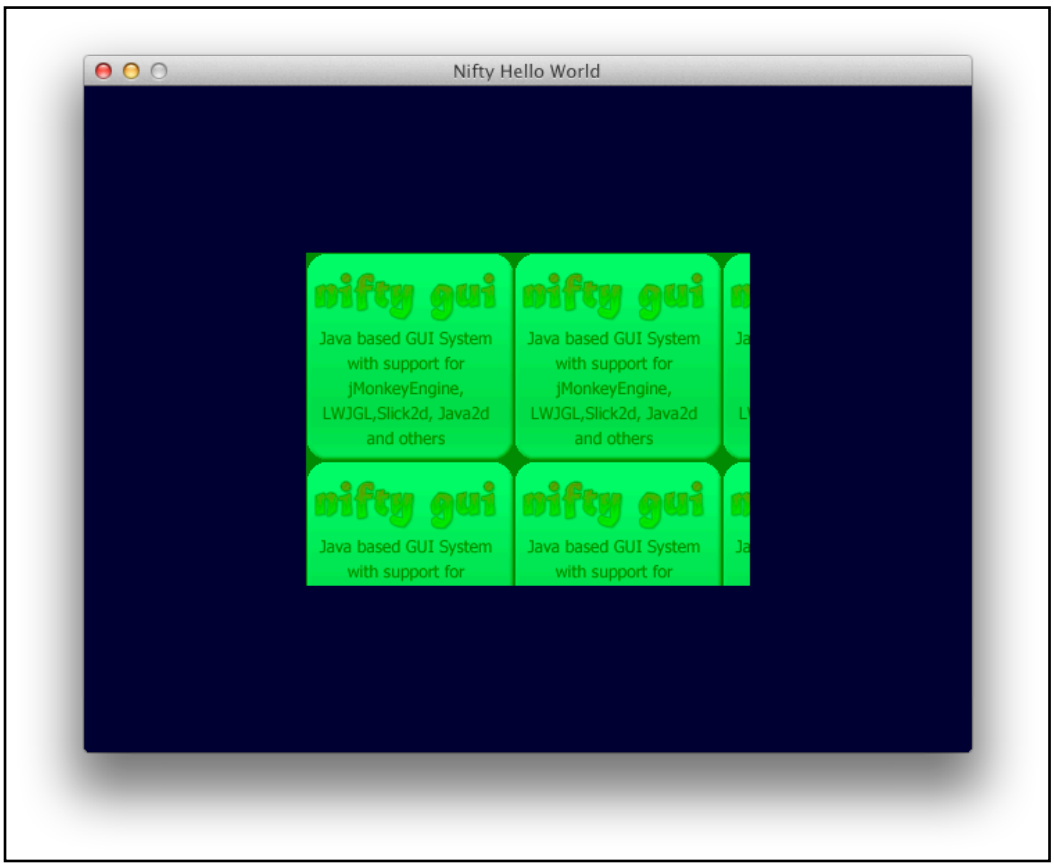
Panel with backgroundColor and backgroundImage attributes applied.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
   <screen id="start">
    <layer id="layer" backgroundColor="#003f" childLayout="center">
      <panel width="50%" height="50%" backgroundImage="nifty-logo-150x150.png"
             imageMode="repeat:0,0,150,150" backgroundColor="#0f08" />
    </layer>
  </screen>
</nifty>
```

Please note that the backgroundImage is rendered before the backgroundColor when you apply both attributes. So the backgroundColor acts as a color overlay when a backgroundImage is given and that's why it uses an alpha value of #8.

In this example we use the repeat image mode to tile the background image and overlay it with a half transparent green color.

We get this as the result:



More about the imageMode="repeat:..." can be found below when we discuss the image element.

As always the example works the same using the Java Builder:

```
Screen screen = new ScreenBuilder("start") {{
  layer(new LayerBuilder("layer") {{
    backgroundColor("#003f");
    childLayoutCenter();
    panel(new PanelBuilder() {{
      width("50%");
      height("50%");
      backgroundImage("nifty-logo-150x150.png");
      imageMode("repeat:0,0,150,150");
      backgroundColor("#0f08");
    }});
  }});
}}.build(nifty);
```

TEXT

The text element is used to output text. Usually renderers are using bitmap based fonts although what kind of font formats are supported depends on the actual system and the Nifty RenderDevice implementation.

You can specify font, color and alignment properties for the text and the text can be modified from Java.

EXAMPLE

Here is a basic example that displays a simple text.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <screen id="start">
    <layer childLayout="center">
      <text font="aurulent-sans-16.fnt" color="#f00f" text="Hello World!"
            align="center" valign="center" />
    </layer>
  </screen>
</nifty>
```

As usual it works the same using the Java Builder pattern:

```
Screen screen = new ScreenBuilder("start") {{
  layer(new LayerBuilder() {{
    childLayoutCenter();
    text(new TextBuilder() {{
      font("aurulent-sans-16.fnt");
      color("#f00f");
      text("Hello World!");
      alignCenter();
      valignCenter();
    }});
  }});
}}.build(nifty);
```

The attributes of the <text> element should be easy to understand. Using the font attribute you specify the font. Using the color attribute you specify the color of the text (including alpha) and finally the text attribute will specify the actual text String that you want to be displayed.

COLOR ENCODED TEXT

Nifty supports encoding colors into the text string. This works with a special syntax. You include a special kind of String directly into the text to change the color. This string starts with „\#“ followed by three values, one for red, green and blue as hexadecimal values (optionally followed by an alpha value). The string has to end with a single „#“ character.

Say you have the String „Hello World“ and you want the word „World“ to be colored in red. Then you can specify the text attribute like so: „Hello \#ff0000#World“ and this text would be displayed like: „Hello World“.

ADDITIONAL TEXT PROPERTIES

There are some additional text properties available for the text element:

„textLineHeight“ as SizeValue, Default: null
The textLineHeight property influences the height of the text element. Usually the height of the text element is calculated as the height of the font.

If you set the textLineHeight property you can override that height. This way you can enforce a certain height of the text element.

„textMinHeight“ as SizeValue, Default: null
The textMinHeight property can be used to set a minimal height of the text element. When the calculated height of the text element is lower than the textMinHeight value then the textMinHeight value is being used.

„textVAlign“ as one of „top“, „center“, „bottom“, Default: „center“
It is possible that the text element area is actually bigger than the text itself.

For instance if you use width=“100%“ and/or height=“100%“ then Nifty will calculate the size of the (text) element like it would do for any other element. But that could mean that the text only needs a small area of the actual space of the element.

With the textVAlign property you can specify how the text should be aligned vertically in the element area. For example you could align the text to the top or bottom of the element. Or you can leave it at the „center“ which is the default.

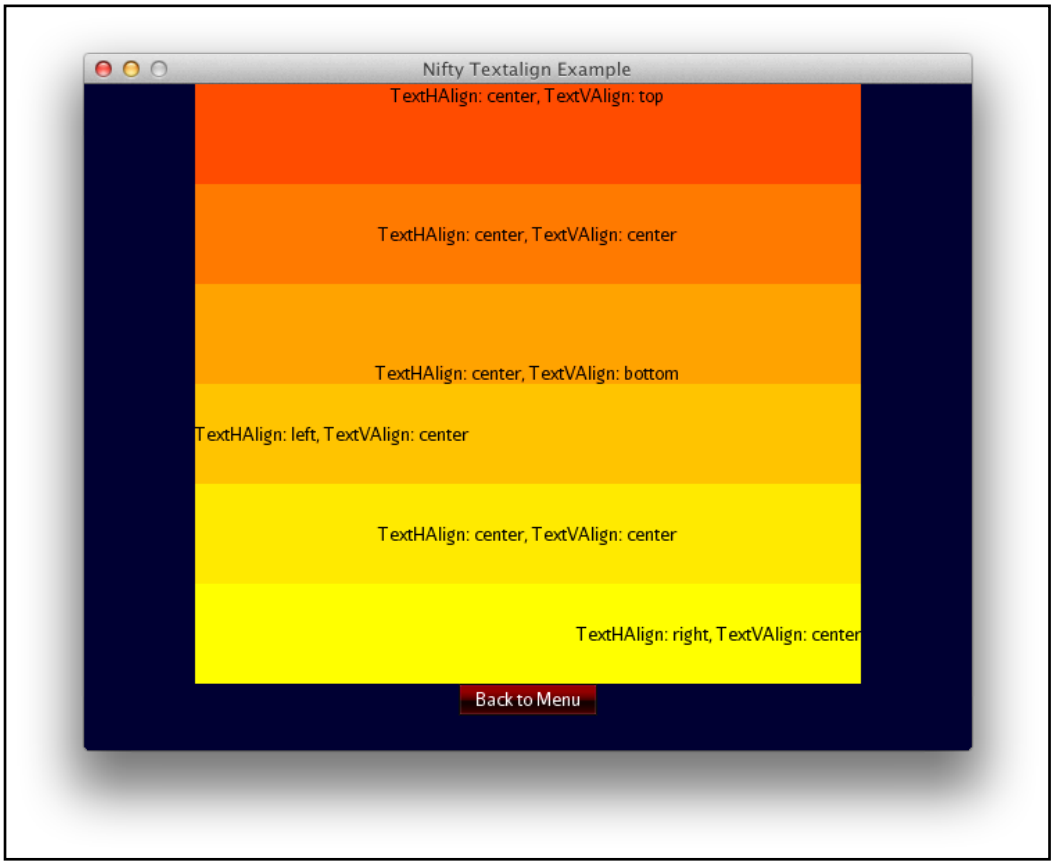
„textHAlign“ as one of „left“, „center“, „right“, Default: „center“
The textHAlign property works the same as the textVAlign property only for the horizontal alignment. It allows you to change the horizontal alignment of the text inside of the text element from the default value, which is again „center“, „left“ or „right“

EXAMPLE

Here is a XML example of a screen that displays text with different alignments. This example is from one of the Nifty examples (slightly modified).

```
<layer id="layer" backgroundColor="#003f" childLayout="vertical">
  <text id="text1" style="nifty-label" height="15%" width="75%"
        backgroundColor="#f60f" text="TextHAlign: center, TextVAlign: top"
        color="#000f" textHAlign="center" textVAlign="top"/>
  <text id="text2" style="nifty-label" height="15%" width="75%"
        backgroundColor="#f80f" text="TextHAlign: center, TextVAlign: center"
        color="#000f" textHAlign="center" textVAlign="center"/>
  <text id="text3" style="nifty-label" height="15%" width="75%"
        backgroundColor="#fa0f" text="TextHAlign: center, TextVAlign: bottom"
        color="#000f" textHAlign="center" textVAlign="bottom"/>
  <text id="text4" style="nifty-label" height="15%" width="75%"
        backgroundColor="#fc0f" text="TextHAlign: left, TextVAlign: center"
        color="#000f" textHAlign="left" textVAlign="center"/>
  <text id="text5" style="nifty-label" height="15%" width="75%"
        backgroundColor="#fe0f" text="TextHAlign: center, TextVAlign: center"
        color="#000f" textHAlign="center" textVAlign="center"/>
  <text id="text6" style="nifty-label" height="15%" width="75%"
        backgroundColor="#ff2f" text="TextHAlign: right, TextVAlign: center"
        color="#000f" textHAlign="right" textVAlign="center"/>
</layer>
```

And this is how it looks.



„selectionColor“ as Color, Default: null

The text element supports selection of text and the selectionColor attribute specifies the color of how the selected text is being rendered. Selecting text is probably only usable in the textfield control which actually allows selecting text by the user of the GUI but the core text element already support this.

„wrap“ as Boolean, Default: false

Usually Nifty will not automatically wrap text lines when they are too long which is the default value of wrap=“false“. In that case Nifty will simply render the text and will eventually draw text outside of the element boundaries.

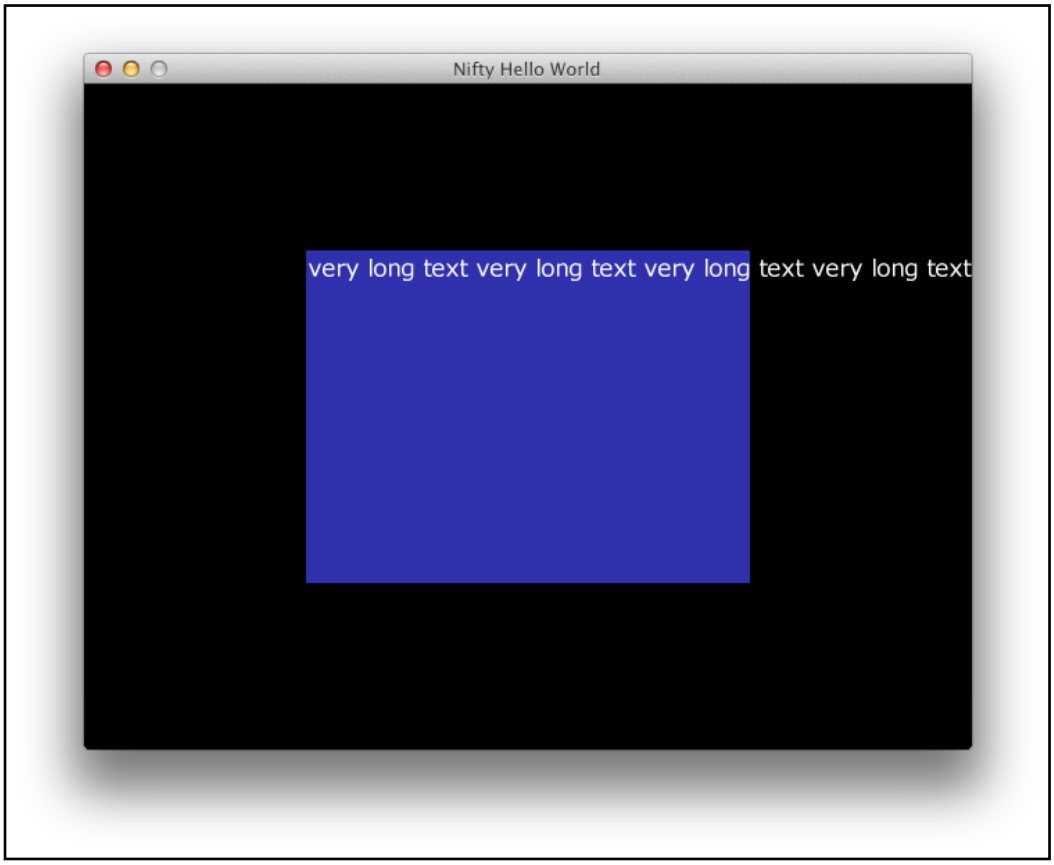
You can change this by setting wrap=“true“. This will make text lines automatically wrap when they would be longer than the element width. Setting wrap=“true“ will only work when you set a width for the text element, so that Nifty knows when to wrap a line.

EXAMPLE

In this example we start with text that is way longer then the element width.

```
  <screen id="start">
    <layer id="layer" childLayout="center">
      <text width="50%" height="50%" backgroundColor="#33af" font="verdana-small-
regular.fnt" textHAlign="left" textVAlign="top" text="very long text very long text
very long text very long text very long text very long text very long text very long
text "/>
    </layer>
  </screen>
```

The text element is 50% of the screen width and is centered in the middle of a screen. This will make the text go over the element boundaries, since the wrap attribute defaults to false:

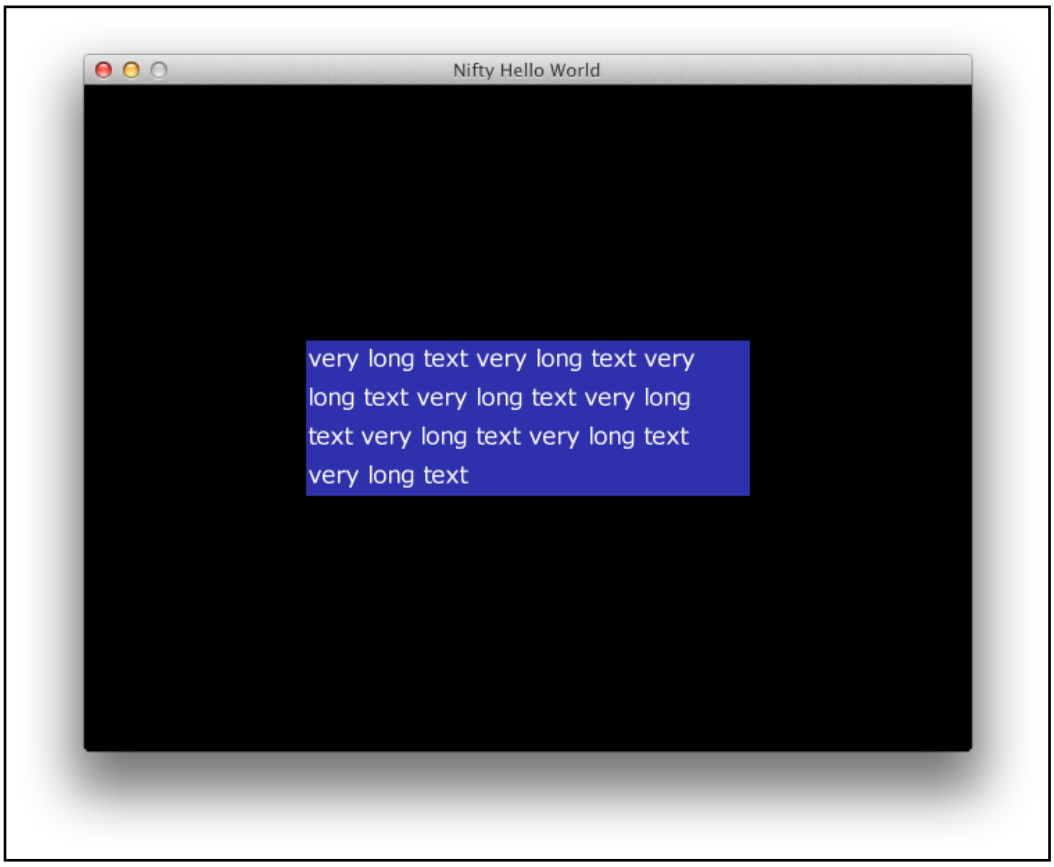


We can change this by adding wrap="true" to the text element.

```
  <screen id="start">
    <layer id="layer" childLayout="center">
      <text width="50%" height="50%" backgroundColor="#33af" font="verdana-small-
regular.fnt" wrap="true" textHAlign="left" textVAlign="top" text="very long text very
long text very long text very long text very long text very long text very long text
very long text "/>
    </layer>
  </screen>
```

Since we have also set a width for the text element Nifty will now wrap the text. Nifty will first try to wrap the lines at any whitespace character. If this is not possible it will try to wrap individual characters.

This is the result of adding wrap="true" to the example XML:



IMAGE

GENERAL PROPERTIES

The image element is used to display an image. In its simple form you only need to specify a filename of the image and Nifty will automatically read it, forces the element to be the size of the loaded image and then displays the image.

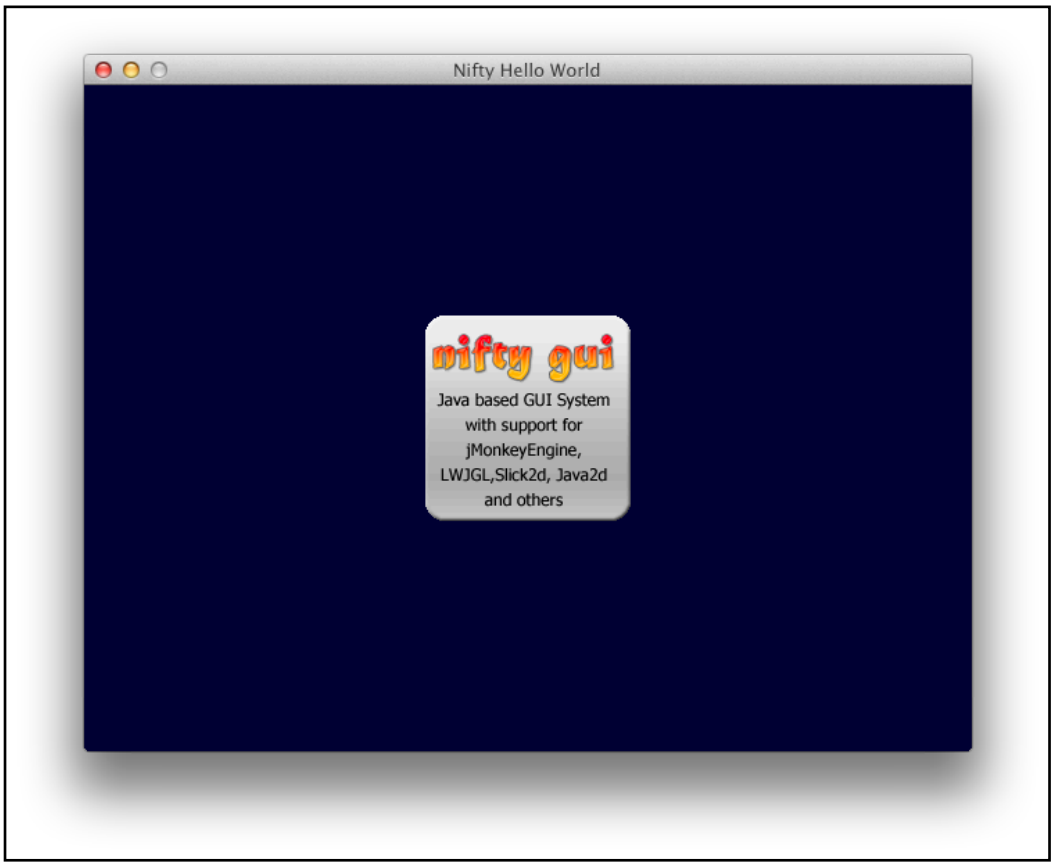
You can change the width and height of the image by providing the width and height properties and Nifty will resize the image accordingly.

EXAMPLE

Simple display of an image using XML.

```
<screen id="start">
  <layer id="layer" childLayout="center">
    <image filename="nifty-logo-150x150.png" />
  </layer>
</screen>
```

Which looks like this.



And as always it works the same using Java Builder:

```
Screen screen = new ScreenBuilder("start") {{
  layer(new LayerBuilder() {{
    childLayoutCenter();
    image(new ImageBuilder() {{
      filename("nifty-logo-150x150.png");
    }});
  }});
}}.build(nifty);
```

But there is a bit more to the image element. The following additional attributes are possible.

„filter“ as boolean, Default: false
Use linear filtering of the image when set to „true“ or nearest when set to „false, which is the default.

„inset“ as SizeValue, Default: 0px
Using the inset parameter you can scale the image smaller or greater than it‘s original size. Using a positive value for inset will make the area of the image smaller. So for instance when you set inset to „20px“ then you get an inner border of 20px and the image is being scaled to fit into the area that is now 20px smaller (at each border). Using negative values for inset will effectively make the image being drawn over the boundary of the element.

IMAGEMODE PROPERTY

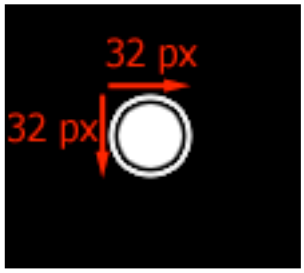
The imageMode attribute can greatly influence the way image data is being interpreted by Nifty. There are lots of different options.

imageMode=„normal“ (the default value)
The standard rendering of images. When the width/height attributes on the <image> element are not set the image is being drawn in its original size. If width and height are set the image will be stretched or shrink accordingly.

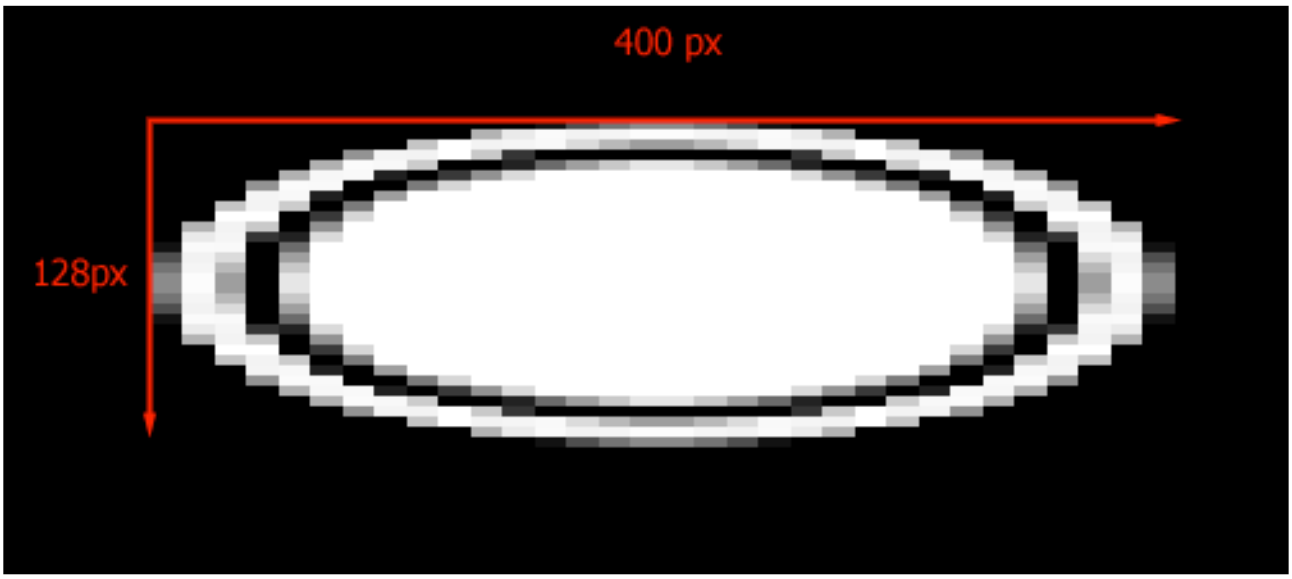
imageMode=“resize:x0,x1,x2,y0,x3,x4,x5,y1,x6,x7,x8,y2"
The resize imageMode enables "smart" resizing of images that allows especially the corners of the image to stay at the same size while the rest of the image is being resized. This mode is especially useful to keep round corners of images round that would otherwise be scaled/shrink.

EXAMPLE

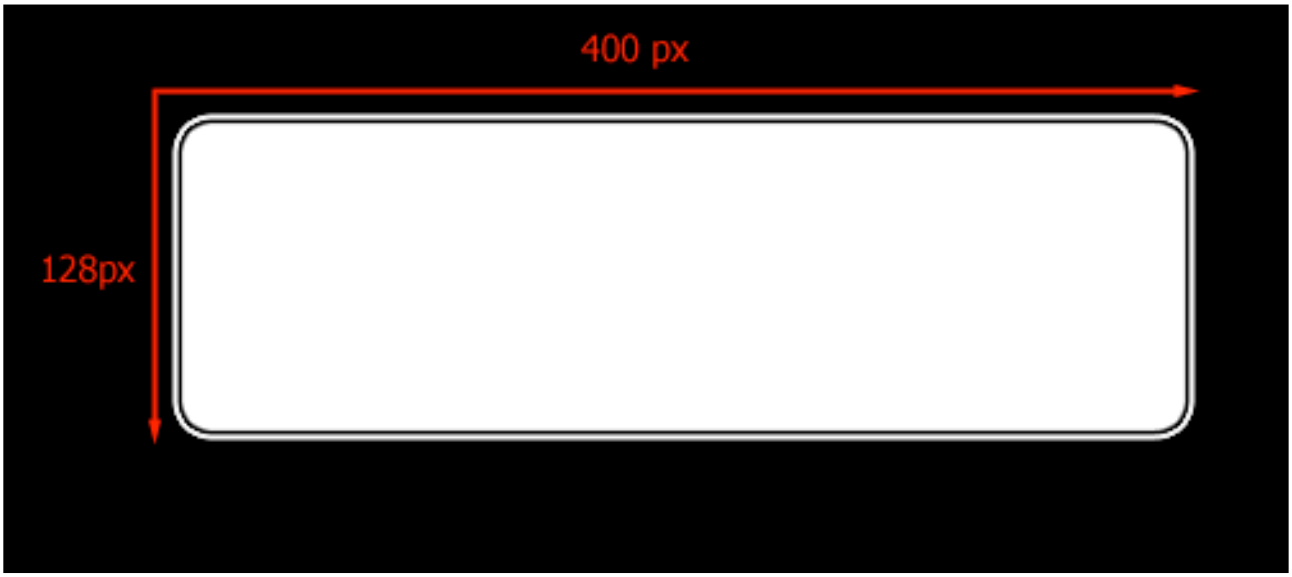
Let‘s assume we have this source image, which is 32x32 pixel in size:



If we would resize this image to, for instance, 400x128 pixel we would end up with a ugly stretched mess of an image.

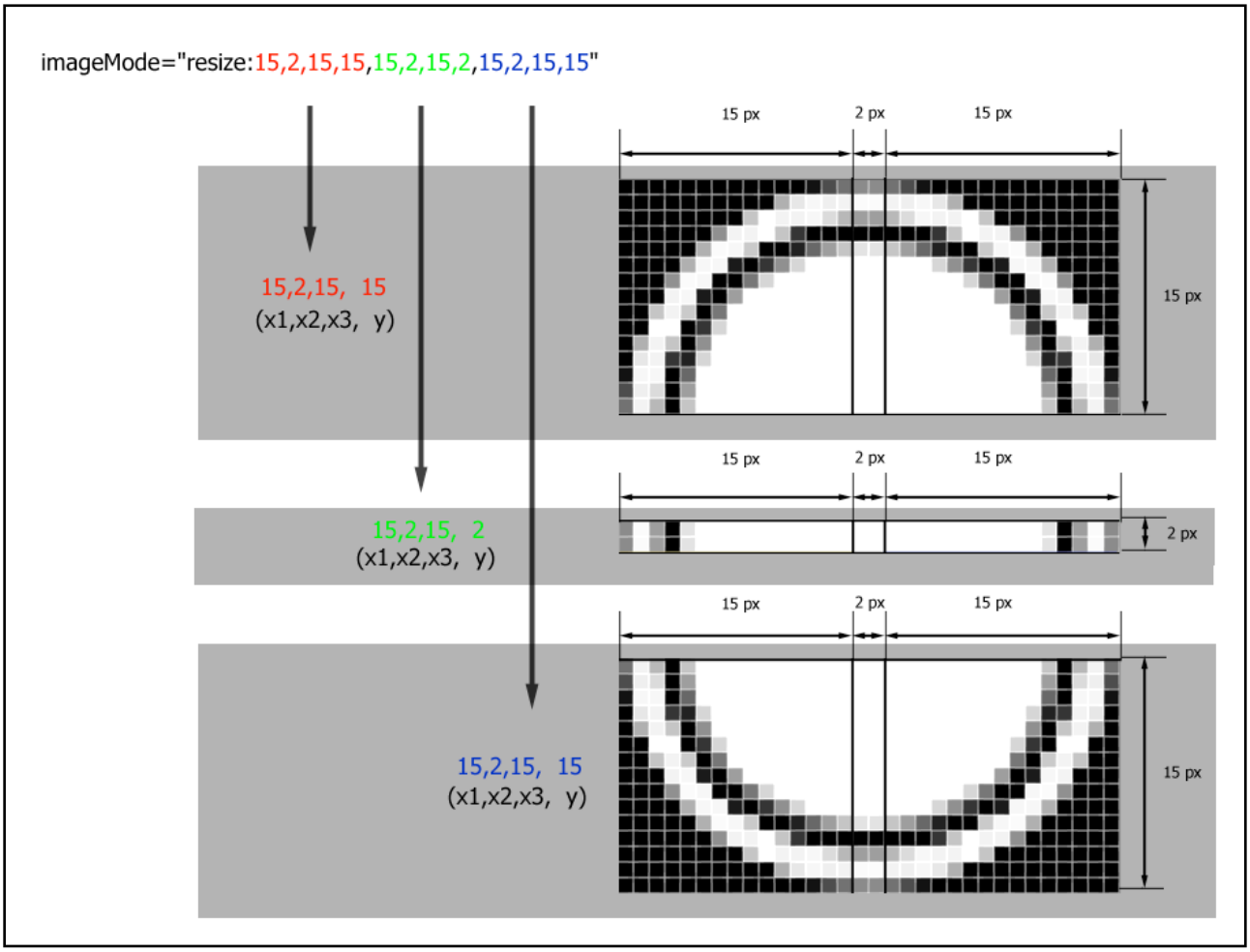


But when we add the magic attribute imageMode="resize:15,2,15,15,15,2,15,2,15,2,15,15" with lots of funny numbers :) to the image we‘ll get a much better result:



The way this works is actually pretty simple. The funny numbers just specify the size of the corners that will stay the same size when the image will be resized. The part of the image that will be scaled is just the middle of the image.

In the following image the funny numbers of the „resize“ imageMode are explained.



This way the top left and top right corners as well as the bottom left and bottom right corners will stay the same size regardless of the final size of the image. The part of the image that will scale is only the part in the middle.

Please note: There is a minimal size of the image that will work for resizing. In the 32x32 pixel image above it's 32x32 pixel. You can't make the image smaller than 32x32 pixel in this example case. But you can of course scale it way bigger and still keep the corners in great shape.

imageMode="subImage:x,y,w,h"
This imageMode will use the given sub image for rendering instead of the whole image. This way you can use the same image and just render different parts of it.

EXAMPLE

imageMode="subImage:10,10,32,32"

Nifty will use the sub image from x="10" and y="10" with a width and height of 32 pixel each.

imageMode="subImageDirect:x,y,w,h" (Nifty 1.3.2)
This imageMode will use the given sub image for rendering instead of the whole image like the „subImage“ mode. Instead of drawing the image scaled to element size the image will be rendered at it‘s original size and position.

imageMode="repeat:x,y,w,h"
This imageMode will use the given sub image for rendering like in the subImage rendering mode but it will repeat the image when it is applied to a bigger area.

imageMode=“sprite:w,h,index"
This will treat the given image as a sprite sheet. The "w" and "h" properties specify the size of a single sprite and the image is being interpreted as being fragmented into w*h sized boxes. This means you have "original width of image / w parameter" sprites on the x-axis and "original height of image / h parameter" sprites on the y-axis. The index property will specify which sprite you‘ll want to draw.

EXAMPLE

imageMode="sprite:32,32,5"

In this example w and h are each 32 pixel in size and therefore the example would expect a sprite sheet with 32x32 pixel sprites. The last parameter (in the example the value 5) will find the 5th sprite in the image and it will use this sprite for rendering.

imageMode=“sprite-resize:w,h,index,x0,x1,x2,y0,x3,x4,x5,y1,x6,x7,x8,y2"
This is a combination of the sprite mode and the resize mode. The sprite given will be treated with the resize parameters to allow resizing of the sprite while keeping the corners the same size.

COMMON ELEMENT ATTRIBUTES

All elements (layer, panel, image and text) will support the following common attributes.

„id“ as String, Default: null
The id attribute of an element allows you to later reference the element from Java. The chapter on modifying elements from Java explain how you can use the id attribute to access elements. The id attribute is optional.

„width“ as SizeValue, Default: null
The width attribute specifies the width of the element for layout types that support the width attribute. The layout chapter has lots of examples on how to use the width property. If you omit the width attribute the default value depends on the layout type of the parent element.

For image elements the default width is the width of the image.

„height“ as SizeValue, Default: null
The height attribute specifies the height of the element and works exactly the same as the width property.

„align“ as one of „left“, „center“, „right“, Default: depends on childLayout type of parent
The align attribute defines the horizontal alignment of the element inside its parent element. How this works exactly depends on the childLayout type of the parent element which will also define the default value. The layout chapter explains this in more detail for the different layout types.

„valign“ as one of „top“, „center“, „bottom“, Default: depends on childLayout type of parent
The valign attribute defines the vertical alignment of the element inside its parent element. Again the default value and if valign is supported depends on the childLayout property of the parent element. The layout chapter will also explain the valign properties in detail.

„childLayout“ as one of „vertical“, „horizontal“, „center“, „absolute“, absolute-inside“, „overlay“, Default: null
The childLayout attribute defines how any child elements of this element will be layout. There is a dedicated chapter available in layout and how the different layout types work.

„childClip“ as Boolean, Default: false
The childClip attribute defines if child elements are allowed to be drawn outside of this element boundary. By default childClip is set to false which means that child elements can be moved or drawn outside of the element bounds. If set to true then child elements will be clipped at the boundary of the element. The layout chapter has an example of childClip set to true when childLayout=„absolute“ is explained.

„visibleToMouse“ as Boolean, Default: false
The visibleToMouse attribute is by default set to false which makes the element not receive any mouse events. You can set it to true so that the element receives mouse events. What that means is explained in the interaction chapter.

„style“ as String, Default: false
Defines the style to be applied to the element. What styles are and how they work is explained in the Nifty Styles chapter.

„visible“ as Boolean, Default: true
Elements will be visible by default but you can hide them by setting visible=“false“. Please note that invisible elements do still participate in layout in Nifty.

„focusable“ as Boolean, Default: false
By default Nifty elements don‘t get the keyboard focus (focusable=“false“). You can make any element to be able to get the keyboard focus by setting focusable=“true“. There are more details to focusable elements that are explained in the Interaction chapter.

„focusableInsertBeforeElementId“ as String, Default: null
By default Nifty will use the original order of the elements as the keyboard focus order. With the focusableInsertBeforeElementId attribute you can enforce a different order by setting the id of an element where this element should be inserted before. This is especially useful when you dynamically add and remove elements and you need to enforce a specific order in the keyboard focus.

„controller“ as fully qualified Java class, Default: null
You can attach a controller class to any element which is an important concept and is therefore discussed in detail in its own section in the Interaction chapter.

POPUP LAYERS

INTRODUCTION

A general Nifty layer element is used to organize elements on the screen in, well, layers. A typical Nifty layer element acts as the parent of other elements. A layer element always has the same size as the screen. You can stack several layers above each other and layers can be hidden, created and removed dynamically.

This regular layer is treated by Nifty as a none blocking element. This means that when you stack several layers above each other and you have elements on the layers that can react to mouse events then all mouse events will reach the elements no matter what layer they are a part of.

Sometimes however, you might want to have a layer that blocks all interaction with the layers below. A modal dialog in a standard windowing system would be the perfect example for this. As long as the modal dialog is shown no interaction with the other elements below that modal dialog are possible.

Nifty popup layers do exactly this. When a "popup" layer is visible it will automatically block all interaction with other layers. Only the "topmost" layer, the popup-layer, will receive events.

Here is a screenshot from one of the Nifty example projects. The exit dialog is part of a special Nifty popup layer. Actually the popup layer covers the whole screen and the exit dialog is just a panel in the center of this popup layer. You can see in the screenshot that everything around the exit dialog is a little darker. That‘s because the popup layer has a semitransparent backgroundColor.

As long as the popup layer is being shown there is no interaction with other elements possible.

DEFINE POPUP LAYERS

A popup layer shares the same properties as a general layer. The only difference is the name of the tag in XML. Instead of <layer> the tag is simply being called <popup>.

EXAMPLE

This example defines a popup and gives it the id „popupExit“.

```
<popup id="popupExit" childLayout="center" backgroundColor="#000a">
  <!-- add the actual popup content here (panels, images, controls) -->
<popup/>
```

When you define the popup content remember that the <popup> tag itself will be treated like a simple layer by Nifty.

As you can see popup layers can have a backgroundColor as well and they need a childLayout for their child elements. The id of a popup is important because popups will not be immediately displayed. Popups are only registered with Nifty when the <popup> tag is parsed by Nifty. Nifty provides special methods to display and hide popup layers dynamically.

Popups can be created using JavaBuilder classes as well. Here is the example using Java Builder to create the same popup:

```
new PopupBuilder("popupExit") {{
    childLayoutCenter();
    backgroundColor("#000a");
    // add the actual popup content here (panels, images, controls)
}}.registerPopup(nifty);
```

Please note that you need to call registerPopup() at the PopupBuilder to actually register the popup with Nifty.

CREATE POPUP INSTANCE

It's important to keep in mind that Nifty will not immediately create a new layer when it processes the <popup> tag or when you call registerPopup(). Before we can display the popup we'll need to create a popup instance first. This allows us to display multiple versions of the same popup when necessary.

To create the actual popup instance you can call nifty.createPopup() with the id of the <popup> tag:

```
Element popupElement = nifty.createPopup("popupExit");
```

Nifty will create a new element for the popup with the original <popup> element as the template. You can think of the new element as an instance of the original popup and Nifty will automatically create a new id for it.

DISPLAY POPUP INSTANCE

When we're ready to show the popup we can do so with the nifty.showPopup(Screen screen, String id, Element defaultFocusElement) method:

```
nifty.showPopup(nifty.getCurrentScreen(), popupElement.getId(), null);
```

Please Note: You need to call showPopup() with the id of the popup instance and not with the original id of <popup> definition!

With the last parameter of the showPopup() method you can set the keyboard focus to an element inside your popup layer if necessary. When you use null the first focusable element in the popup will automatically get the focus.

CLOSE AND REMOVE A POPUP

You can remove a popup by calling the nifty.closePopup(String id) method:

```
nifty.closePopup(popupElement.getId());
```

The Popup layer will be removed from the current screen.

You can reuse the popup instance and display it again when required.

LAYOUT

INTRODUCTION

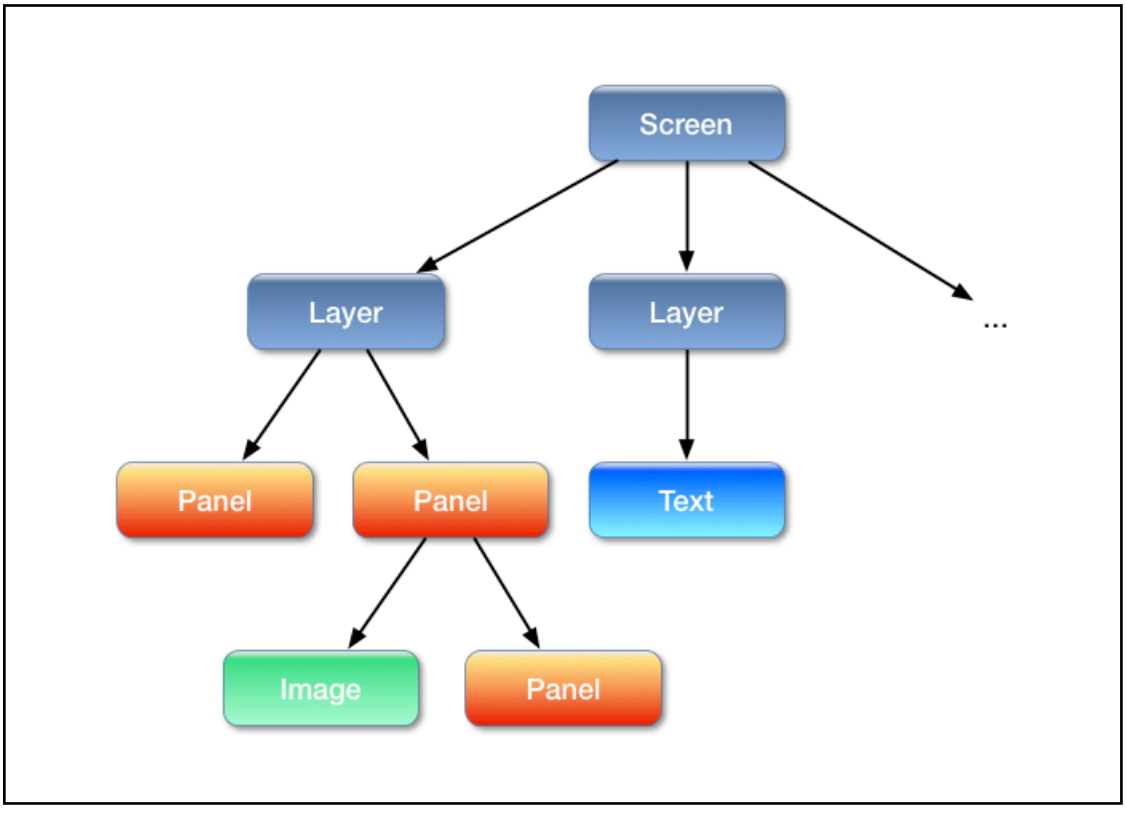
The key to understanding layout in Nifty is, that every Element can have child elements and that the parent element decides how these child elements are arranged within the parent element. The layout process starts at the top which is usually a Nifty layer element and is performed top down for all elements in the current screen. So what you end up is basically a tree of elements.

Let's assume your hierarchy of elements looks like this:

```
<screen>
  <layer>
   <panel />
   <panel>
     <image />
     <panel />
   </panel>
  </layer>

  <layer>
    <text />
  </layer>
</screen>
```

You'll get the following tree structure of elements internally:

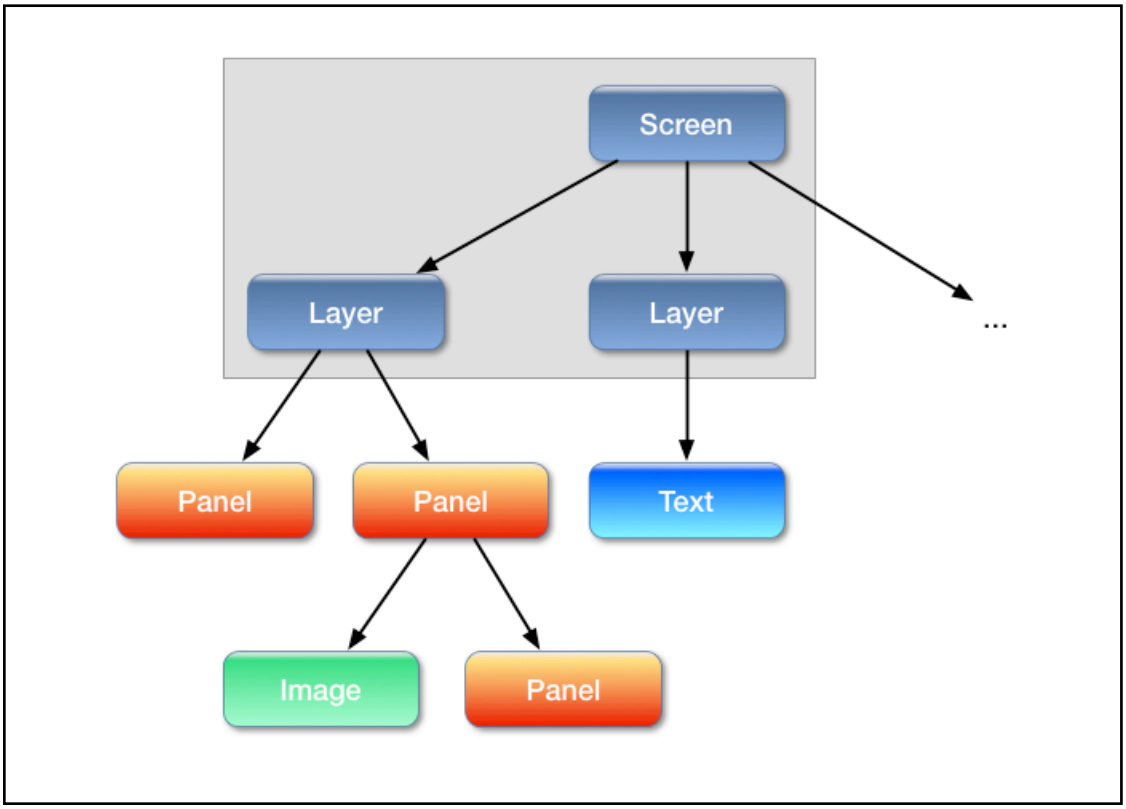


The layout process (and the rendering as well) starts at the top, at the screen element and then goes depth first through all elements. In the example illustration Nifty will visit the first layer, the one drawn on the left and will then continue to the first panel in this layer, the one without any child

elements on the left. Since this panel has no child elements there is nothing to do for the layout so Nifty will continue with the second panel of the first layer (the panel drawn on the right). Since this panel has two child elements, an Image and another panel it will visit these element to layout their content. Since they don't contain any child elements there is nothing to do and Nifty can now layout the image and the panel according to the layout rules that are described in this chapter. This whole process continues until all elements have been laid out. On every level in the tree the childLayout attribute of the parent element will define how the child elements will be positioned inside the parent element.

Before we get into all of the specific layout algorithms that Nifty provides there is one exceptional element that does not use a childLayout attribute: Layers are by default exactly the same size as the screen and therefore it's not necessary to specify a childLayout attribute at the screen element.

So in the relationship between the Screen and the Layers there is no childLayout attribute necessary (grey box in the picture):

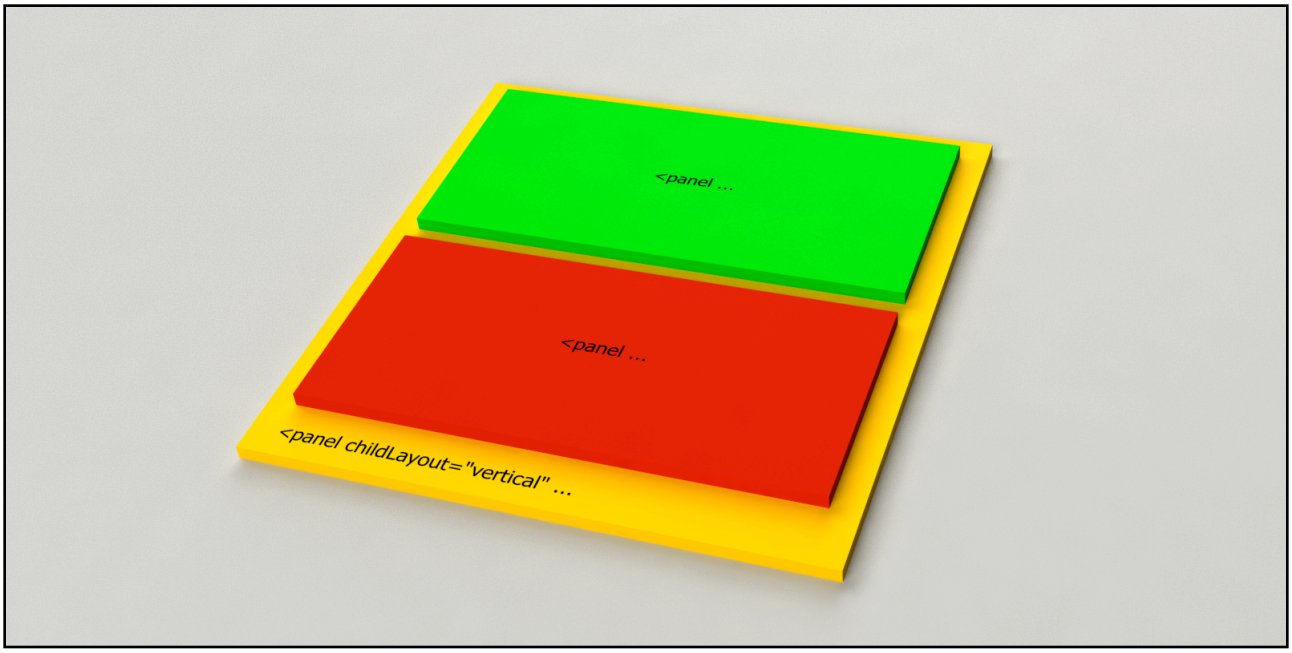


Starting with the Layer element all elements in the hierarchy should have a childLayout attribute set unless they are the last elements in the tree. Elements without any child elements don't need the childLayout attribute.

VERTICAL LAYOUT

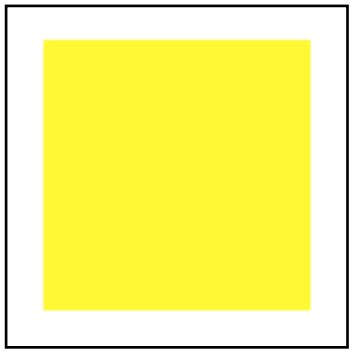
The childLayout="vertical" layout type will layout all child elements in a vertical way. This means that the second child element will always be positioned below the first element and so on. In the following illustration there is a single parent panel (colored in yellow) with two child panels. The first panel is colored in green and the second one in red.

If we don't set any constraints on any of the child elements then Nifty will look at the size of the parent and will try to fit all child elements into the parent element. In this case Nifty will automatically set the height of each child element to 50% of the parent height.



EXAMPLE

Here is another example. We start with a parent panel that has a fixed size. Let's say it has: width="100px" and height="100px". To see it better we set a yellow backgroundColor to the parent element. Without any child elements added we get a yellow rectangle:



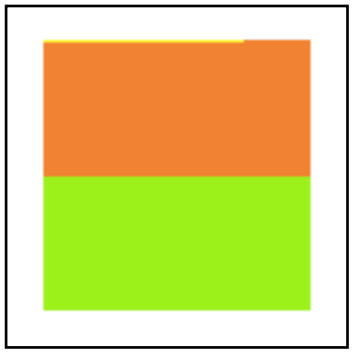
Next we will add two panels to the element and we set the childLayout attribute of the parent panel to „vertical". To better see what's going on we set a half transparent red and green background color for the child elements.

Our XML will now look like this:

```
<panel width="100px" height="100px" childLayout="vertical" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008"/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

If we don‘t constraint the child panels in any way then Nifty will use the available space (the height of 100px) to layout the two child elements. This will set the height of both panels to 50% which will be calculated to 50px. Since we‘re using childLayout=“vertical“ the width of the child panels will automatically be set to 100%.

We end up with this:

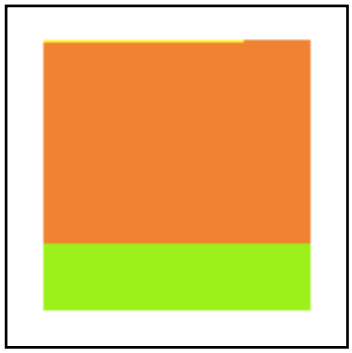


Of course we can modify the height attribute. Let‘s say we set the first panel height to 75%.

```
<panel width="100px" height="100px" childLayout="vertical" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" height=“75%“/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

Percent values are always calculated with respect to the corresponding value of the parent element. So with height=“75%“ for the first panel, the height will be calculated as 75% of 100px which is 75px.

And we will end up with this image:

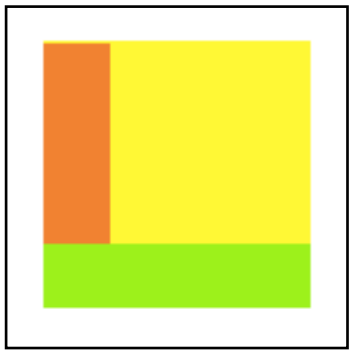


Please note that Nifty will automatically calculate the remaining space for all child elements without any height constraint. So in this case the green panel will automatically set to the remaining space which are 25px.

We‘re not limited of setting the height constraint. We can modify the width constraint as well. So in the following image we‘ll set the width of the first panel to 25px forcing it to this width.

```
<panel width="100px" height="100px" childLayout="vertical" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" height="75%" width=“25px“/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

By default Nifty will align the panel to the left which will lead to the following image.

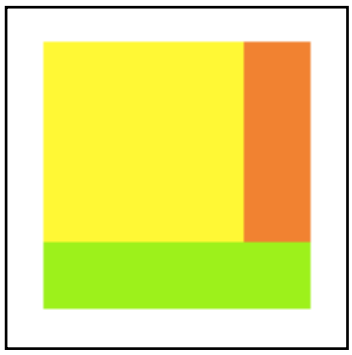


Now that the first child panel does not take up all of the vertical space anymore the align property can be used to specify how the panel should be aligned.

For instance we can set the attribute to right alignment using align=“right“ while keeping the width constraint to 25px.

```
<panel width="100px" height="100px" childLayout="vertical" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" height="75%" width="25px" align=“right“/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

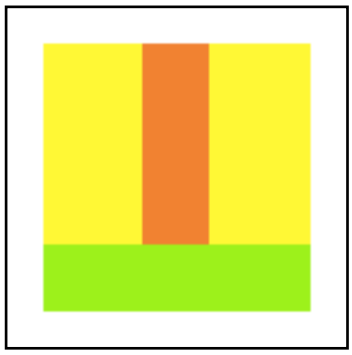
As expected this will align the first panel to the right.



And with align=“center“ we can align the first child panel in the center of the parent panel if we want to.

```
<panel width="100px" height="100px" childLayout="vertical" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" height="75%" width="25px" align=“center“/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

Which will get us this image.



Please note that since we‘re using childLayout=“vertical“ the child elements will always be aligned vertically. Therefore we can only change the horizontal alignment of the elements using the align attribute but not the vertical alignment using the valign attribute.

HORIZONTAL LAYOUT

The childLayout="horizontal" layout works exactly the same as the vertical layer but lays out the child elements horizontally instead of vertical.

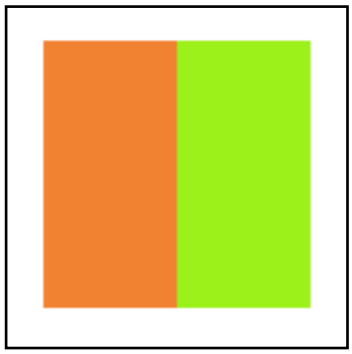
The horizontal layout illustrated:

EXAMPLE

We use exactly the same parent panel with a width of 100px and a height of 100px but this time using a childLayout="horizontal" instead of „vertical".

```
<panel width="100px" height="100px" childLayout="horizontal" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008"/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

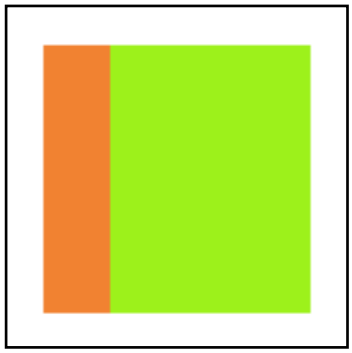
The result shouldn't be too much of a surprise. We have two horizontal aligned panels.



And as before we can add width or height constraints to the panels. Let's say we force the width of the first panel to 25px.

```
<panel width="100px" height="100px" childLayout="horizontal" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" width=“25px“/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

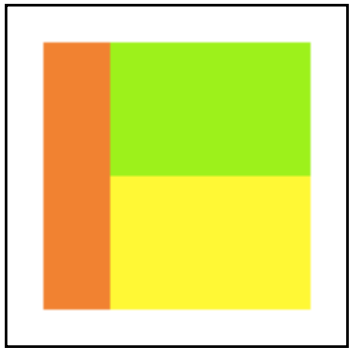
This will change the width of the first panel to 25px while the second panel - that we‘ve not set a width for - automatically takes up the remaining space.



We can change the height of a panel as well. Let‘s change the second child panel height to 50%.

```
<panel width="100px" height="100px" childLayout="horizontal" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" width="25px"/>
  <panel id="green" backgroundColor="#0f08" height=“50%“/>
</panel>
```

This will change the height of the second panel to 50%. Since this panel now is smaller we can see the yellow background color of the parent panel.

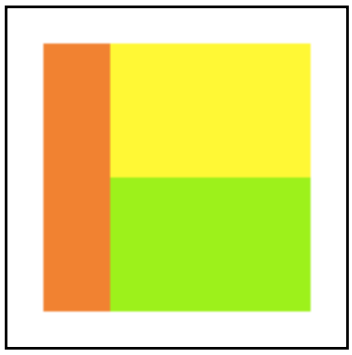


Similar to the vertical childLayout Nifty will set a default alignment if you don‘t specify one yourself. This time it will set the vertical alignment (valign) to valign=“top“.

But we can change this, f.i. to valign=“bottom“ when we need to.

```
<panel width="100px" height="100px" childLayout="horizontal" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" width="25px"/>
  <panel id="green" backgroundColor="#0f08" height="50%" valign=“bottom“/>
</panel>
```

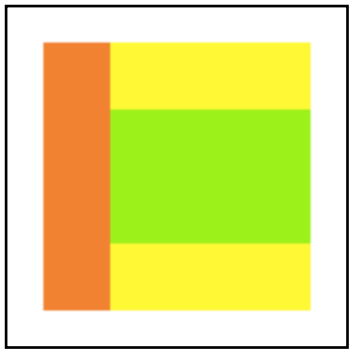
And sure enough will Nifty align the panel to the bottom of the parent panel.



And last but not least valign=“center“ works as well:

```
<panel width="100px" height="100px" childLayout="horizontal" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" width="25px"/>
  <panel id="green" backgroundColor="#0f08" height="50%" valign=“center“/>
</panel>
```

Which will vertically align the green panel in the parent element.



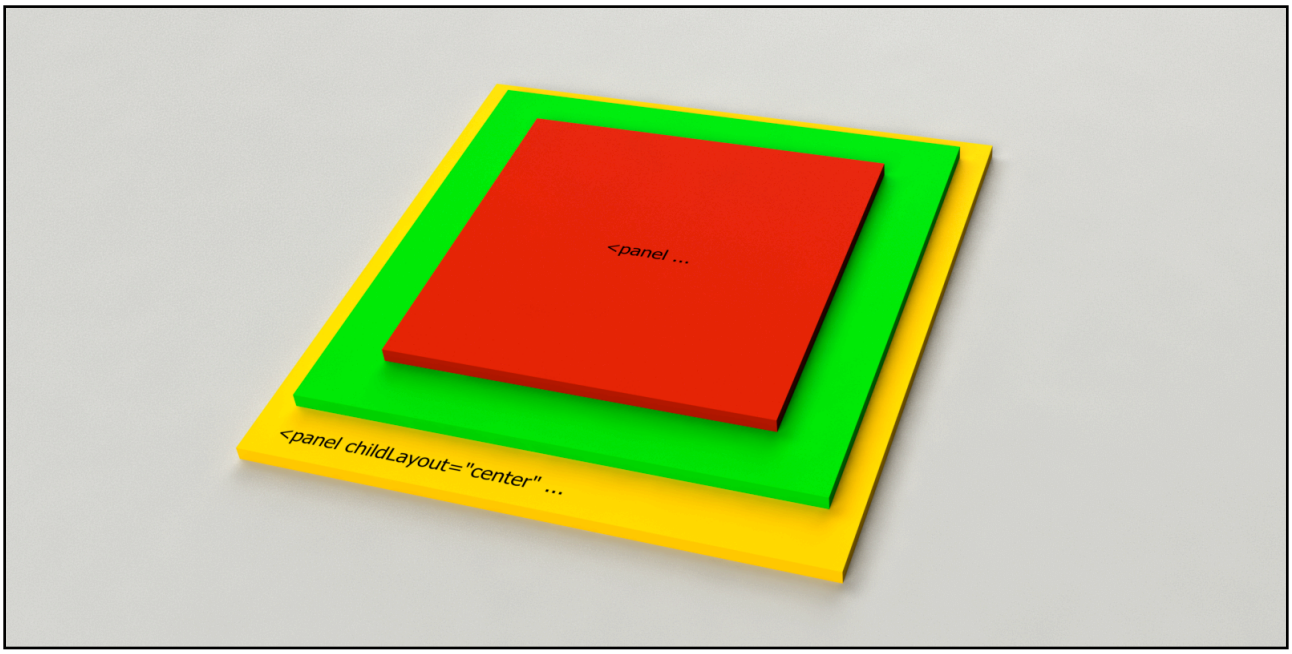
Similar to childLayout=“vertical“ is the horizontal align attribute not supported. Since the horizontal layout will always position the child elements horizontal it makes not much sense to change the horizontal alignment.

CENTER LAYOUT

The "center" Layout will center all of it's child elements in the middle of the parent area.

If you have more than one child element then all of them are centered and rendered above each other. The first child element will be rendered first and all of the other elements will be rendered above the others.

Here is an illustration using two child elements with childLayout="center".



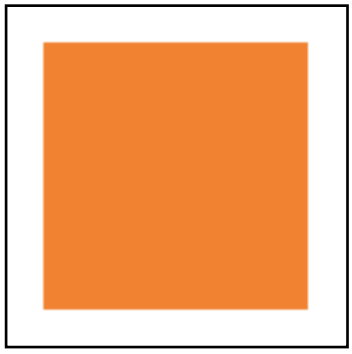
EXAMPLE

As before we'll use the 100x100 pixel parent panel with the yellow background color. This time we'll only add a single red panel to it.

We'll start without any width or height constraints set for the child panel:

```
<panel width="100px" height="100px" childLayout="center" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" />
</panel>
```

Well, the picture we get might be a little surprise:



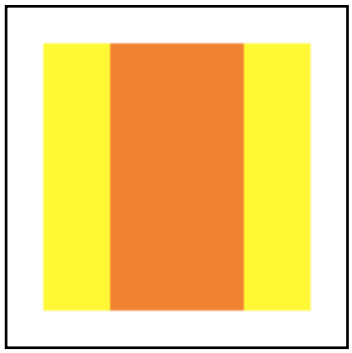
Shouldn't the red panel be centered in its parent?

Yes and actually it does center it. Since the red panel does not have a size set Nifty will automatically resize it to a width and height of 100%. What else could it do? You haven‘t specified a size.

So lets change that next. Again we start with a width constraint. Let‘s set it to „50%“.

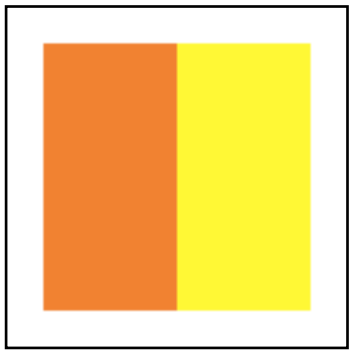
```
<panel width="100px" height="100px" childLayout="center" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" width="50%" />
</panel>
```

Which looks like this:



The width of the red panel is now 50% of the parent width (50 pixel) and it is centered in the parent element.

Attention Nifty 1.3.1 specific
If you would try the exact same example using Nifty 1.3 or earlier versions you would get a slightly different result:



The reason for the difference are the old default values for the align and valign attributes in older versions of Nifty. For enhanced flexibility Nifty allows you to specify the align and valign attributes even when you‘re using childLayout=“center“. This allows you to center a panel horizontally but still have it aligned at the top of the parent if necessary.

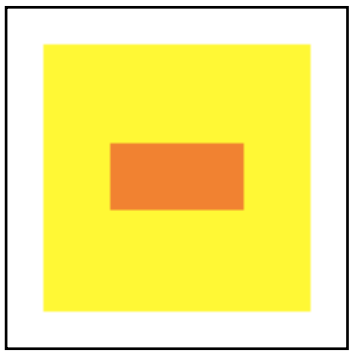
The default values for align and valign have been „left“ and „top“ in older versions of Nifty but have now been changed to align=“center“ and valign=“center“ in Nifty 1.3.1. Which will in most cases make more sense when you use childLayout=“center“ :)

Another thing that has been improved in Nifty 1.3.1 is that childLayout=“center“ now supports more than one child element! In versions prior 1.3.1 childLayout=“center“ would only support a single child elements and odd things would happen to the other elements. This has now been improved as well and childLayout=“center“ supports more than one child element.

But back to out example. Let's change the height constraint for out red panel as well. We'll keep the width at 50% and set the height to 25px.

```
<panel width="100px" height="100px" childLayout="center" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" width="50%" height="25px" />
</panel>
```

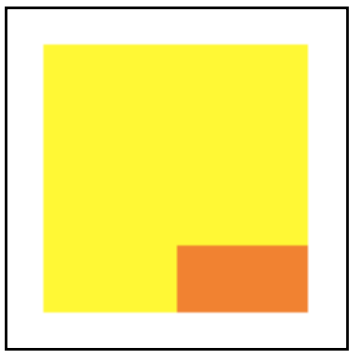
Which will give us this result.



And just to prove that align and valign really works we set align to „right" and valign to „bottom".

```
<panel width="100px" height="100px" childLayout="center" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" width="50%" height="25px"
         align="right" valign="bottom" />
</panel>
```

Which will align the red panel in the bottom right of the parent panel.



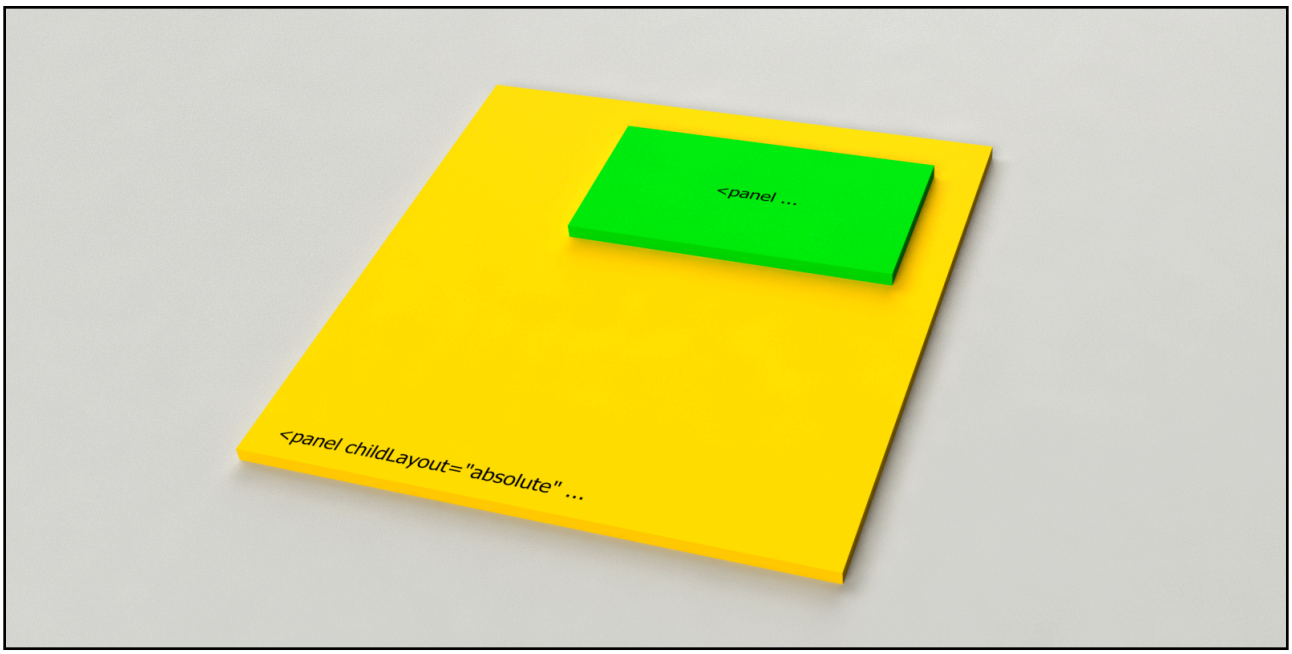
So the center panel can not only center a panel but you can still influence where the panel goes using the align and valign attributes.

ABSOLUTE LAYOUT

The "absolute" layout does not layout elements at all. It allows you to specify the position of the child elements with the "x" and "y" attributes directly as well as its size with the width and height attributes. The absolute layout will just take these values and apply them to the element.

The „absolute“ layout does not perform any other layout. This means that you‘ll need to specify all of the attributes by yourself. Make sure that when you use „absolute“ that you specify x, y, width and height.

Here is an illustration of childLayout=“absolute“:



EXAMPLE

As before we use our yellow parent element but this time we add three child panels with different values for x, y, width and height.

```
<panel width="100px" height="100px" childLayout="absolute" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" x="0px" y="0px" width="25px" height="25px"/>
  <panel id="green" backgroundColor="#0f08" x="0px" y="25px" width="100%"
         height="25px"/>
  <panel id="blue" backgroundColor="#00f8" x="50%" y="50%" width="50%" height="50%"/>
</panel>
```

And we get this result.

Three panels at different positions inside the parent. The coordinates you give the child elements are always in respect to the position of the parent element. So the coordinates x="0px" and y="0px" will correspond to the top left corner of the parent element.

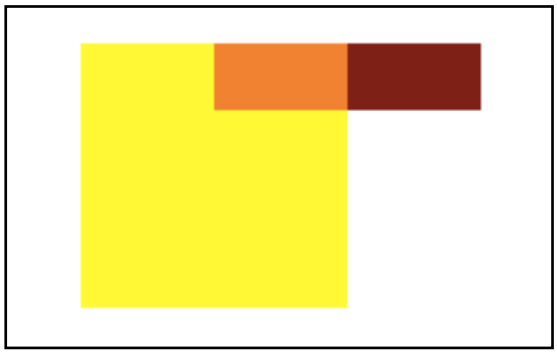
CLIPPING

By default Nifty does not clip content that is larger than the parent. So for instance using the absolute layout we can make child elements that are rendered outside of the parent area.

Here is an example of an child element that has 100% of its parent width but is being moved 50px to the right:

```
<panel width="100px" height="100px" childLayout="absolute" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" x="50px" y="0" width="100%" height="25px"/>
</panel>
```

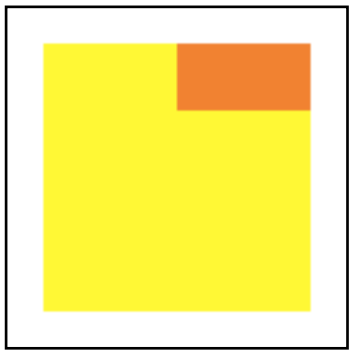
We end up with this result:



The red child element is drawn out of the bounds of the parent element since Nifty does not clip the content by default. This might or might not be what we want. In case we don't want to draw out of the parent area but we can't adjust the x position (or width) of the child element we can enable clipping by adding the childClip="true" attribute to the parent element. Nifty will now make sure that whatever we do we won't be able to draw outside of the parent element area by clipping content that would be outside.

```
<panel width="100px" height="100px" childLayout="absolute" backgroundColor="#ff0f"
       childClip="true">
  <panel id="red" backgroundColor="#f008" x="50px" y="0" width="100%" height="25px"/>
</panel>
```

And finally with childClip="true" we get this result:



Without changing the width of the child element nothing is being drawn outside of the parent element.

childClip="true" will not only work with childLayout="absolute" but with every layout type and element.

ABSOLUTE INSIDE

This layout type is related to the child clipping we discussed before. Sometimes we don‘t want to clip the child content and we don‘t want to resize our panel either. Instead we‘d like to dynamically adjust the position of the panel so that it is forced inside of the parent area.

This is especially important when the actual position of the child panel is controlled by the user. The perfect example is a hint panel that opens when the user hovers a button for instance. Since the button (or the mouse pointer) could be located anyway and maybe close to the border of the screen, the hint could be drawn outside of the screen. In that case clipping would not help us at all. A better approach would be to move the hint panel so that it stays within the screen.

Of course we could do the math ourself and when we need to display the hint panel we calculate a position for the panel where it is inside without changing its width or height.

Nifty can do that automatically for us using childLayout=“absolute-inside“.

This childLayout works exactly the same as the „absolute“ layout we‘ve just discussed with one exception: It‘s not possible to set a x or y position that would make the child element render outside of its parent area. In case it would be drawn outside Nifty will automatically adjust the position (x and y) so that the child element will stay inside the parent element.

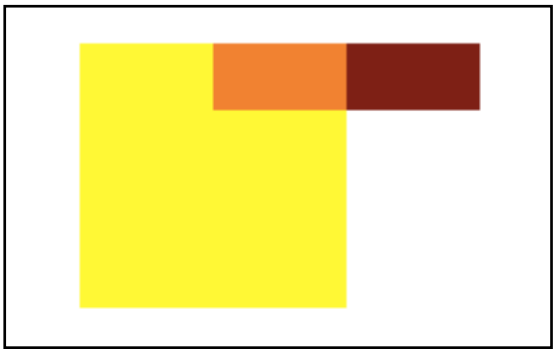
EXAMPLE

Let‘s start with the clipping example we‘ve discussed before. We have a child panel that is exactly as wide as its parent but we position it in the middle of the parent panel using x=“50px“.

Here is the XML using the original example with childLayout=“absolute“.

```
<panel width="100px" height="100px" childLayout="absolute" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" x="50px" y="0" width="100%" height="25px"/>
</panel>
```

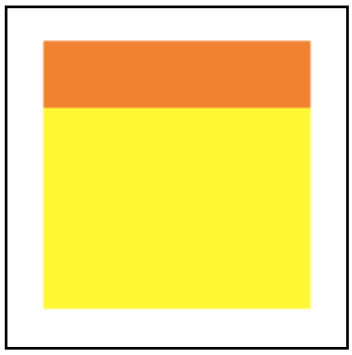
We end up with the panel being drawn out of the parent area.



So now we change the layout to childLayout=“absolute-inside“.

```
<panel width="100px" height="100px" childLayout="absolute-inside"
       backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" x="50px" y="0" width="100%" height="25px"/>
</panel>
```

We end up with the panel repositioned to stay within the parent panel:



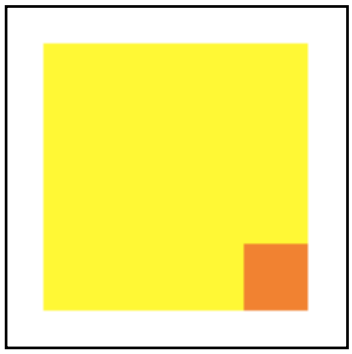
As mentioned above the main purpose of childLayout="absolute-inside" is to keep child panels inside of the parent especially when the user can control the position.

Here is another example demonstrating this.

We try to position a 25x25px red panel way outside of the parent panel setting x and y to 500px.

```
<panel width="100px" height="100px" childLayout="absolute-inside"
       backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" x="500px" y="500px" width="25px"
         height="25px"/>
</panel>
```

Nifty will try to keep the panel inside of the parent element and therefore it will reposition it to the lower right corner.



OVERLAY LAYOUT

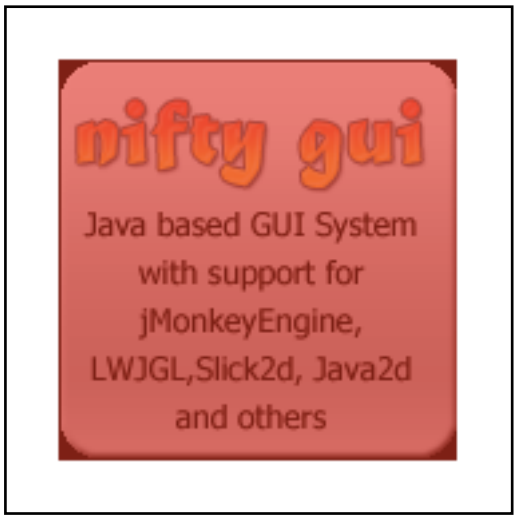
The „overlay“ childLayout is special since it doesn‘t layout anything at all. All of the child elements will be exactly the same size as the parent element „overlayed“ above the parent. You can imagine this layout type as some form of stacking of the child elements.

EXAMPLE

In the following example we overlay a semi transparent red panel above an image. In this case the image acts as the parent panel using childLayout=“overlay“.

```
<image filename="nifty-logo-150x150.png" childLayout="overlay">
  <panel id="red" backgroundColor="#f008" />
</image>
```

And here is the resulting image.



PADDING

Nifty supports a padding parameter similar to the CSS-Padding mechanism. In Nifty the „padding“ attribute is applied to an element and will reduce the inside of the element without changing it‘s size. This means there is less room available for the child elements of this element.

Currently padding is only supported for childLayout values of „vertical“, „horizontal“ and „center“.

A Nifty padding attribute consists of at least one and up to 4 comma separated Nifty SizeValues. These values work exactly like in the CSS-Padding property to yield individual values for left, right, top and bottom padding:

No. of values	Description	Example	Result (left, right, top, bottom)
one	Left, right, top and bottom are set to the same value.	„10px“	10px, 10px, 10px, 10px
two	The first value is used for top and bottom padding and the second value is used for left and right padding.	„10px,50px“	50px, 50px, 10px, 10px
three	The first value is used for top padding. The second value is used for left and right padding and the third value is used for bottom padding.	„1px,2px,3px“	2px, 2px, 1px, 3px
four	The values are applied in the order: top, right, bottom, left.	„1px,2px,3px,4px“	4px, 2px, 1px, 3px

Since the values are Nifty SizeValues you can use % values as well. For instance the value „10%“ will use 10% of the width and 10% of the height of the element as the padding value.

If you want to specify a single padding value for one of the sides of the element you can use the individual attributes `paddingLeft`, `paddingRight`, `paddingTop` or `paddingBottom`. So for example you could set `paddingLeft="10px"` to get a left padding of 10px and no padding on the other sides.

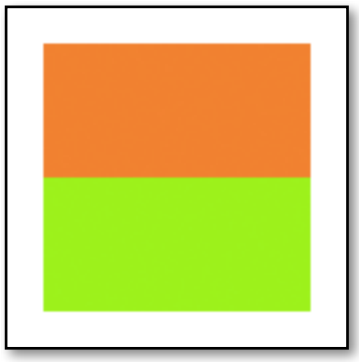
Let‘s look at some examples for padding now.

EXAMPLE FOR VERTICAL LAYOUT PADDING

We‘ll start with our two colored panel example from the layout example above:

```
<panel width="100px" height="100px" childLayout="vertical" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008"/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

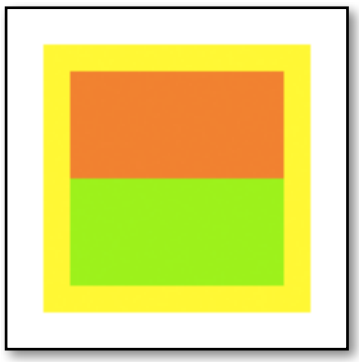
And we get the same result: the red and the green panel are vertically laid out.



Now lets add a padding of 10px.

```
<panel width="100px" height="100px" childLayout="vertical" backgroundColor="#ff0f"
padding=“10px“>
  <panel id="red" backgroundColor="#f008"/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

And we get this result:



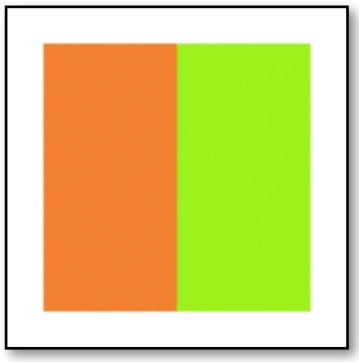
We now have a border of 10px inside of the element and we can now see the yellow backgroundColor which was completely covered by the child panels before.

EXAMPLE FOR HORIZONTAL LAYOUT PADDING

For the horizontal padding example we start again with the layout example shown before:

```
<panel width="100px" height="100px" childLayout="horizontal" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008"/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

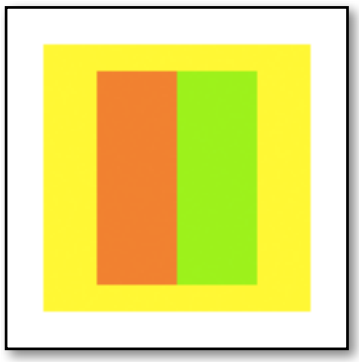
Here is the result of the starting point XML:



Now let's modify the XML and use a padding with two values:

```
<panel width="100px" height="100px" childLayout="horizontal" backgroundColor="#ff0f"
padding="10px,20px,,>
  <panel id="red" backgroundColor="#f008"/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

The value of `padding="10px,50px"` will give use a top and bottom padding of 10px and a left and right padding of 20px as shown in this picture:



We could get the same result by specifying the individual padding values for each side of course as well:

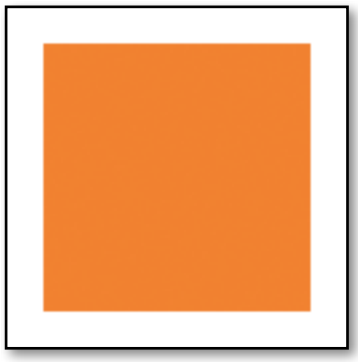
```
<panel width="100px" height="100px" childLayout="horizontal" backgroundColor="#ff0f"
paddingTop="10px" paddingBottom="10px" paddingLeft="20px,, paddingRight="20px">
  <panel id="red" backgroundColor="#f008"/>
  <panel id="green" backgroundColor="#0f08" />
</panel>
```

EXAMPLE FOR CENTER LAYOUT PADDING

Finally padding for the center layout works the same. Again we start with the example we've used above:

```
<panel width="100px" height="100px" childLayout="center" backgroundColor="#ff0f">
  <panel id="red" backgroundColor="#f008" />
</panel>
```

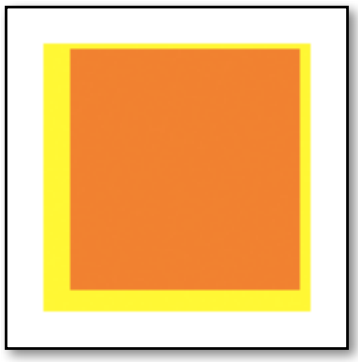
This gives us the centered red panel:



Now we'll use different padding values for each of the sides using the 4 value padding attribute:

```
<panel width="100px" height="100px" childLayout="center" backgroundColor="#ff0f"
padding="2px,4px,8px,10px">
  <panel id="red" backgroundColor="#f008" />
</panel>
```

And we'll get a padding of 2px on the top, 4px on the right, 8px on the bottom and 10px on the left as shown in this picture:



MARGIN (NIFTY 1.3.2)

Nifty 1.3.2 added support for basic margin support. It can be used to add an outer margin to elements. Basically it says: „hey parent layout, when you calculate my layout consider that I'm that much wider/higher but when you later render me just use my actual size" :)

It works similar to the HTML/CSS attribute with the same name although the implementation in Nifty is more basic, for instance margins don't collapse in certain situations as they do in HTML/CSS. There are even more restrictions like there is currently only margin support for „center", „vertical" and „horizontal" layouts. Other layouts will simply ignore the attribute.

You can specify margin values in the same way as you specify padding values. This means you can use a single „margin" attribute and specify a single, two, three or four values and they are interpreted exactly like in the table shown above. Or you can specify individual margin values using „marginLeft", „marginRight", „marginTop" or „marginBottom" attributes.

Please Note that contrary to the padding attributes margin values can only be used with „px" values currently. There is no support for „%" values at the moment.

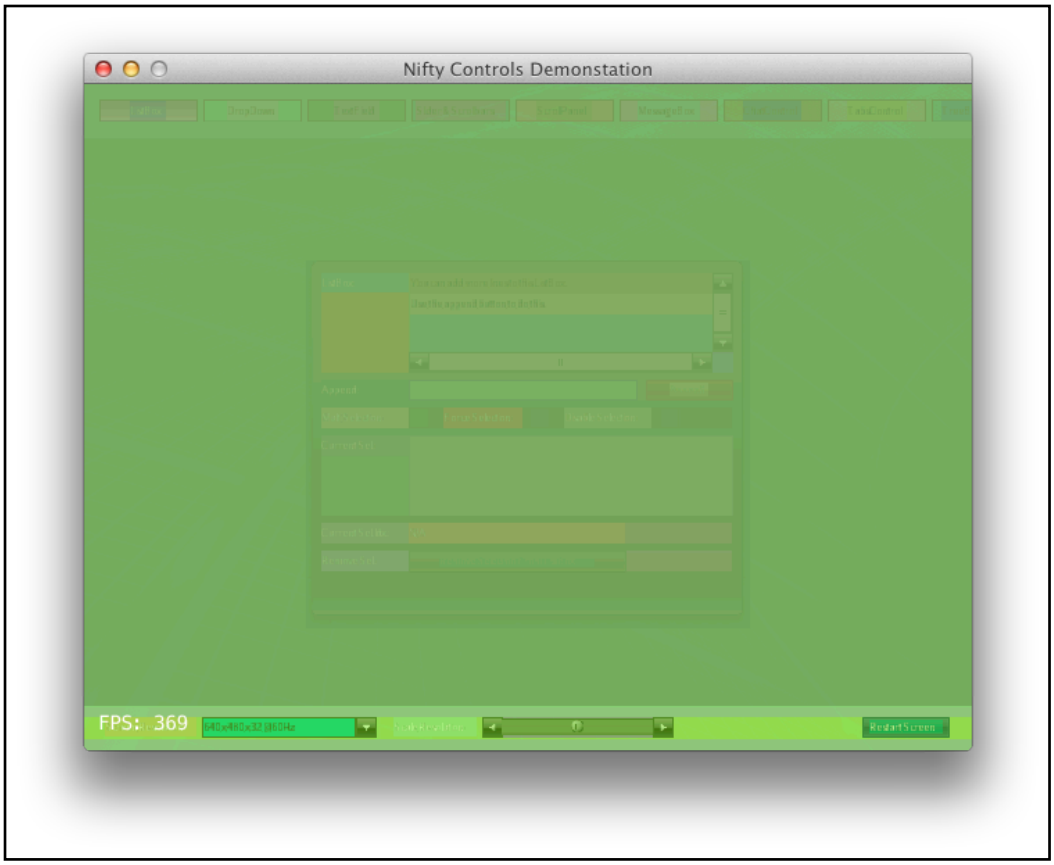
TROUBLESHOOTING LAYOUT

Sometimes layout can be tricky to debug. There are two ways to help you in debugging layout.

Nifty allows you to enable a debug rendering mode using the following method.

```
nifty.setDebugOptionPanelColors(true);
```

When being set to true Nifty will render a randomly colored rectangle above each element it renders. This way you can easily see where the layout is off or where maybe changing some alignment property might fix your issue.



Another way to troubleshoot layout issues is to get a reference to the current screen (f.i. using nifty.getCurrentScreen() or by keeping the Screen reference of the ScreenController bind() call) and then call screen.debugOutput(). This call will return a String that contains all of the attributes of every element on the screen including it's state and position.

This way you can check in detail if every element is exactly where it should be or where there is an error. The output of screen.debugOutput() looks something like that:

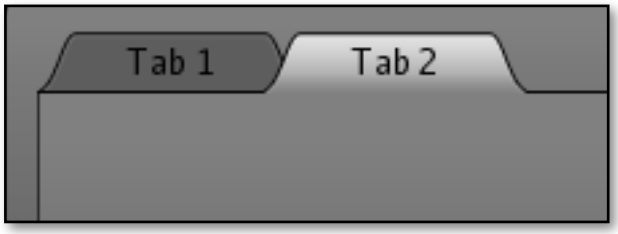
```
+[layer]
  style [null]
  state [normal]
  position [x=0, y=0, w=1024, h=768]
  constraint [null, null, null, null]
  padding [0px, 0px, 0px, 0px]
  flags [ enabled(0), visible]
  effects [  {}]
  [navigation] PanelType childLayout [horizontal]
   style [null]
   state [normal]
   position [x=0, y=0, w=1024, h=63]
   constraint [null, null, 100%, 63px]
   padding [20px, 20px, 20px, 20px]
   flags [ enabled(0), visible]
   effects [   {}]
   [menuButtonListBox] ControlType childLayout [center]
    style [null]
...
```

RENDER ORDER (NIFTY 1.3.2)

The standard order Nifty renders child elements is the order in which they are defined in XML or with Java Builder.

Let‘s say you use an image element with a childLayout=“overlay“ and you define two child panels, one with a red backgroundColor and the second one with a green backgroundColor. In this case Nifty will render the image, then the red panel and finally the green panel on top.

In most cases this is what you want but sometimes you need to modify this order. The new TabControl uses this feature to render the active Tab above all other tabs:



In Nifty 1.3.2 elements will be rendered ascending by a new attribute, the „renderOrder“ value which is an Integer. The default value for "renderOrder" is 0 which means that the original render order of the elements will be used (the order the elements are defined in XML). Internally the index in the elements list is used as the default „renderOrder“ value. So when you have a panel with 3 child elements and you don't specify a „renderOrder“ Nifty will internally use the values 0, 1 and 2.

You can change the „renderOrder“ value to some high value, like 10000 to enforce rendering this element on top of all other or to some very small number like -10000 to render it below the others.

Please note: When you use the same „renderOrder“ for two elements then the id will be used for comparison. If the elements don't have ids or the same id then object.toString() is being compared (the reference in memory).

BASIC EVENTHANDLING

INTRODUCTION

A GUI makes only sense if you can interact with it. This means that you can click on buttons, change scrollbars, enter text into a textfield and so on. Nifty supports several different mechanism to support interaction using the mouse and keyboard as the input device.

But before we discuss all of the events we need to talk about another important concept, the controller class.

ELEMENT CONTROLLERS

In the screen section of the elements chapter we've discussed the ScreenController. The ScreenController is the class that all interaction events will be routed to. But this is not the only way to attach a class to a Nifty element.

It is possible to add a class to any element using the controller attribute of an element. With it you can specify a fully qualified Java class that has to implement the Controller interface:

```
public interface Controller {
  void bind(
      Nifty nifty,
      Screen screen,
      Element element,
      Properties parameter,
      Attributes controlDefinitionAttributes);
  void init(Properties parameter, Attributes controlDefinitionAttributes);
  void onStartScreen();
  void onFocus(boolean getFocus);
  boolean inputEvent(NiftyInputEvent inputEvent);
}
```

So this looks a bit like the ScreenController interface and indeed it works in a similar way. If you attach a Controller to an element then Nifty will call the methods of your Controller instance at the appropriate times. Similar to the way Nifty calls the methods of a ScreenController.

The Controller, if present, will be the target for all events that the element generates (more about interact events below) and the event will travel the tree upwards calling the Controller of other elements all the way up to the ScreenController.

We'll see an example on how that works in a second but let's first look at basic mouse interaction events.

MOUSE EVENTS

INTRODUCTION

At the moment Nifty supports the following mouse events that you can intercept directly at any element that has visibleToMouse="true":

onClick
The element is being clicked by the mouse.

onClickRepeat
The element is being clicked by the mouse and the mouse button is hold down. This event is automatically generated as long as the mouse button is pressed.

onRelease
The mouse button is being released while the mouse cursor is over the element.

onClickMouseMove
The mouse button is moved while the mouse button is pressed and the mouse cursor is still hovering the element.

All events will use the primary mouse button. If necessary you can address the individual mouse buttons with additional events that work exactly as the standard ones but will only be executed when a specific mouse button is used.

First mouse button (usually the left mouse button)
onPrimaryClick
onPrimaryClickRepeat
onPrimaryRelease
onPrimaryClickMouseMove

Second mouse button (usually the right mouse button)
onSecondaryClick
onSecondaryClickRepeat
onSecondaryRelease
onSecondaryClickMouseMove

Third mouse button (usually the middle mouse button)
onTertiaryClick
onTertiaryClickRepeat
onTertiaryRelease
onTertiaryClickMouseMove

With the <interact> XML element, which you can add to any Nifty element, you can specify what method should be called when a mouse event occurs for the element. Nifty will try to call the method on the element controller first (if available) and then on all of the parent elements (when they have a Controller) until the event reaches the ScreenController.

The way this works is with Java reflection. You specify a method as a string, something like: „myMethod()“ and Nifty will look up a method with the name „public void myMethod()“ on the Controller.

EXAMPLE

Let‘s say we have an image. When the image is clicked with the mouse we want to execute a method with the name „next()“.

Again we start with our simple XML example containing a single Screen, a single layer and the image:

```
<screen id="start">
  <layer id="layer" childLayout="center">
    <image filename="nifty-logo-150x150.png" />
  </layer>
</screen>
```

To add the mouse event handler we add the <interact> XML element to the image element. And because Nifty will call the ScreenController when no other Controller class is available we add a ScreenController to the screen as well.

```
<screen id="start" controller="my.stuff.StartScreen">
  <layer id="layer" childLayout="center">
    <image filename="nifty-logo-150x150.png">
      <interact onClick="next()" />
    </image>
  </layer>
</screen>
```

Since by default Nifty elements do not receive mouse events it would be necessary to set the attribute visibleToMouse="true" but because we've added an <interact> tag Nifty will do that for us automatically.

Whenever you now click on the image Nifty will look for the method „public void next()" on the class „my.stuff.StartScreen" and if it exists it will call it.

Here is the ScreenController implementation with the next() method:

```
package my.stuff;

public class StartScreen implements ScreenController {

  @Override
  public void bind(Nifty nifty, Screen screen) {
  }

  @Override
  public void onStartScreen() {
  }

  @Override
  public void onEndScreen() {
  }

  public void next() {
    System.out.println("next() clicked! woohoo!");
  }
}
```

And of course the same works using Java Builder only.

```
new ScreenBuilder("start") {{
  controller(new MyScreenController());
  layer(new LayerBuilder("layer") {{
    childLayoutCenter();
    image(new ImageBuilder() {{
      filename("nifty-logo-150x150.png");
      interactOnClick("next()");
    }});
  }});
}}.build(nifty);
nifty.gotoScreen("start");

...

public class MyScreenController implements ScreenController {
  @Override
  public void bind(Nifty nifty, Screen screen) {
  }

  @Override
  public void onStartScreen() {
  }

  @Override
  public void onEndScreen() {
  }

  public void next() {
    System.out.println("next() clicked! woohoo!");
  }
}
```

CALL METHODS WITH STRING PARAMETERS

It is possible to use String parameters for the <interact> methods. So you can write onClick as follows:

```
<interact onClick="next(hello, there)" />
```

And when you just add a next() method with two String parameters to your ScreenController, Nifty will resolve the right method and calls it:

```
public class MyScreenController implements ScreenController {
...
  public void next(String param1, String param2) {
    System.out.println("next() clicked with parameters: " + param1 + ", " + param2);
  }
}
```

This will only work with Strings. So even when you write „next(12.2)“ then Nifty will still call the next() method with the String „12.2“.

MOUSE COORDINATES FOR ONCLICK AND ONCLICKMOUSEMOVED

A special method signature is supported to allow access to the mouse coordinates where the click occurs. Nifty will first look automatically for a method with two int parameters:

```
public class MyScreenController implements ScreenController {
...
  public void nextWithCoords(int x, int y) {
    System.out.println("next() clicked at: " + x + ", " + y);
  }
}
```

If it finds one it calls it with the x and y coordinates of the mouse cursor at the time of the click.

If no method with two int parameters could be found it will look for a method without any parameters and it will call this instead.

ADDITIONAL MOUSE EVENTS

There are some additional mouse events available.

onMouseOver

This method is executed when the mouse is hovering the element. So your method gets called when ever you move the mouse over the element.

If you define the callback method in Java to take two parameters, an Nifty Element instance and a NiftyMouseInputEvent instance then Nifty will call you with those parameters.

So when your method looks like this:

```
public void someMethodName(Element element, NiftyMouseInputEvent event);
```

Then you'll get the Element instance where the mouse event occurs and the NiftyMouseInputEvent which gives you access to the state of the mouse buttons and the position of the event.

Please note that you still only specify „someMethodName()" in the <interact> element event though the method has actually parameters.

onMouseWheel

This method is called when the mouse wheel is moved while the mouse hovers the element you've attached the <interact> onMouseWheel method to.

The additional method parameters Element and NiftyMouseEvent work exactly the same as in the onMouseOver case. For onMouseWheel you'll probably use this version because otherwise you don't get access to the actual position of the mouse wheel :)

ONCLICKALTERNATEKEY

There is one special attribute for the <interact> element. The onClickAlternateKey is a String that can optionally be set for the <interact> element. If you set it, then it will be used as the alternateKey for the current Nifty instance and all screens that are currently registered with Nifty.

This enables all effects that have been marked with the same key using the alternateEnable="key" attribute or it will disable effects that have been marked with alternateDisable="key". „key" in this example would be the value you provided for onClickAlternateKey.

You can read more about the alternate magic :) in the effects reference section of the next chapter.

ELEMENT CONTROLLER EXAMPLE

Now that we know what basic events are possible we can come back to our element controller example.

EXAMPLE

We start again with the screen that displays an image centered in the middle and with an onClick interaction method added to the image. As before the screen still has the same ScreenController attached (the one with the „public void next()“ method still in place) but now we set a controller attribute at the image element as well.

```
<screen id="start" controller="my.stuff.StartScreen">
  <layer id="layer" childLayout="center">
    <image filename="nifty-logo-150x150.png" controller="my.stuff.ElementController">
      <interact onClick="next()" />
    </image>
  </layer>
</screen>
```

The class „my.stuff.ElementController“ looks like this:

```
public class ElementController implements Controller {
  private Element element;

  @Override
  public void bind(
      Nifty nifty,
      Screen screen,
      Element element,
      Properties parameter,
      Attributes controlDefinitionAttributes) {
    this.element = element;
    System.out.println("bind() called for element: " + element);
  }

  @Override
  public void init(Properties parameter, Attributes controlDefinitionAttributes) {
    System.out.println("init() called for element: " + element);
  }

  @Override
  public void onStartScreen() {
    System.out.println("onStartScreen() called for element: " + element);
  }

  @Override
  public void onFocus(boolean getFocus) {
    System.out.println("onFocus() called for element: " + element + ", with: " +
getFocus);
  }

  @Override
  public boolean inputEvent(NiftyInputEvent inputEvent) {
    return false;
  }

  public void next() {
    System.out.println("next() clicked for element: " + element);
  }
}
```

And now lets run this example and click on the image!

We get the following output on the console:

```
bind() called for element: null (de.lessvoid.nifty.elements.Element@fba0f36)
init() called for element: null (de.lessvoid.nifty.elements.Element@fba0f36)
onStartScreen() called for element: null (de.lessvoid.nifty.elements.Element@fba0f36)

next() clicked for element: null (de.lessvoid.nifty.elements.Element@ed0220c)
next() clicked! woohoo!
```

So what's happening is that a bunch of methods are called to initialize the ElementController class (bind(), init() and onStartScreen() in this order) and when we click on the image the next() method of the ElementController is called first and then the next() method on the ScreenController is called as well.

If you change the next() method of the ElementController like so:

```
public boolean next() {
  System.out.println("next() clicked for element: " + element);
  return true;
}
```

then the method at the ScreenController is not called anymore.

Changing the return value of an interaction method to boolean and returning true prevents calling any other methods.

KEYBOARD EVENTS

NIFTY INPUT EVENTS AND NIFTYINPUTMAPPING

A NiftyInputEvent is a device neutral representation of an input event. The idea is that we don't want to be dependent directly on a keyboard when we code a GUI. If we later plan to add a game pad or some other input controller that is not a keyboard, we don't want to change the code of all of our elements. What Nifty does instead is to abstract the physical input event and creates an abstract representation for it. If we only depend on the abstract representation we can easily add support for different physical input devices later.

This all sounds probably more complicated then it really is. Take for example the TAB key which is usually used to change the focus from one element to the next. The physical input event would be the Keyboard TAB key pressed. Now instead of checking the TAB key directly Nifty maps the TAB key to the NiftyInputEvent.NextInputElement enum. Now our code only depends on NiftyInputEvent.NextInputElement and not the actual key. If we later support a gamepad which might have special keys to switch to the next input element we don't need to change the control code. We just have to add a different input mapping to Nifty and we're all set. That's the basic idea.

The way that physical events are mapped to NiftyInputEvents is just a class that implements the de.lessvoid.nifty.input.NiftyInputMapping interface, which looks like so:

```
public interface NiftyInputMapping {

  /**
   * Convert the given KeyboardInputEvent into a NiftyInputEvent.
   * @param inputEvent KeyboardInputEvent to convert
   * @return converted NiftInputEvent
   */
  NiftyInputEvent convert(KeyboardInputEvent inputEvent);
}
```

Currently Nifty only supports KeyboardInputEvents so the NiftyInputMapping interface currently only supports converting keyboard events into NiftyInputEvents but that might change later.

So a NiftyInputMapping implementation gets the KeyboardInputEvent and converts the keyboard event into a NiftyInputEvent. You can implement your own mapping but Nifty comes with predefined NiftyInputMappings that you can use directly.

You can find the default mappings in the de.lessvoid.nifty.input.mapping package.

Here is one of the existing mappings, the de.lessvoid.nifty.input.mapping.DefaultInputMapping class.

```
public class DefaultInputMapping implements NiftyInputMapping {

  public NiftyInputEvent convert(final KeyboardInputEvent inputEvent) {
    if (inputEvent.isKeyDown()) {
      if (inputEvent.getKey() == KeyboardInputEvent.KEY_F1) {
        return NiftyInputEvent.ConsoleToggle;
      } else if (inputEvent.getKey() == KeyboardInputEvent.KEY_RETURN) {
        return NiftyInputEvent.Activate;
      } else if (inputEvent.getKey() == KeyboardInputEvent.KEY_SPACE) {
        return NiftyInputEvent.Activate;
      } else if (inputEvent.getKey() == KeyboardInputEvent.KEY_TAB) {
        if (inputEvent.isShiftDown()) {
          return NiftyInputEvent.PrevInputElement;
        } else {
          return NiftyInputEvent.NextInputElement;
        }
      }
    }
    return null;
  }
}
```

SCREEN LEVEL KEYBOARD EVENTS

It is possible to handle keyboard input events at the screen level. Screen level input events will be handled by the ScreenController independent of the keyboard focus of any other elements on the screen. You can think of screen level input events as global events. The perfect example would be the possibility to cancel a screen with the ESC key. This kind of events should be handled by screen level events.

There are two steps necessary to make this work. The first step is to implement the KeyInputHandler interface in your ScreenController class. The Nifty KeyInputHandler interface is very simple and looks like this:

```
public interface KeyInputHandler {
  /**
   * handle a key event.
   * @param inputEvent key event to handle
   * @return true when the event has been consumed and false if not
   */
  boolean keyEvent(NiftyInputEvent inputEvent);
}
```

The second step is to actually enable input processing at the screen level by providing the inputMapping or inputMappingPre attribute. You have to provide a NiftyInputMapping implementation. This will enable screen level keyboard input.

When Nifty processes a keyboard event it will first use the inputMappingPre class (if provided) to convert the keyboard event into a NiftyInputEvent. The NiftyInputEvent is then sent to the ScreenController.

Next the keyboard event is being sent to all elements that have the keyboard focus.

And as the last step the inputMapping is used (if provided) to convert the Keyboard event into a NiftyInputEvent and is then sent to the ScreenController as well.

This way you can decide if you want to process keyboard events before or after the regular processing using inputMapping or inputMappingPre. Processing of the keyboard event is stopped when a KeyInputHandler returns true.

KEYBOARD EVENTS FOR INDIVIDUAL ELEMENTS

Handling Keyboard events at the element level requires that the attribute focusable is set to true for the element. This way the element can get the keyboard focus which is required for the element to get keyboard events. The controller and the inputMapping attributes have to be set as well (although Nifty will use it's DefaultInputMapping class if you don't specify one).

With all of these things in place Nifty will call the inputEvent method of the controller when the element has the keyboard focus.

Please note that when the NiftyInputMapping does not map the key the inputEvent is null.

You usually handle keyboard events in a custom control implementation and not in single elements. This means that you would build your own control, let's say a text input field as a custom control and all keyboard input handling will be performed in the control. The chapter on Nifty controls will explain all of the details on how you can create your own control.

NIFTY EVENT CONSUMING AND DISABLING EVENT PROCESSING (NIFTY 1.3.2)

Events processed by Nifty will be marked internally as consumed. Such events will usually not be forwarded by Nifty InputSystem implementations to other parts of your application (e.g. your game below a GUI overlay).

Sometimes however it is desired to completely disable event processing by Nifty so that all events will be passed to your application. Nifty 1.3.2 added support for this for all events and for individual elements.

DISABLE EVENT PROCESSING GLOBALLY

Use `nifty.setIgnoreMouseEvents(true)` to completely disable mouse event processing in Nifty. The NiftyInputConsumer.processMouseEvent() method will always return false in that case. The Nifty InputSystem implementation should forward all of these mouse events to your game.

Use `nifty.setIgnoreKeyboardEvents(true)` to completely disable keyboard event processing in Nifty. The NiftyInputConsumer.processKeyboardEvent() method will always return false in that case as well.

DISABLE EVENT PROCESSING FOR INDIVIDUAL ELEMENTS

The new methods to ignore mouse events and keyboard events at the global Nifty level have now been added at the element level as well. You can now ignore mouse and keyboard events for individual elements too. You simple use the same method names `setIgnoreMouseEvents(true)` and `setIgnoreKeyboardEvents(true)` for a single Element and this element will no longer process any elements.

> Please note that this is a bit redundant to the existing visibleToMouseEvents attributes but we've kept the same names to be consistent with the method on the main Nifty instance. The `visibleToMouseEvents` attribute should be used to generally define visibility to mouse events and ignoreMouseEvents / ignoreKeyboardEvents to temporarily disable access to the element.

EVENTBUS EVENTHANDLING

INTRODUCTION

There is a different way to handle events that works without the <interact> element. Instead of registering events for each and every element/control Nifty utilizes the EventBus mechanism.

The EventBus mechanism supports loose coupling between Nifty (as the creator of events) and your application (as the receiver of events) using a publish/subscribe mechanism. Nifty publishes events to some "global" EventBus and your application subscribes to specific Events that it is interested in.

What sounds not very different from the standard Listener pattern is in fact a great improvement. There is only a dependency to the EventBus and not between the actual objects that communicate. This helps to decouple the objects from one another. Nifty does not need to know who will receive the event in the end. It just creates the event, publishes it on the EventBus and everybody interested in it will be notified automatically.

Nifty uses the EventBus project (http://www.eventbus.org/) for this. The project is available as Open Source under the Apache License, Version 2.0. It's only 80KB in size, so it should not hurt the download/memory footprint of Nifty that much.

You subscribe to the EventBus using the id of the element as the topic. So it's important that all the elements you want to be notified about have the id attribute set.

When an event occurs Nifty will generate a new event class which implements the NiftyEvent marker interface:

```
public interface NiftyEvent<T> {
}
```

So whenever an event occurs Nifty calls you (via the EventBus) with a specific event class that implements the NiftyEvent interface. Typically you'll use methods of the specific event class to request details of the event. This chapter will list all specific event classes and when they are being published by Nifty.

There exist several mechanism to register for events with Nifty. Some of them use Java annotations to make this very easy. This chapter will explain how this works.

SUBSCRIBE FOR NIFTYEVENTS

We start again with the screen that contains a single image and since we'll try to subscribe for mouse click events we'll make sure to set visibleToMouse="true". To subscribe for events it's necessary that the element has an id attribute set, so we'll set id="imageId" as well.

Please note that in the following XML there is no <interact> element necessary!

```
<screen id="start" controller="my.stuff.StartScreen">
  <layer id="layer" childLayout="center">
    <image id="imageId" filename="nifty-logo-150x150.png" visibleToMouse="true" />
  </layer>
</screen>
```

The ScreenController „my.stuff.StartScreen“ is for now just an empty implementation of the ScreenController interface.

USING THE @NIFTYEVENTSUBSCRIBER ANNOTATION

This is probably the simplest way to register for events with Nifty. It uses the @NiftyEventSubscriber annotation with the id of the element we'd like to receive events for. The annotation will work with single methods. The annotated method is required to have exactly two parameters:

- The first parameter must be a String and will contain the elementId that has published the event.

- The second parameter will be an instance of a class that implements the NiftyEvent interface.

EXAMPLE

Let's add the following method to the ScreenController class my.stuff.StartScreen:

```
@NiftyEventSubscriber(id="imageId")
public void onClick(String id, NiftyMousePrimaryClickedEvent event) {
  System.out.println("element with id [" + id + "] clicked at [" + event.getMouseX() +
                     ", " + event.getMouseY() + "]");
}
```

And that's all there is to do! :) The onClick() method will be called when the element is clicked.

The @NiftyEventSubscriber annotation will use the id you set to subscribe to all events the element with the given id generates that match the second parameter. In the example it will only subscribe to the NiftyMousePrimaryClickedEvent. To do this it will look at the annotated method signatures second parameter.

Sometimes you want to register the same method for several different elements. The @NiftyEventSubscriber annotation offers a second syntax for this. Instead of using a single id you can set the pattern attribute which is a regular expression matching the id of all the elements you want to subscribe to.

EXAMPLE

```
@NiftyEventSubscriber(pattern="im.*")
public void onClick(String id, NiftyMousePrimaryClickedEvent event) {
  System.out.println("element with id [" + id + "] clicked at [" + event.getMouseX() +
                     ", " + event.getMouseY() + "]");
}
```

The onClick method will be called for all elements with an id that starts with „im“.

Using the @NiftyEventSubscriber annotation is automatically supported in all ScreenController and Controller implementations. This means that all you need to do is to add an annotation to a method and Nifty will automatically subscribe/unsubscribe all matching methods for you.

USING THE @NIFTYEVENTSUBSCRIBER ANNOTATION IN ANY CLASS

To use the @NiftyEventSubscriber annotation in a class that is not a Nifty ScreenController or a Nifty Controller implementation but just any other class there is an additional step required.

You'll need to call nifty.subscribeAnnotations() with an instance of your class. Nifty will then scan the given instance for all @NiftyEventSubscriber annotations and subscribe all of them with the EventBus.

If you don't need your object anymore you should call nifty.unsubscribeAnnotations() with the instance again to unsubscribe all annotated methods.

This is how the methods look like and you can find them on the Nifty instance:

```
// call this to subscribe all annotated methods of the given object
public void subscribeAnnotations(Object object);

// call this to unsubscribe all annotated methods of the given object
public void unsubscribeAnnotations(Object object);
```

SUBSCRIBE DIRECTLY FOR EVENTS WITHOUT ANNOTATIONS

If for some reason you can't use annotations you can directly subscribe for Nifty events using a special subscribe() method.

You'll need to directly implement the org.bushe.swing.event.EventTopicSubscriber interface:

```
public interface EventTopicSubscriber<T> {
   public void onEvent(String topic, T data);
}
```

where T is a specific NiftyEvent implementation.

And you'll need to call a method on the Nifty instance to subscribe for the event:

```
public <T, S extends EventTopicSubscriber<? extends T>> void subscribe(
   Screen screen,
   String elementId,
   Class<T> eventClass,
   S subscriber);
```

To unsubscribe for the event again you'll need to call the unsubscribe method with the same class that you've subscribed before (the org.bushe.swing.event.EventTopicSubscriber implementation).

```
public void unsubscribe(String elementId, Object object);
```

Everything else works exactly the same.

NIFTYEVENT REFERENCE

The following reference lists all available EventBus events that Nifty publishes. All of the listed classes are implementations of the marker interface NiftyEvent.

ELEMENT BASED EVENTS

These events will be published by Nifty when element state changes. All classes are in the de.lessvoid.nifty.elements.events package.

Classname	Event
ElementDisableEvent	Published when an element will be disabled.
ElementEnableEvent	Published when an element will be enabled.
ElementHideEvent	Published when an element will be hidden.
ElementShowEvent	Published when an element will be shown.
de.lessvoid.nifty.elements.Element	Published when an element changed layout specific properties and the Element class itself is published as the NiftyEvent in this case.

MOUSE BASED EVENTS

These events will be published by Nifty when a mouse event occurs. All classes are in the de.lessvoid.nifty.elements.events package.

Classname	Event
NiftyMouseEvent	Published for any mouse event. The NiftyMouseEvent has getters for the individual attributes.
NiftyMouseMovedEvent	Published when the mouse cursor has been moved. The class has getters for the current mouse coordinates.
NiftyMouseWheelEvent	Published when the mouse wheel position has been changed.
NiftyMousePrimaryClickedEvent	Published when the primary (usually the left mouse button) has been pressed.
NiftyMousePrimaryClickedMovedEvent	Published when the primary (usually the left mouse button) has been pressed and the mouse is being moved.
NiftyMousePrimaryReleaseEvent	Published when the primary (usually the left mouse button) has been released.
NiftyMouseSecondaryClickedEvent	Published when the secondary (usually the right mouse button) has been pressed.

Classname	Event
NiftyMouseSecondaryClickedMovedEvent	Published when the secondary (usually the right mouse button) has been pressed and the mouse is being moved.
NiftyMouseSecondaryReleaseEvent	Published when the secondary (usually the right mouse button) has been released.
NiftyMouseTertiaryClickedEvent	Published when the tertiary (usually the middle mouse button) has been pressed.
NiftyMouseTertiaryClickedMovedEvent	Published when the tertiary (usually the middle mouse button) has been pressed and the mouse is being moved.
NiftyMouseTertiaryReleaseEvent	Published when the tertiary (usually the middle mouse button) has been released.

INPUT EVENTS

When an element is enabled to retrieve keyboard events the converted NiftyInputEvent itself is published using the id of the element.

STANDARD CONTROLS EVENTS

All of the standard controls publish events. The following table lists all the NiftyEvent classes that are currently available for the standard controls. All classes are in the de.lessvoid.nifty.controls package.

This time the classes are only listed and they should be self explaining. The complete reference is available in the wiki at

http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=Nifty_Standard_Controls_%28Nifty_1.3%29.

Classname	Classname
ButtonClickedEvent	ChatTextSendEvent
CheckBoxStateChangedEvent	ConsoleExecuteCommandEvent
DraggableDragCanceledEvent	DraggableDragStartedEvent
DropDownSelectionChangedEvent	DroppableDroppedEvent
FocusGainedEvent	FocusLostEvent
ImageSelectSelectionChangedEvent	ListBoxSelectionChangedEvent
MenuItemActivatedEvent	RadioButtonGroupStateChangedEvent
RadioButtonStateChangedEvent	ScrollPanelChangedEvent
ScrollbarChangedEvent	SliderChangedEvent

Classname	Classname
TabSelectedEvent	TextFieldChangedEvent
TreeItemSelectedEvent	

GENERAL MOUSE EVENT PROCESSING CHANGES WITH NIFTY 1.3.2

When you use the <interact /> callback methods like <interact onClick="..." /> everything works as expected: Mouse events will travel down all elements that are visibleToMouse="true" until some element has been found with an <interact /> event handler. When the event handler is found this mouse event is consumed and is not send to any other elements "below" this element.

However there is a problem with this approach especially with EventBus events. EventBus events are published and because of their very nature are not able to provide any information if the event has been consumed by anybody or not at all.

This is a problem when you imagine several elements overlayed, like a button on a panel and you have a <interact onClick="..." /> event handler on the panel but you're using the Eventbus ButtonClickedEvent for the button. In this case Nifty processes the mouse click event for the button first, because it is "above" the panel. The button will publish its click event on the EventBus and then the event will be forwarded to the panel! Since the panel has a <interact onClick="..." /> event handler this event handler will be executed as well.

The problem is that we've now executed two event handlers (for the button and the panel) even though from an intuitive point of view we've expected that the button should have consumed the event.

Because of this problem Nifty 1.3.2 has changed the general event processing as follows:

To keep things simple Nifty will now stop processing directly when it encounters the first visibleToMouse="true" element. This means events will not travel any further when a visibleToMouse="true" element has been detected. When you have two elements above each other that both have visibleToMouse="true" then only the topmost element will now get the event.

Please Note:

It might be necessary to change your code slightly! Especially you must be careful where you add visibleToMouse="true" because this can now really block events from traveling down the element tree.

EFFECTS

INTRODUCTION

Now that you have all of your elements arranged nicely on the screen and you know how you can interact with them it's time to make your Nifty GUI really nifty with the use of effects.

A Nifty effect is basically just some change of a render state before or after an element is rendered. Additionally effects are time aware and can change the render state over time.

Here is an example XML that just displays an image.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <screen id="start">
    <layer childLayout="center">
      <image filename="nifty-logo-150x150.png" />
    </layer>
  </screen>
</nifty>
```

And we get the Nifty Logo centered in the middle of the screen as the output.

Now we change the XML and add an effect to the image to make it fade in when the screen starts.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <screen id="start">
    <layer childLayout="center">
      <image filename="nifty-logo-150x150.png">
        <effect>
          <onStartScreen name="fade" />
        </effect>
      </image>
    </layer>
  </screen>
</nifty>
```

And now the image slowly fades in when the screen is started. The „fade“ effect will by default change the alpha value of anything it is applied to from 0% to 100% over the time of one second. This, however, are only default values which we can easily change:

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <screen id="start">
    <layer childLayout="center">
      <image filename="nifty-logo-150x150.png">
        <effect>
          <onStartScreen name="fade" start="#f" end="#0" length="15000" />
        </effect>
      </image>
    </layer>
  </screen>
</nifty>
```

Now the image will be initially visible but it will fade out over 15000ms which are 15 seconds. After the 15 seconds the effect will be disabled because it has ended and it it will be removed so that the image will be fully visible again. If we want the effect to be applied even when it has ended we can add neverStopRendering=“true“ and the effect will stay active with its last value hiding the image forever.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <screen id="start">
    <layer childLayout="center">
      <image filename="nifty-logo-150x150.png">
        <effect>
          <onStartSceeen name="fade" start="#f" end="#0" length="15000"
                         neverStopRendering=“true“/>
        </effect>
      </image>
    </layer>
  </screen>
</nifty>
```

EFFECT EVENTS

You can attach effects to any element. There are the following events that can trigger an effect.

onStartScreen
The effect is started when the screen the element is a part of begins.

onEndScreen
The effect is started when the screen the element is part of ends.

onFocus
When the element is able to get the keyboard/input focus then this effect is active as long as the element has the keyboard focus.

onGetFocus
The element just got the keyboard/input focus.

onLostFocus
The element just lost the keyboard/input focus.

onClick
The element has been clicked by the mouse or is activated by a keyboard interaction.

onHover
The mouse cursor is currently hovering the element.

onStartHover
The mouse cursor just began hovering the element.

onEndHover
The mouse cursor just left the element.

onActive
The element has been initialized and is now ready and active.

onCustom
This effect can only be started from Java and will not be started by Nifty. You'll need to manually trigger the effect (Explained below).

onHide
The element is about to be hidden.

onShow
The element is shown again (after first being hidden).

onEnabled
The element is enabled.

onDisabled
The element is disabled.

HOVER EFFECTS

Hover effects are a special kind of an effect. Additional to the normal effect parameters they can take the distance of the mouse cursor into account and change effect parameters accordingly. To specify the additional parameters there is a nested <hover> element you can apply to the onHover effect.

All effects can be used as hover effects. However not all effects really support the hover mode.

EXAMPLE

Here is an example of the hover effect in action.

```
<onHover name="textSize" maxSize="120%">
  <hover hoverFalloffType="linear"
         hoverFalloffConstraint="both"
         hoverWidth="200%"
         hoverHeight="200%" />
</onHover>
```

So when you apply this to a text element and you move your mouse around the text element then the textSize effect will dynamically change the size of the text in an area 200% around the element and with a maximal size of 120% of the original text size.

Please note: Elements in Nifty don't receive mouse events by default. To get the above example working you'll need to set "visibleToMouse" to „true“ on the element you apply that effect to or you won't see any change at all :)

Here is an explanation of all the available hover effect attributes:

„hoverFalloffType“ one of „none“ or „linear“, Default: „none“
The falloff type to use. When being set to „none“ hover is actually disabled and „linear“ will take the linear distance between the center of the element the effect is applied to and the mouse position into account.

„hoverFalloffConstraint“ one of „none“, „vertical“, „horizontal“ or „both“, Default: „none“
You can constraint the falloff of the hover effect to only „vertical“ or only „horizontal“ which means that only the specified axis is being taken into account. With „none“ the falloff is disabled.

„hoverWidth“ as a Nifty SizeValue, Default: not set
You can specify the width of the hover area as a Nifty SizeValue. This means you can specify the width as a pixel value (adding „px“ to the value) or as a percent value (adding „%“ to the value).

„hoverHeight“ as a Nifty SizeValue, Default: not set
You can specify the height of the hover area as a Nifty SizeValue. This means you can specify the height as a pixel value (adding „px“ to the value) or as a percent value (adding „%“ to the value).

MANUALLY STARTING EFFECTS

Most of the effect events are started automatically by Nifty when certain events occur. But sometimes you want or need to trigger some effects manually.

This is possible using methods that the Element class provides. The simplest one looks like this:

```
public void startEffect(final EffectEventId effectEventId);
```

The EffectEventId is an enum that describes all the possible events. You can use the startEffect() methods like this:

```
Element element = screen.findElementByName("elementId");
element.startEffect(EffectEventId.onStartScreen);
```

There is another version of startEffect() available that takes a second parameter.

```
public void startEffect(EffectEventId effectEventId, EndNotify effectEndNotiy);
```

The EndNotify is a simple callback interface which Nifty calls when the effect has ended. By implementing this interface you can execute some code when the effect you've just started ends.

The EndNotify is very simple and it consists of only a single perform() method without any parameters and with no return value.

```
public interface EndNotify {
  void perform();
}
```

And there is yet another version of startEffect() available that is especially useful for the onCustom effect. You can attach multiple onCustom effects to an element. When you would use the regular startEffect() method then all of the onCustom effects will be activated.

But sometimes you need multiple custom effects and you want to start the individual custom effects under different circumstances. So in that case you need a way to select which one of the custom effects you want to start.

To do this you can set the „customKey" parameter for the effect and using the startEffect() method that takes as a third parameter the „customKey". Nifty will find all the effects with a matching „customKey" and will only start the effects that match.

```
public void startEffect(EffectEventId effectEventId,
                        EndNotify effectEndNotiy,
                        String customKey);
```

EXAMPLE

Here we demonstrate the onCustom effect. We've attached two onCustom effects to an image. One of onCustom effects fades the image in and the other one fades the image out. To achieve that we'll add customKey="fadeIn" to one of the effects and customKey="fadeOut" to the other.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <screen id="start"
controller="de.lessvoid.nifty.examples.helloworld.HelloWorldStartScreen">
    <layer childLayout="center">
      <image id="imageId" filename="nifty-logo-150x150.png">
        <effect>
          <onCustom customKey="fadeIn" name="fade" start="#0" end="#f" />
          <onCustom customKey="fadeOut" name="fade" start="#f" end="#0"
neverStopRendering="true" />
        </effect>
      </image>
    </layer>
  </screen>
</nifty>
```

The ScreenController for the screen looks like this:

```
public class HelloWorldStartScreen implements ScreenController {
  private Nifty nifty;
  private Element image;

  @Override
  public void bind(final Nifty newNifty, final Screen newScreen) {
    this.nifty = newNifty;
    this.image = newScreen.findElementByName("imageId");
  }

  @Override
  public void onStartScreen() {
    image.startEffect(EffectEventId.onCustom, new FadeInEnd(), "fadeIn");
  }

  @Override
  public void onEndScreen() {
  }

  class FadeInEnd implements EndNotify {
    @Override
    public void perform() {
     System.out.println("fadeIn has ended.");
     image.startEffect(EffectEventId.onCustom, new FadeOutEnd(), "fadeOut");
    }
  }

  class FadeOutEnd implements EndNotify {
    @Override
    public void perform() {
      System.out.println("fadeOut has ended.");
    }
  }
}
```

So there are some interesting things happening in here. In the bind() method of the ScreenController we‘ll lookup the image element by it‘s id attribute in the XML (id=“imageId).

Then in the onStartScreen() method we will start the onCustom effect with the customKey „fadeIn“ and we will add a FadeInEnd instance as the EndNotify so that Nifty can call us back when the fadeIn effect has ended.

This will make the image fade in as specified in the XML file and when the fade in has finished Nifty will call the perform() method of FadeInEnd. We output a string to the console and then start a new custom effect. This time we use „fadeOut“ as the customKey and call the startEffect method with a FadeOutEnd instance so that Nifty can call us back when the „fadeOut“ effect has ended.

This will make the image fade out and in the perfom() method of the FadeOutEnd we‘ll simply output another string to the console.

EFFECT PARAMETERS

There are some effect parameters that change the way in how the effect is applied to a given element. We‘ll explain all of them in this section.

„startDelay“ in ms, Default: 0
Whenever an effect gets active it immediately is being applied by default. But sometimes you want to delay the effect start for a bit so that the effect better synchronizes with other things or you simply want to pause it for a while. In this case you can set the "startDelay" attribute of the effect to some integer value that represents the time in ms to delay the start of the effect.

„length“ in ms, Default: 1000
How long the effect lasts depends on the "length" parameter. The standard value is one second (1000ms). You can set the „length“ parameter to any value you want. The special value "infinite" is also supported so that the effect never ends.

„oneShot“ as a boolean, Default: false
If you want an effect to only run for a single time when it gets activated you should set "oneShot“ to „true".

„timeType“ one of "infinite", "linear" or "exp", Default: „linear“
Time in Nifty is only by default "linear". For special effects you can also use "exp" time which calculates time^value. For instance when you set the "factor" attribute to 2 this would mean time^2.

You can achieve infinite effects by setting timeType to "infinite" too. This is the same as setting length to "infinite".

„factor“ as a number, Defaut: 1
When using timeType="exp" you can use the "factor" attribute to specify the factor, f.i. factor="1.5" would calculate time^1.5.

„inherit“ as a boolean, Default: false
Determines if the effect is only applied to the element that contains the effect (inherit="false") or if the effect is applied to all child elements as well (inherit="true"). For example a "move" effect with inherit="true" would also move all child elements.

„post“ as a boolean, Default: false
Effects are applied before an element is rendered when post is set to "false" which is the default. This way the effect can modify some state, f.i. the current alpha value and when the element gets rendered this modified state is applied. There are however effects that are better applied after the element is rendered. A perfect example would be a color overlay. This effects should be applied after the element is rendered by setting post to "true".

„overlay“ as a boolean, Default: false
The "overlay" Attributes is used to render an effect after all child elements are rendered. If you would use a "post" effect, the effect would be applied after the element has been rendered but before the child elements are rendered. So f.i. an "imageOverlay" effect would be overwritten by child elements. With overlay=“true" you can add special effects after the element and the children have been rendered.

„alternateEnable“ as a String, Default: null
This is a very powerful attribute. You can attach a tag to an effect and tell Nifty that this effect is only enabled when the tag you set matches the alternateKey you set for the whole screen. When you start a Screen from Java there is an option to set this alternateKey tag. Nifty will filter all effects that match this tag and will activate the effect only when the tags match.

„alternateDisable“ as a String, Default: null
This works the same as "alternateEnable" but disables the effect when the tag set on the effect matches the global screen tag.

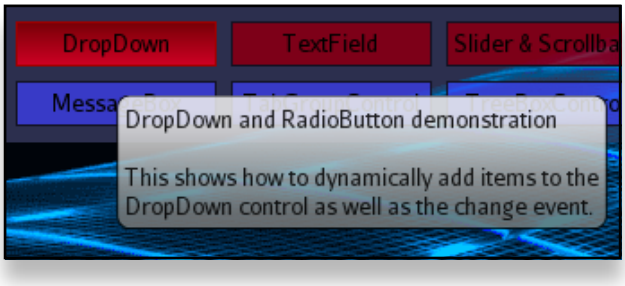
„onStartEffect“ as a callback, Default: null
For special effects you can attach a callback to the effect that is executed when the effect is about to be started. The callback is resolved with the current ScreenController.

„onEndEffect“ as a callback, Default: null
For special effects you can attach a callback to the effect that is executed when the effect has ended. The callback is resolved with the current ScreenController.

„neverStopRendering“ as a boolean, Default: false
Keep rendering this effect rendered even when the actual effect time is over.

DYNAMICALLY CHANGE EFFECT PARAMETER

Sometimes you want to use the same effect but tweak some of the effect parameters slightly. The perfect example for this would be the „hintEffect“.



The „hintEffect“ exposes a „hintText“ property that defines the text of the hint. This is fine as long as the text is static and you can specify it in XML for instance. But what if you need to change the hint text?

You can do that by changing the effect properties before you display the effect:

```
// Get the element that has the effect attached.
Element element = nifty.getCurrentScreen().findElementByName("test");

// Get all effects of the element. This is a list since there could be
// multiple effects attached for the given EffectEventId and effect class.
List<Effect> hoverHintEffects = element.getEffects(EffectEventId.onHover, Hint.class);

// this is probably not necessary since you know what element and effect you request
if (hoverHintEffects.size() != 1) {
  throw new RuntimeException("sanity check failed");
}

// Simply get the first effect from the list
Effect hoverEffect = hoverHintEffects.get(0);

// and finally change the Effectparameters
hoverEffect.getParameters().setProperty("hintText", "change me");
```

When the effect will be activated the next time it will use the new parameters.

EFFECTS REFERENCE

You can find a reference of all effects including examples online in the Nifty wiki at

http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=Effects.

CUSTOM EFFECTS

You're not limited to the build-in effects. You can easily define your own effects.

When you apply an effect in Nifty the name of an effect connects the effect to its implementing class. For all build-in effects there is a class inside of Nifty that registers all of the default effects. The class is called de.lessvoid.nifty.NiftyDefaults and you‘ll find a lot of lines like the following one in this class:

```
nifty.registerEffect(new RegisterEffectType(
      "alphaHide", "de.lessvoid.nifty.effects.impl.AlphaHide"));
... // more lines like this
```

This will link the effect name „alphaHide“ to the implementing class for the effect „de.lessvoid.nifty.NiftyDefaults“.

So creating your own effect is a two step process. First you‘ll need to implement the Nifty de.lessvoid.nifty.effects.EffectImpl Interface and then you‘ll need to define a name for your new effect and register it with the <registerEffect> XML element in your Nifty XML:

```
<registerEffect name="my-cool-new-effect" class="my.package.MyCoolNewEffect" />
```

When you don‘t use XML you can do the same with the Nifty instance:

```
nifty.registerEffect("my-cool-new-effect", "my.package.MyCoolNewEffect");
```

With the effect registered with Nifty you can use it like the build-in effects with name="my-cool-new-effect" when applying any of the effects.

The EffectImpl interface you need to implement is not really complicated. It looks like this:

```
public interface EffectImpl {

  /**
   * initialize effect.
   * @param nifty Nifty
   * @param element Element
   * @param parameter parameters
   */
  void activate(Nifty nifty, Element element, EffectProperties parameter);

  /**
   * execute the effect.
   * @param element the Element
   * @param effectTime current effect time
   * @param falloff the Falloff class for hover effects.
   * This is supposed to be null for none hover effects.
   * @param r RenderDevice to use
   */
  void execute(Element element, float effectTime, Falloff falloff,
               NiftyRenderEngine r);

  /**
   * deactivate the effect.
   */
  void deactivate();
}
```

Nifty calls the activate() method when the effect starts and deactivate() when the effect ends.

The execute() method is called each frame to actual render the effect. In the execute() method you have access to the element the effect is attached to as well as the effectTime, which runs from 0.0 to 1.0 and the NiftyRenderEngine. The NiftyRenderEngine allows you to modify render states in your effect to modify how the element gets drawn. The falloff parameter gives you access to the hover effect parameters.

You can render anything in execute. What about a particle effect? =)

RUNTIME ELEMENT MODIFICATION

INTRODUCTION

This chapter explains how you can access Nifty elements from Java and modify properties of your GUI at runtime. If you access and modify the standard controls you'll usually use the dedicated control API that the standard controls provide. How this works is explained in the Controls chapter.

Here we concentrate on how you can access and modify the core elements (panel, image, text) as well as how you can dynamically modify the layout of elements in general.

ACCESS ELEMENTS

Before you can modify elements you need to access them. At the root of every Nifty GUI is the screen so we'll need to first access the screen.

If you don't have a Screen instance already there are a couple of methods available at the Nifty instance to access Screens:

```
// get the current (active) screen
public Screen getCurrentScreen();

// get a screen with the given id
public Screen getScreen(final String id);

// request all ids of all screens currently registered with Nifty
public Collection < String > getAllScreensName();
```

Another way is the bind() method of the ScreenController or the Controller interface. Nifty will call you with the Screen instance that the ScreenController or Controller is being bound to. You can keep that Screen instance around in your implementation of those interfaces.

Once you have a Screen instance you can begin accessing it's content.

For instance you can get a list of all layer elements or you can get a reference to individual elements using the id of the element. You can access layer elements by id as well:

```
// get all layers
public List<Element> getLayerElements();

// get an element by id. this returns null when the Element could not be found
public Element findElementByName(String name);
```

So you'll notice that Nifty treats everything as a Nifty Element. This is an important concept to understand the way Nifty works internally:

Every core element (Layer, Panel, Image, Text) is treated equally by Nifty as an Element. The difference is only in the way the core element will get rendered on the screen. Which we will talk about in a minute.

First we'll take a look at how we can request and change common Element properties of an Element.

REQUEST ELEMENT PROPERTIES

You can request element properties by simply calling a getter method on the Element class. Here are some selected interesting properties:

getX(), getY(), getWidth(), getHeight()
These methods give you the coordinates of the element on the screen. These are the resolved values after layout has been performed. You get the coordinates in absolute pixel coordinates (0,0 is the top left corner of the screen).

getStyle()
You can access the name of the style that has been applied to this element.

getParent()
Every element has exactly one parent element. With getParent() you can access the parent Element.

getNifty()
This is a helper method that gives you easily access to the Nifty instance this Element is attached to.

getId()
Returns the id of the element which can be handy sometimes.

getElements()
Return all child elements of this element.

getEffects()
This is another helper method to access all of the effects attached to this Element. With it you can modify effect attributes if necessary. You'll need to provide the EffectEventId you are interested in as well as the class of the EffectImpl you want to access and the method will give you the EffectImpl back.

getControl()
If you have a reference to an Element which is actually a control you can use the getControl() method to access the Controller of the Element.

getNiftyControl()
If you have a reference to an Element which is actually a standard control (or any control that implements the NiftyControl interface) you can use the getNiftyControl() method to access the control API interface of the control. This is the preferred method to access the standard control API of a standard control.

getConstraintX(), getConstraintY(), getConstraintWidth(), getConstaintHeight()
getConstraintHorizontalAlign(), getConstraintVerticalAlign()
As explained in the Nifty Layout chapter layout in Nifty works by setting a couple of constraints on the Element which the layout mechanism tries to satisfy. So when you set width="50%" for an Element this is actually a width constraint for the element. With these getters you can access the original values of the constraints. The return values are SizeValues or the appropriate enums for Horizontal- and VerticalAlign.

isClipChildren()
You can check if this element will clip the content of it's child element at this elements boundary. This corresponds to the value of the childClip attribute.

isEffectActive()
You can ask the element if a certain EffectEventId is still active.

isEnabled(), isVisible(), isFocusable(), isVisibleToMouseEvents()
You can request the states of enabled, visible and so on.

You can find out more about these methods in the JavaDoc for the Nifty Element class.

MODIFY ELEMENT PROPERTIES

The Element class has some methods available that we can call directly to modify some properties.

disable() and enable()
You can enable and disable elements with a simple call to these methods. Nifty will ensure that the element can not be interacted with if it is disabled. It will also start the appropriate effects and it will make sure to shift the focus to the next element when the element you disable() has the keyboard focus currently.

hide() and show()
You can change the visibility of an Element with the hide() and show() methods. Please keep in mind that hiding an Element will not remove it from the layout (e.g. it will still take up place!). If you want to remove the Element you'll need to remove it from the screen. This is explained further below.

Especially hide() supports an optional EndNotify callback which will be called when the Element has been completely hidden. This is sometimes important when you want to trigger other actions and you have an onHide effect applied to the element and you want to trigger the action when the element has been completely hidden.

setFocus()
If the element is focusable then you can manually make it active by calling setFocus() on the element.

setFocusable()
You can change if the element can get the keyboard focus with this method.

setId()
If necessary you could change the id attribute dynamically.

startEffect(), stopEffect()
You can start and stop effects dynamically as well. This is explained in the Effect chapter in detail.

MODIFY LAYOUT

Since Nifty has done the layout of your screen you can't move elements freely around. This is why there is no direct setX(), setY(), setWidth() or setHeight() method available. What you can do is change the constraints and then tell Nifty to relayout the whole screen or individual elements which we will look into in a minute.

There are a couple of setConstraint*() methods available to change constraints:

setConstraintX(), setConstraintY()
You can change the x and y coordinates of an element only when the parents childLayout supports direct x and y coordinates. Currently these are „absolute" and „absolute-inside". Please note that the method take SizeValue as a parameter so that they can allow % values as well.

setConstraintWidth(), setConstraintHeight()
Change the width and height constraints. These are SizeValues again to allow % values.

setConstraintHorizontalAlign(), setConstraintVerticalAlign()
Change the horizontal or vertical alignment values. Again this will only work for childLayouts that support alignment and needs a relayout after you've changed the values.

After you've changed the constraints you'll need to tell Nifty to take your changes and apply them to the screen. You can change several constraints at once and then apply them all.

There are two ways to do that:

1. You can call screen.layoutLayers()
   This will go through all layers of the screen and adjust all elements according to the changed constraints.

2. You can call element.layoutElements()
   This will go to all child elements of the element you've called that method on and only these child elements (and all of their child elements) will be laid out.

The difference is that in the 1. case all elements are being relayout which might take a bit longer then when the layout is applied to only a subset of the element tree in the 2. case.

Please note that you'll need to call layoutElements() for the parent element of the element you've changed constraints or your change will have no effect. So when you are unsure or you've changed a lot of elements it is probably safer to call screen.layoutLayers().

Future Nifty version might optimize this so it's not necessary to call layoutLayers() or layoutElements() manually but at the moment this is still a required step.

MOVE ELEMENTS TO ANOTHER PARENT

Sometimes it is necessary that you move elements to another parent element. In that case you'll need to call a method on the element which you want to move:

```
// mark this element for being moved to the given destination element
public void markForMove(Element destination);

// same as above but call a method when the element has been moved
public void markForMove(Element destination, EndNotify endNotify);
```

You can't directly move the element because at the point in your code you call this method Nifty might still be traversing the element tree. This is why you can only mark the element to be moved and Nifty will take care of this when it's ready.

If you need to perform an action when the element has been moved you can use the second version of the method which allows you to provide an EndNotify implementation.

REMOVE ELEMENTS

Removing elements works the same as moving them to a new parent. You can't remove them directly but you can mark them for being removed later when Nifty is ready with processing the Element tree.

```
// mark this element for being removed
public void markForRemoval();

// same as above but call a method when the element has been removed
public void markForRemoval(EndNotify endNotify);
```

When the element is being removed its onEndScreen effect is activated. When the onEndScreen effect ends it is finally removed from the screen.

CHANGE PANEL, IMAGE AND TEXT PROPERTIES

Panel, Image and Text only exists as Element instances. So there is no Panel or Image or Text class in Nifty. But there are some attributes that are only present for panel, image or text. For instance a font attribute makes only sense for the text element.

The way Nifty handles this case is that every Element contains a set of ElementRenderers. ElementRenderer implementations define the way how a specific element is being rendered on the screen. Currently Nifty has an Image-, Panel- and TextRenderer which match the core elements for Image, Panel and Text.

If you need to change attributes that are specific for these elements then you will need to access the specific ElementRenderer implementation from an Element and use it to change the attributes.

Accessing the specific ElementRenderer is supported by the Element class with a special method:

```
// get a specific ElementRenderer from an element
public <T extends ElementRenderer> T getRenderer(Class <T> requestedRendererClass);
```

So for instance if you need to change the image of an image element, you'll first request the element from the screen using the id of your element and then you use the getRenderer() method to access the ImageRenderer where you can simply set a new NiftyImage:

```
// find the element
Element imageElement = screen.findElementByName("imageId");

// get the ImageRenderer
ImageRenderer imageRenderer = imageElement.getRenderer(ImageRenderer.class);

// change the image
imageRenderer.setImage(nifty.getRenderEngine().createImage("new-image.png", false);
```

Accessing and changing the other ElementRenderers works exactly the same. You can find all of the ElementRenderer implementations in the de.lessvoid.nifty.elements.render package.

NIFTY STYLES

PRINCIPLES

If you have lots of elements on your screen that all share some common attributes, then Nifty styles are a way to reduce the duplication and make your GUI definition more manageable.

Here is an example. Without style definitions your Nifty XML file might look like this.

```
<text font="myfont.fnt" backgroundColor="#f00f" text="stuff 1" />
<text font="myfont.fnt" backgroundColor="#f00f" text="stuff 2" />
```

So we have two text elements using the same font and the same backgroundColor. Having the same attributes in several different places makes changes to the attributes difficult. If I want a green background instead or a different font for all of the text elements I'd need to find all the occurrences of the backgroundColor and font attributes and set a new value for all of them.

Of course in this example I could simply use a search and replace feature of my text or XML editor but when I have lots of elements I might not be able to easily solve the problem this way.

With Nifty style definitions we can extract all of the attributes that we need to use multiple times and define them just once. This definition is called a style in Nifty and it can be applied to elements where we need the attributes. This way Nifty styles are a great way to organize complex GUI definitions into more manageable components.

For our example with the two text elements we'd like to extract the font and the backgroundColor attribute into a style. To define the style we use the <style> XML element and we need to give the style definition a name with the "id" attribute of the <style> element.

```
<style id="redBackgroundCaption">
 <attributes font="myfont.fnt" backgroundColor="#f00f" />
</style>
```

As always we can do the same using the Java Builder pattern.

```
new StyleBuilder() {{
  id("redBackgroundCaption");
  backgroundColor("#8fff");
}}.build(nifty);
```

So, that's it basically. We can define any attribute we'd like to apply to other elements later and give the style definition the name „redBackgroundCaption" so that we can later reference this exact style.

With the style definition in place we can rewrite our original example to use this style.

```
<text style="redBackgroundCaption" text="stuff 1" />
<text style="redBackgroundCaption" text="stuff 2" />
```

Nifty will now apply the attributes from the style definition to the text elements.

And we are now able to simply change the style definition and all of the elements where this style is applied will automatically update accordingly. Besides the benefit of reducing duplication this makes the GUI definition simpler, easier to read and easier to maintain as well.

OVERWRITE ATTRIBUTES

If required you can overwrite any attribute that has been defined by a style by directly applying the attribute directly at the element. This allows you to use a base style for your elements and if required you can use a different value for some of the attributes.

Nifty will apply all of the attributes of the style definition first and all of the attributes that you've specified last.

```
<text style="redBackgroundCaption" text="stuff 1" />
<text style="redBackgroundCaption" text="stuff 2" font="other.fnt"/>
```

In this examples the second text element will use a different font although the „redBackgroundCaption" style is still being applied.

ORGANIZE STYLES IN FILES

To better organize your style definitions you can put them in a separate XML file and include it into your actual XML with the <useStyle> element.

Here is an example Nifty style XML file „styles.xml".

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty-styles>
  <!-- define a style with the name „myStyle" -->
  <style id="myStyle">
  </style>

  <!-- you can have more <style> definitions here -->
</nifty-styles>
```

To include a Nifty style XML file you can use the <useStyles> element in XML.

```
<useStyles filename="styles.xml" />
<!-- you can now use „myStyle" in here -->
```

Or you can call the method „loadStyleFile" on the Nifty instance:

```
nifty.loadStyleFile("styles.xml");
```

Style files are a great way to switch the look and feel of your GUI. If you put the visual appearance of your GUI in a Nifty style file you can change the look by simply using a different <useStyles> file.

CONTROLS

BASICS

The basic building blocks of a Nifty GUI are the core elements: panel, image and text. Building GUIs out of those elements is possible but it's not very practicable. What we want to use instead are abstractions, like buttons, input fields, scrollbars and so on.

The Nifty GUI way to do that are controls. A Nifty GUI control is the combination of multiple panels, images and texts that together form a component. The component (control) is defined once and then it is used multiple times. You can see a control as some form of template as well. Nifty controls can be defined in XML or from Java using the JavaBuilder.

Before we dive into all of the details on how to create your own controls we'll first take a look on how you can use the standard controls that Nifty provides.

STANDARD CONTROLS AND STYLES

CONTROL INCLUDE

Nifty provides a standard set of controls that you can simply use in your own GUIs. All you need to do is to add „nifty-default-controls-<version>.jar" and the „nifty-style-black-<version>.jar" to your Java classpath and then you use the <useControl> and the <useStyles> tag to include both into your XML.

EXAMPLE

Include the Nifty „default-default-controls.xml" to use the standard controls and the „nifty-default-styles.xml" to use the standard look'n'feel.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
  <!-- include the style file for the standard controls -->
  <useStyles filename="nifty-default-styles.xml" />

  <!-- include the standard controls -->
  <useControls filename="nifty-default-controls.xml" />

...
```

You'll need to include both, the styles and the control to access the standard controls. Without the style file the controls don't know how they should look ;)

Of course you can do the same using Java only by calling two methods that the Nifty instance provides:

```
// load default styles
nifty.loadStyleFile("nifty-default-styles.xml");

// load standard controls
nifty.loadControlFile("nifty-default-controls.xml");
```

Once you've included both XML files the standard controls are available.

EXAMPLE

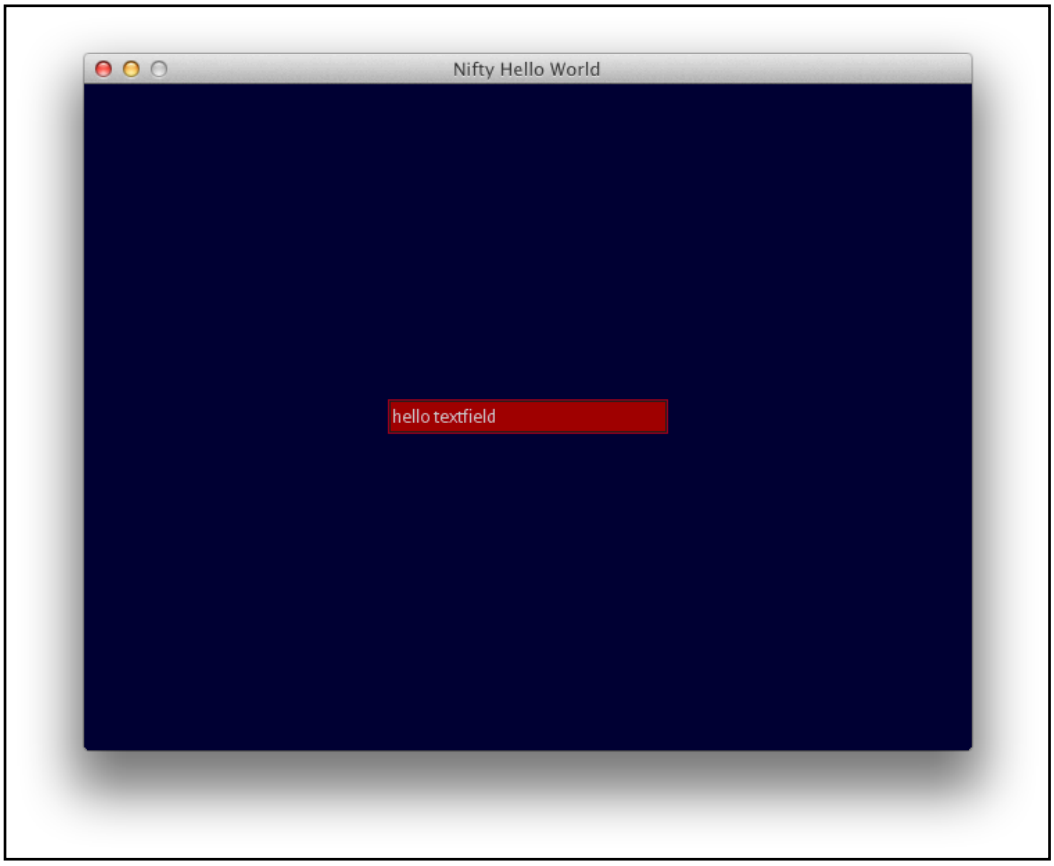
You can insert a control into your GUI with the <control> Tag. Here is an example to use the standard textfield control in a Nifty GUI XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty xmlns="http://nifty-gui.sourceforge.net/nifty-1.3.xsd" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://nifty-
gui.sourceforge.net/nifty-1.3.xsd http://nifty-gui.sourceforge.net/nifty-1.3.xsd">

  <!-- load styles -->
  <useStyles filename="nifty-default-styles.xml" />
  <useControls filename="nifty-default-controls.xml" />

  <!-- start screen -->
  <screen id="start">
    <layer backgroundColor="#003f" childLayout="center">
      <!-- use the textfield control -->
      <control id="input" name="textfield" width="200px" text="hello textfield"/>
    </layer>
  </screen>
</nifty>
```

Which will give us a simple textfield centered in the middle of the screen which is 200px width and contains the initial text of „hello textfield“:



As you see you can use the <control> tag in the same way as you would use any other Nifty element (panel, image, text).

Using the Java Builder we can get the same result when we use the following Java source:

```
// create screen
new ScreenBuilder("start") {{
  layer(new LayerBuilder("layer") {{
    childLayoutCenter();
    backgroundColor("#003f");
    control(new TextFieldBuilder("input", "hello textfield") {{
      width("200px");
    }});
  }});
}}.build(nifty);

// tell Nifty that it should show the „start“ screen
nifty.gotoScreen("start");
```

For all of the standard controls there are specific Java Builders that you can use to create a control. In the example above there is the TextFieldBuilder being used to create the textfield control. The specific Java Builders have the advantage that they are designed for a specific control and therefore expose special methods to use.

CONTROL API

Creating a control and adding it to your GUI is great but a control only makes sense when we can interact with it and in the case of the textfield actually get the text that the user provides or the control is useless.

To do this all of the standard controls provide an API to access the controls functions. In case of the textfield the API is the interface de.lessvoid.nifty.controls.TextField and you can access it using the findNiftyControl() method of the screen.

EXAMPLE

Access the TextField control API interface using the findNiftyControl() method of the screen class:

```
TextField textField = screen.findNiftyControl("input", TextField.class);
```

For a better understanding of what that means, here is the actual TextField interface that you get back.

```
public interface TextField extends NiftyControl {

  /**
   * Get the current TextField text.
   * @return text
   */
  String getText();

  /**
   * Set the Text of the TextField.
   * @param text new text
   */
  void setText(String text);

  /**
   * Change the max. input length to a new length.
   * @param maxLength max length
   */
  void setMaxLength(int maxLength);

  /**
   * Set the cursor position to the given index.
   * @param position new cursor position
   */
  void setCursorPosition(int position);

  /**
   * Enable a password character that is displayed instead of the actual text.
   * @param passwordChar character to use, like '*'
   */
  void enablePasswordChar(final char passwordChar);

  /**
   * Disable the password character which displays the text again,
   */
  void disablePasswordChar();

  /**
   * Checks if a password character is currently enabled.
   * @return true if password character is enabled and false if not.
   */
  boolean isPasswordCharEnabled();
}
```

So once you've the TextField interface you can call it's methods.

EXAMPLE

Get the text of a textfield using the TextField interface API.

```
TextField textField = screen.findNiftyControl("input", TextField.class);
String text = textField.getText();
```

There exists similar interfaces for the other standard controls.

CONTROL EVENTS

Some of the controls support EventBus notifications when interesting things happen to the control. The textfield control generates an event whenever the text of the textfield changes.

EXAMPLE

Subscribe for the TextFieldChangedEvent to listen for any text change events.

```
@NiftyEventSubscriber(id="input")
public void onTextfieldChange(final String id, final TextFieldChangedEvent event) {
  System.out.println(event.getText());
}
```

There exists similar events for the other standard controls as well.

CONTROL REFERENCE

You can find a reference of all standard controls online in the Nifty wiki.

http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=Nifty_Standard_Controls_%28Nifty_1.3%29

CUSTOM CONTROLS

CONTROL DEFINITION

Creating your own Nifty controls is not complicated. Let's see next how you can do that.

EXAMPLE

Control definition for a simple button control.

```
<controlDefinition style="nifty-button"
                   name="button"
                   controller="de.lessvoid.nifty.controls.button.ButtonControl"
                   inputMapping="de.lessvoid.nifty.input.mapping.MenuInputMapping">
  <panel style="#panel" focusable="true">
    <text id="#text" style="#text" text="$label"/>
  </panel>
</controlDefinition>
```

So basically that's a control definition in Nifty XML. We'll look at all the little details in a moment but first here is the same control definition using the Java Builder pattern:

```
new ControlDefinitionBuilder("button") {{
  controller("de.lessvoid.nifty.controls.button.ButtonControl");
  inputMapping("de.lessvoid.nifty.input.mapping.MenuInputMapping");
  style("nifty-button");
  panel(new PanelBuilder() {{
    style("#panel");
    focusable(true);
    text(new TextBuilder("#text") {{
      style("#text");
      text(controlParameter("label"));
    }});
  }});
}}.registerControlDefintion(nifty);
```

And now let‘s take a look at the details.

With the name attribute you can obviously name your control and if you later want to use the control you can select the control with its name.

With the control definition in place we can use the control with the <control> tag using „button“ as the name and this will work the same as with the name=“textfield“ before.

EXAMPLE

Use the newly defined button control using the <control> Tag.

```
<control id="theButton" name="button" label="OK" />
```

Using the newly defined button control using the Java Builder pattern:

```
control(new ControlBuilder("theButton", "button") {{
  parameter("label", "OK");
}});
```

When Nifty parses a Nifty XML file and it finds a control, it looks up a matching control definition using the control name. If a corresponding control definition is found the content of the control definition actually replaces the control tag. So all the panels, images and text elements that make up the control definition are inserted into the element tree at the position of the control. If you look at it in this way you can imagine controls as a form of a template.

Like screens it is possible to attach a controller class to a control. This works exactly the same as with screens, whenever something happens the controller is the first address that is called.

When resolving all of the GUI elements Nifty keeps track of the current controller class. The controller of a control is a Java class that gets all the events of the control. So let's say that we have a onClick() event on any element inside the control definition that event will not travel immediately to the screen controller but to the controller of the control. So this way you have a Java class representing the control and that gets all the events. This is an important mechanism to get controls working.

So to wrap that part up here is the Controller interface all Control classes need to implement:

```
public interface Controller {

  /**
   * Bind this Controller to a certain element.
   * @param nifty nifty
   * @param element the Element
   * @param parameter parameters from the xml source to init the controller
   * @param listener the ControllerEventListener
   */
  void bind(
      Nifty nifty,
      Screen screen,
      Element element,
      Properties parameter,
      Attributes controlDefinitionAttributes);

  /**
   * Init the Controller. You can assume that bind() has been called for all other
controls on the screen.
   * @param parameter
   * @param controlDefinitionAttributes
   */
  void init(Properties parameter, Attributes controlDefinitionAttributes);

  /**
   * Called when the screen is started.
   */
  void onStartScreen();

  /**
   * This controller gets the focus.
   * @param getFocus get focus (true) or loose focus (false)
   */
  void onFocus(boolean getFocus);

  /**
   * input event.
   * @param inputEvent the NiftyInputEvent to process
   * @return true, the event has been handled and false, the event has not been
handled
   */
  boolean inputEvent(NiftyInputEvent inputEvent);
}
```

You'll implement this interface and register it with the controlDefinition with the controller attribute.

You can put several of your own control definitions into a XML file and include it the same way as we've included the nifty default controls or you can define the control definitions directly in your Nifty XML.

CONTROL PARAMETERS

In the case of the button example we don't want all of our buttons to have the same label. So we need a way to customize the control.

The way this works is that we can override some attributes when we actual use the control. Maybe you remember this strange syntax in the button example:

```
<controlDefinition name="button" ...>
  <panel style="#panel" focusable="true">
    <text id="#text" style="#text" text="$label"/>
  </panel>
</controlDefinition>
```

If a attribute value inside of the control definition begins with the "$" character you can later set this value by using the value after the „$“ character as another attribute. You can think of the „$“ character as a way of introducing a new attribute for your control!

Assigning a value to this new attribute when you use the control will replace the value in the control definition. This works not only for text attributes but for all attributes of all elements!

EXAMPLE

The „label“ attribute of the button control is set to the value „OK“.

```
<control id="theButton" name="button" label="OK" />
```

The same works when using the Java Builder:

```
control(new ControlBuilder("theButton", "button") {{
  parameter("label", "OK");
}});
```

Please note the syntax: The method to set control parameters is called „parameter“ and you'll need to specify the attribute you want as the first parameter and the value as the second parameter.

CONTROL STYLES

The last piece of information that is missing are control styles. When you define your control with the control definition tag you are free to apply any style to the elements that your control uses. This works the same as we've seen before and you can just add a style attribute to the elements.

However there is one problem. When we use the control, let's say the button control, the style of the button will always be fixed. If, for instance, the button is defined as a red button then this button will always be applied in red and you always get a red button. This might be ok but what we really want is a control style. If I use the button and apply a different style, let's say the green button style, I want to use the same control but with the green button style applied.

And actually you can!

If you take a look at the control definition we've shown before, you've noticed two things:

1. The control definition itself has a style attribute and

2. the elements that make up the control use strange style names that begin with a # character.

The style attribute for the control definition is the default style that Nifty applies when you actually use the control. So if you don't set any other style when you use the control then Nifty will simply use the style that was given in the control definition tag.

Style names inside a control definition that start with a # character are called "sub styles". When Nifty resolves styles it combines the style name of the control definition ("nifty-button") and the style at the element inside the control ("#panel") to build a final style name ("nifty-button#panel"). And this allows us to define the style for the sub style too.

EXAMPLE

This is a sub style for the panel inside of the nifty-button style.

```
<style id="nifty-button#panel">
  <attributes backgroundImage="button/button.png"
              imageMode="sprite-resize:100,23,0,2,96,2,2,2,96,2,19,2,96,2,2"
              paddingLeft="7px"
              paddingRight="7px"
              width="100px"
              height="23px"
              childLayout="center"
              visibleToMouse="true" />
</style>
```

So using a control and not setting a style attribute will fall back to the style that was set in the control definition.

All the elements that have a sub style attached will get resolved using the combination of the style of the control definition and the sub style that was attached to the element. Nifty will resolve all of the sub styles and apply the attributes to the elements as we've seen before.

But this allows us to create a complete new style for a control. All we need to do is to create styles that consists of our name for the base style, e.g. "green-button" and the sub style given in the control definition, e.g. "#panel". So we simply define a style: "green-button#panel". When we later use this style on the control tag we can simply use our new style "green-button".

You can use the existing styles (and sub styles) as the base for your own styles. You only need to make sure that you'll define all sub styles of the control.

You can even change styles dynamically from Java using element.setStyle(). However there is one catch: Changing sub styles is not supported at runtime currently. So you can only apply the „green-button“ style when you create the button but not change a „red-button“ style to a „green-button“ style at runtime.

(You could change the style at runtime but only when this style does not have sub styles applied).

INTEGRATION WITH OTHER SYSTEMS

INTEGRATION WITH JME3

This topic is covered in detail on the jMonkeyEngine3 wiki that you can find online at http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:nifty_gui.

INTEGRATION WITH SLICK2D

The Nifty Slick2D Renderer is the binding between Nifty GUI and Slick2D in matters of graphic, user input and sound.

BASIC SETUP

The Slick2D renderer provides access to Nifty GUI by extending the

org.newdawn.slick.Game,
org.newdawn.slick.BasicGame,
org.newdawn.slick.state.GameState and
org.newdawn.slick.state.BasicGameState

of Slick2D. Each class or interface is implemented twice. One overlay type and one pure Nifty GUI type.

The overlay type of the classes are meant to display Nifty GUI as overlay over graphics rendered outside of Nifty GUI. The pure Nifty GUI classes are meant to display only the Nifty GUI on the screen.

Each of the implementations have at least one abstract method that you'll need to implement.

```
protected abstract void prepareNifty(Nifty nifty, StateBasedGame game);
protected abstract void prepareNifty(Nifty nifty);
```

It's one of the two methods written above. The first one is for GameState based implementations and the second one is for Game based implementations.

Inside this method you have to load the things the Nifty GUI is supposed to display. How you load the GUI is up to you. Either build it inside this method or load a XML file.

When using the overlay classes you'll get a few more methods that you'll need to implement.

```
protected abstract void initGameAndGUI(GameContainer container)
    throws SlickException;

protected abstract void initGameAndGUI(GameContainer container, StateBasedGame game)
    throws SlickException;
```

These two methods are supposed to be used to initialize the Nifty instance and the game in case you need it. Initializing the GUI is done by calling the initNifty functions that are provided by the super classes. These functions have various implementations and it's possible to use whatever fits. Preparing Nifty GUI is not supposed to be done in this function. As named before the prepareNifty() functions are used for this.

The overlay classes now also add two more functions called updateGame() and renderGame(). Both functions are respective used to handle the game unit. Using those functions ensures that Nifty receives the update and render calls properly.

So Nifty GUI does not need to be updated or rendered by hand. The library handles this internally.

RESOURCE LOADING API

The resource loading API provides an easy way to add your own methods of loading resouces (images/sounds/cursors/fonts) into the rendering environment. The basic loader storages can be found in the package de.lessvoid.nifty.slick2d.loaders. There it is possible to register more loaders to the Slick devices that are used to load resources.

There are already some implementations of these loaders that utilize most of the possibilities to load the data.

There are:

- Font loaders: de.lessvoid.nifty.slick2d.render.font.loader
- Cursor loaders: de.lessvoid.nifty.slick2d.render.cursor.loader
- Image loaders: de.lessvoid.nifty.slick2d.render.image.loader
- Sound loaders: de.lessvoid.nifty.slick2d.sound.sound.loader
- Music loaders: de.lessvoid.nifty.slick2d.sound.music.loader

For example when writing a new loader to load images the class needs to implement the de.lessvoid.nifty.slick2d.render.image.loader.SlickRenderImageLoader interface.

This class has to this class has to make sure to load this image or throw a de.lessvoid.nifty.slick2d.render.image.SlickLoadImageException in case loading the image fails. Then the class needs to be added to the loaders stored in de.lessvoid.nifty.slick2d.loaders.SlickRenderImageLoaders. This loader list will try all registered image loaders in order to load a resource and use the first one that does not throw a exception.

Those loaders are automatically utilised by the RenderDevice and SourceDevice implementations that are provided by the Slick2D-Renderer.

INPUT FORWARDING

Especially for the cases where Nifty-GUI is used as overlay over another game is often required that the game receives all the input event that were not handled by Nifty.

For this purpose there are a few implementations provided that take care for the input forwarding.

de.lessvoid.nifty.slick2d.input.PlainSlickInputSystem
Receives the input events from Slick and forwards them to the Nifty-GUI. All events not handled by the Nifty-GUI are discarded. In case your application is just supposed to display the Nifty-GUI, this one is the best choice.

de.lessvoid.nifty.slick2d.input.NiftySlickInputSystem
Receives the input events from Slick and forwards them to the Nifty-GUI. All events not handled by the Nifty-GUI are forwarded to a de.lessvoid.nifty.NiftyInputConsumer.

de.lessvoid.nifty.slick2d.input.SlickSlickInputSystem
Receives the input events from Slick and forwards them to the Nifty-GUI. All events not handled by the Nifty-GUI are forwarded to a org.newdawn.slick.InputListener

So in case you implement a listener of one of the two libraries, the required implementations are available. In case you want to write your own input system you should consider using de.lessvoid.nifty.slick2d.input.AbstractSlickInputSystem. This class already implements the required logic to forward to the Nifty-GUI and sends all events not handled by the Nifty-GUI to the abstract function handleInputEvent(...). Also this class takes care for handling events Nifty-GUI usually does not need. Such as high-level events like dragging, (double-)clicking, and so on.

REFERENCE

You can find additional informations online here.

Resource	URL
Project Page	http://sourceforge.net/projects/nifty-gui/
Ohloh.net	http://www.ohloh.net/p/nifty-gui
Wiki	http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=Main_Page
Blog	http://nifty-gui.lessvoid.com/
Twitter	http://twitter.com/#!/niftygui
Github	https://github.com/void256/nifty-gui

The End