Developers Guide

Authors

Shai Almog, Chen Fishbein, Eric Coolman

Table of contents

Introduction

Codename One is a set of tools for mobile application development that derive a great deal of its architecture from Java. It stands both as the name of the startup that created the set of tools and as a prefix to the distinct tools that make up the Codename One product.

The goal of the Codename One project is to take the complex and fragmented task of mobile device programming and unify it under a single set of tools, APIs and services to create a more manageable approach to mobile application development without sacrificing development power/control.

History

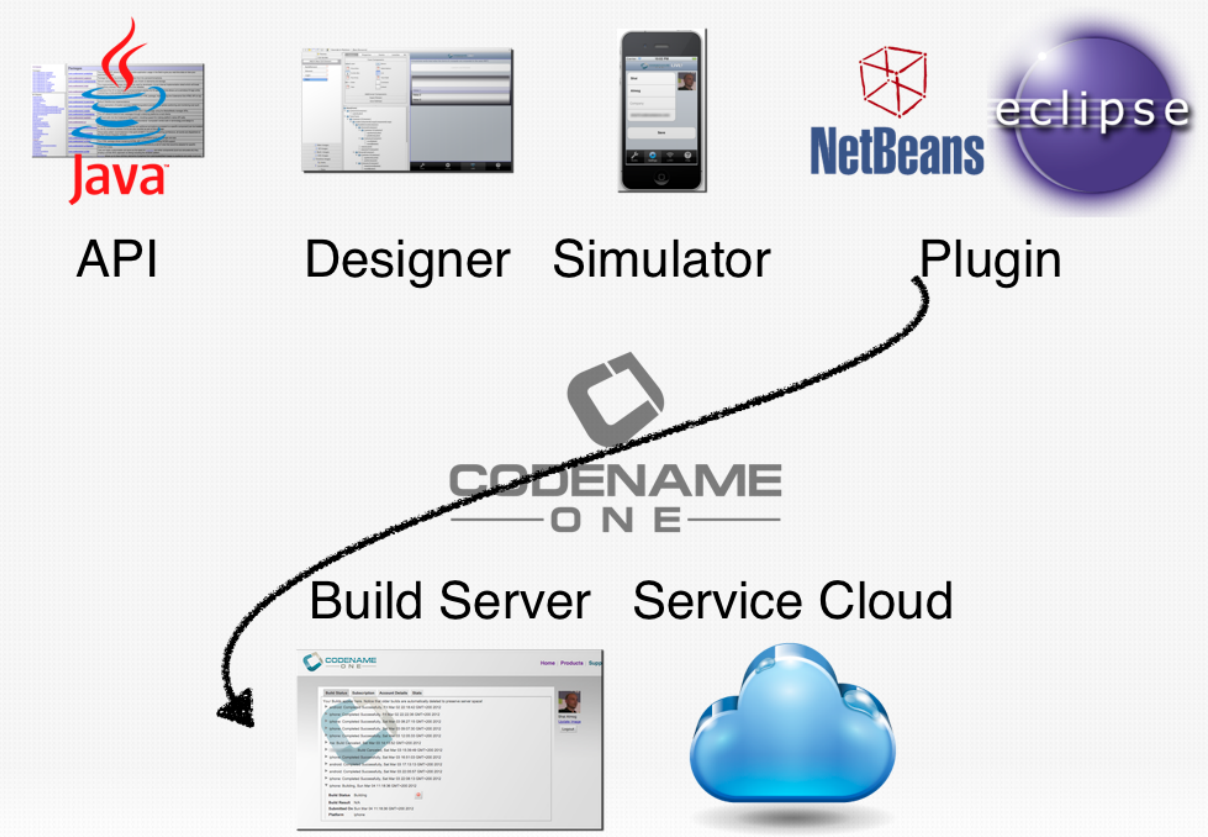
Codename One was started by Chen Fishbein & Shai Almog who authored the Open Source LWUIT[1] project at Sun Microsystems starting at 2007. The LWUIT project aimed at solving the fragmentation within J2ME/Blackberry devices by targeting a higher standard of user interface than the common baseline at the time. LWUIT received critical acclaim and traction within multiple industries but was limited by the declining feature phone market.
In 2012 the Codename One project has taken many of the basic concepts developed within the LWUIT project and adapted them to the smartphone world which is experiencing similar issues to the device fragmentation of the old J2ME phones.

How Does It Work

Codename One has 4 major parts: API, Designer, Simulator, Build/Cloud server.

- API - abstracts platform specific functionality
- Designer - allows developers/designers to design the GUI/theme and package various resources required by the application
- Simulator - allows previewing and debugging applications within the IDE
- Build/Cloud server - the server performs the build of the native application, removing the need to install additional software stacks.

[1] See http://lwuit.blogspot.com/ http://lwuit.java.net/



Codename One unifies the pieces illustrated above allowing developers to use them as a single coherent product. When building the final native application the build server produces the actual native application removing the need for a dedicated machine/installation.

Limitations & Capabilities

J2ME & RIM are very limited platforms to achieve partial Java 5 compatibility Codename One automatically strips the Java 5 language requirements from bytecode and injects its own implementation of Java 5 classes. Not everything is supported so consult the Codename One JavaDoc when you get a compiler error to see what is available.

Due to the implementation of the NetBeans IDE it is very difficult to properly replace and annotate the supported Java API's so the completion and error marking might not represent correctly what is actually working and implemented on the devices. However, the compilation phase will not succeed if you used classes that are unsupported.

Lightweight UI

The biggest differentiation for Codename One is the lightweight architecture which allows for a great deal of the capabilities within Codename One. A Lightweight component is a component which is written entirely in Java, it draws its own interface and handles its own events/states. This has huge portability advantages since the same code executes on all platforms, but it carries many additional advantages.
The components are infinitely customizable just by using standard inheritance and overriding paint/event handling. Theming and the GUI builder allow for live preview and accurate reproduction across platforms since the same code executes everywhere.

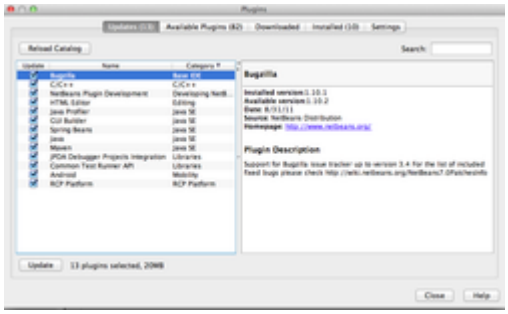
Installation

Installing Codename One In NetBeans

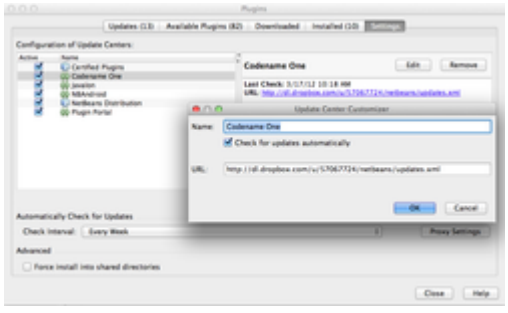
For the purpose of this document we will focus mostly on the NetBeans IDE for development, however most operations are almost identical in the Eclipse IDE as well. For instructions specific for Eclipse please go to the following section.

These instructions assume you have downloaded NetBeans 7.x,installed and launchedit.

Select the Tools->Plugins menu option & select the Settings tab

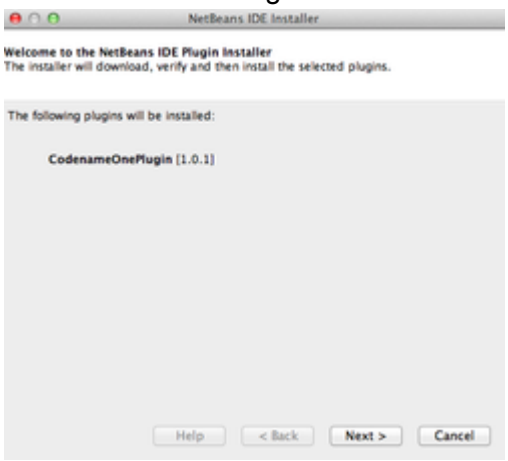


Click the "Add" button & enter the details below



For the name enter: Codename One

For the URL enter:
https://codenameone.googlecode.com/svn/trunk/CodenameOne/repo/netbeans/updates.xml
In The Available Plugins Tab Click "Reload Catalog" Then Check The CodenameOne Plugin

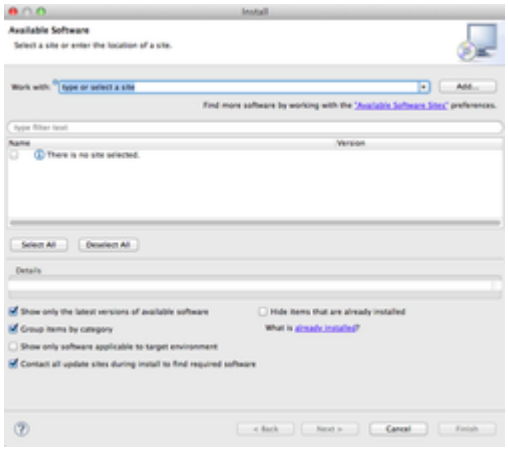


After that click the install button below. Follow the Wizard instructions to install the plugin

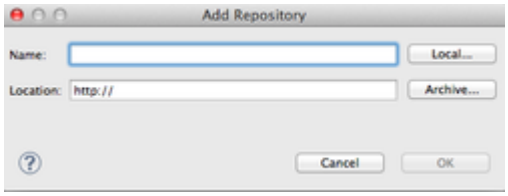
You will be informed that the plugin is unsigned which is indeed true, you just continue anyway.

Installing Codename One In Eclipse

Startup Eclipse and click Help->Install New Software. You should get this dialog

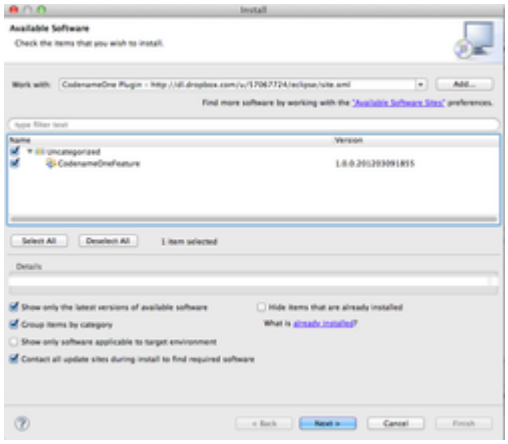


Click the Add button on the right side & fill out the entries



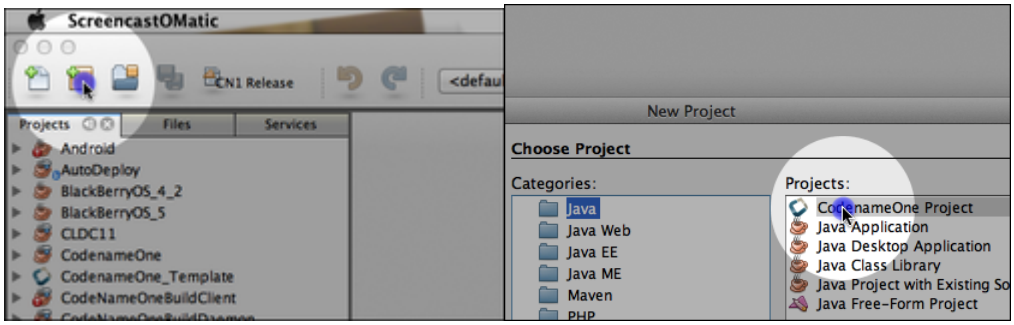
Enter Codename One for the name &https://codenameone.googlecode.com/svn/trunk/CodenameOne/repo/eclipse/site.xml for the location.

Select the entries & follow the wizard to install

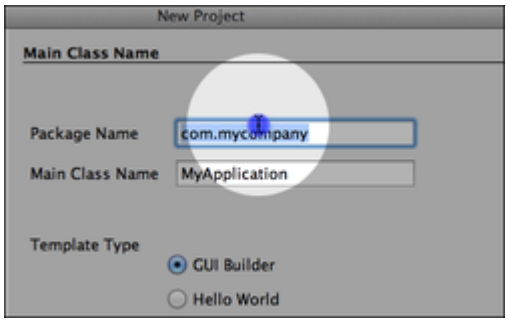


GUI Builder Hello World

Start by creating a new project in the IDE and selecting the Codename One project.

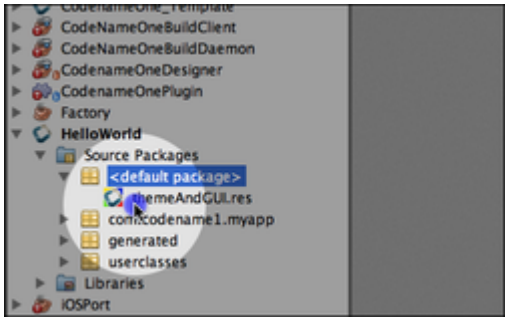


Use a proper package name for the project!

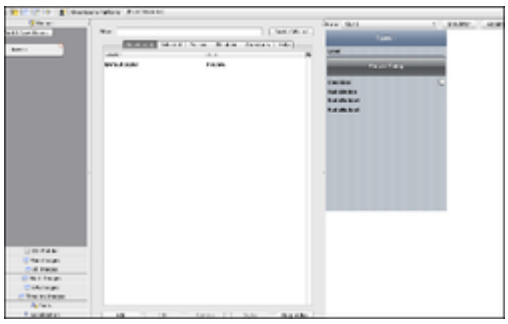


It is important to use proper package names for the project since platforms such as iOS & Android rely on these names for future updates. They are thus painful to change! The convention is the common Java convention of reverse domain names (e.g. com.codenameone for the owner of codenameone.com etc.).

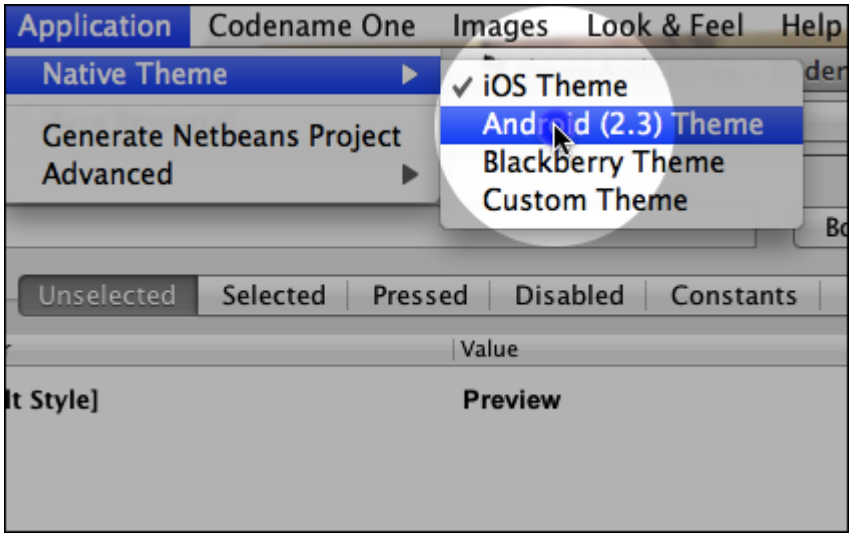
By default a GUI builder project is created, we will maintain this default for the purpose of this hello world.



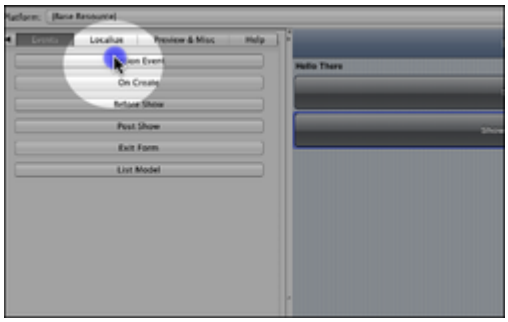
Since this is a GUI builder project you can open the Codename One Designer with the content of the theme.res file by double clicking the file. FYI: If you choose to rename this file in the future it is important to update the resource file name in the project properties settings.



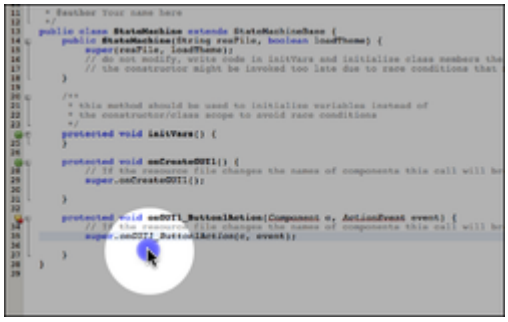
Within the Codename One Designer we can see several categories the most interesting of which are theme & GUI. We will start by opening the theme section and clicking the entry there. The default Codename One theme, derives from the platform native theme. We can easily change the "base theme" in the preview to see how the theme will act in different devices by clicking the native theme option in the application menu as such.



To edit the UI for the application we need to select the GUI section and click “GUI 1” within that section. We are then faced with a drag and drop interface to manipulate the GUI of the application we saw within the theme preview. All changes made here are reflected instantly to the theme preview.



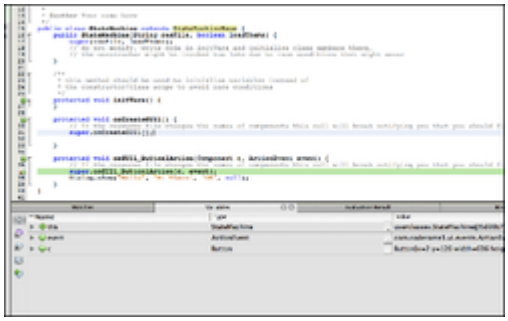
The GUI builder allows us to bind events, we will drag a new button into the GUI and use this functionality to select the button from the GUI and then add an action event to a button listener to a button..



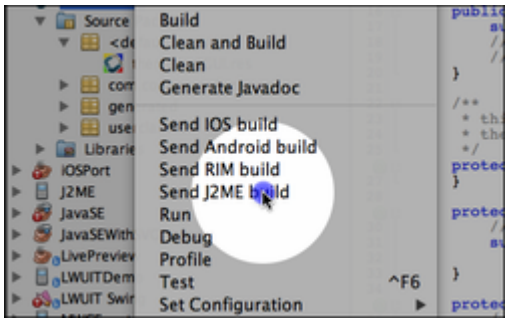
Clicking the action event button creates a new method within the IDE which we can can use to bind functionality to the button. Within the code we show a dialog by adding the code: Dialog.show("Hello", "Hi There", "OK", null);

If we inspect the Codename One project to some degree we will notice that it is pretty close to a standard Java project hence it can be modified/used in very similar ways. You can use the

debugger and most refactoring functionality as usual. **Notice** that you should not change the classpath settings for the project since library support is a more complex issue than just modifying the classpath!
Furthermore, if you change package/class names for some of the core classes you will need to update the project settings accordingly.



In order to get a native application for the devices you need to send a build to the build server. Before you get started you will probably need to read the signing section of this guide in order to produce an actual working application. Right click the project and select the device type for which you wish to build. If you haven't registered in the build server just visit http://www.codenameone.com/ and signup for free to get an account there!



Within the build server at http://www.codenameone.com/ you can follow the status of your current build.

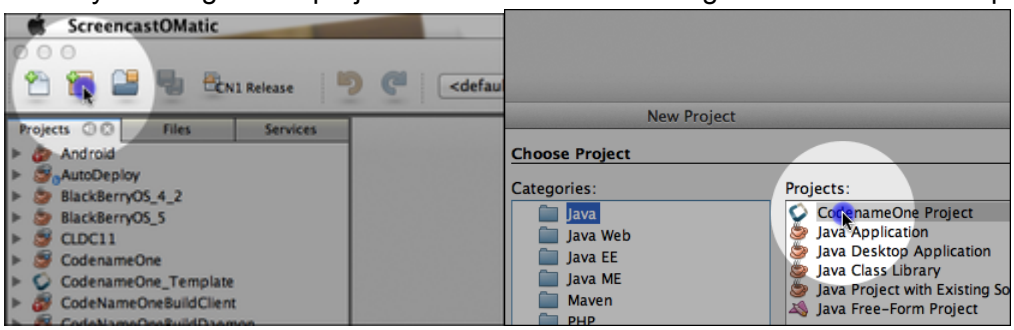
In the build server we can follow the progress of the build and the results for the current build. We can email a link to the deployment files, download them or use a QR reader to install them

to the phone. The Codename One LIVE! application allows following the status and installing applications directly from the build server.

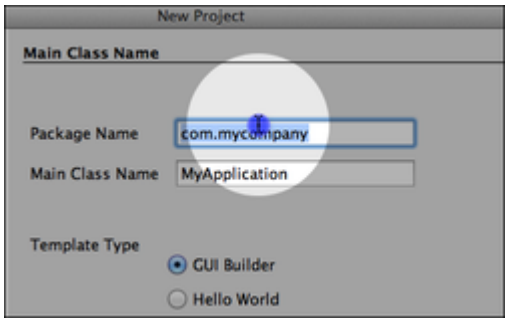
Manual Hello World

Some developers prefer writing all their code and avoiding the GUI builder like a plague, for them this tutorial covers the creation of a hello world application without the GUI builder.

Start by creating a new project in the IDE and selecting the Codename One project.



Use a proper package name for the project!



It is important to use proper package names for the project since platforms such as iOS & Android rely on these names for future updates. They are thus painful to change! The convention is the common Java convention of reverse domain names (e.g. com.codenameone for the owner of codenameone.com etc.).

Make sure to select the Hello World project and NOT the GUI Builder project!

Within your project main class you will see a start method that contains the code:

```
Form f = new Form("Hello World");
```

to add a button to the form that will show a dialog add the following:

```
Button d = new Button(“Show Dialog”);
f.addComponent(d);
d.addActionListener(new ActionListener() {
    public void ActionPerformed(ActionEvent ev) {
        Dialog.show("Hello", "Hi There", "OK", null);
    }
});
```

This will show a dialog when the user clicks the button. You can use the play button in the IDE to run the application.

Basics: Themes, Styles, Components & Layouts

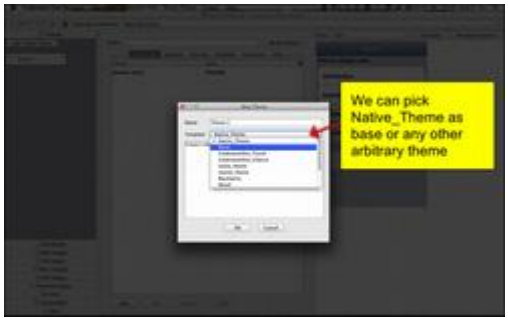
This chapter covers the basic ideas underlying a Codename One application. It focuses mostly on the issues related to UI.

What Is A Theme, What Is A Style & What Is a Component?

Every visual element within Codename One is is a component, the button on the screen is a Component and so is the Form in which it is placed. This is all represented within the Component class which is probably the most central class in Codename One.
The appearance of the component is determined by several style objects, every component has 4 style objects associated with it: Selected, Unselected, Disabled & Pressed.
Only one style is applicable at any given time and it can be queried via the getStyle() method. A style contains the colors, fonts, border, spacing information relating to how the component is presented to the user.
A theme allows the designer to define the styles externally via a set of UIID's (User Interface ID's), the themes are created via the Codename One Designer tool and allow developers to separate the look of the component from the application logic.

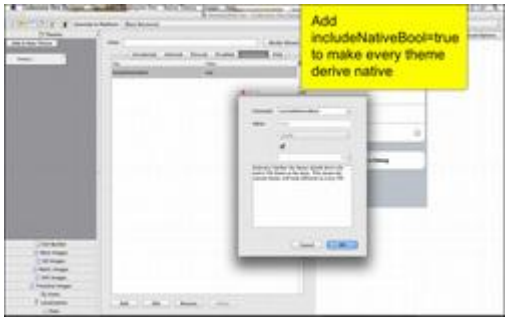
Creating A Native Theme

When creating a Codename One theme the default uses the platform native theme



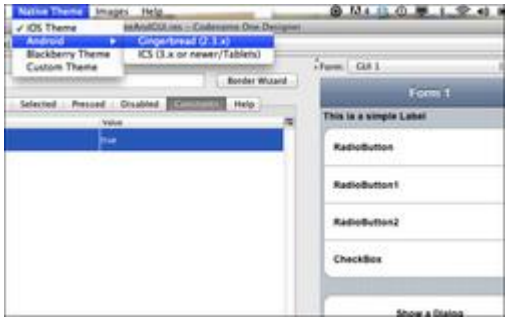
You can easily create a theme with any look you desire or you can "inherit" the platform native theme and start from that point. When adding a new theme you are given the option.

Any theme can be configured to derive a native theme



Codename One uses a theme constant called "includeNativeBool", when that constant is set to true Codename One starts by loading the native theme first and then applying all the theme settings. This effectively means your theme "derives" the style of the native theme first, similar to the cascading effect of CSS.

By avoiding this flag you can create themes that look EXACTLY the same on all platforms. You can simulate different OS platforms by using the native theme menu option



Developers can pick the platform of their liking and see how the theme will appear in that particular platform by selecting it and having the preview update on the fly.

You can easily create deep customizations that span across all themes

In this case we just customized the UIID of a label and created a style for the new UIID. When deriving a native theme its important to check the various platform options to make sure that basic assumptions aren't violated. E.g. labels might be transparent on one platform but opaque on others. Or labels might look good in a dialog in Android but look horrible in an iOS dialog (hint: use the DialogBody UIID for text content within a dialog).

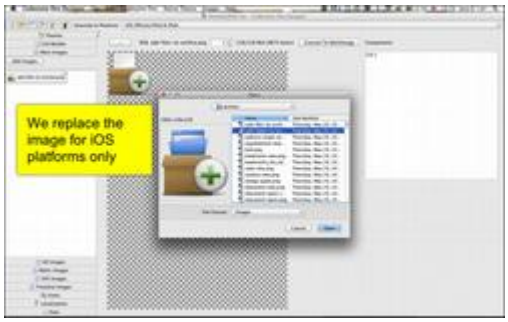
Codename One allows you to override a resource for a specific platform

A common case we run into when adapting for platform specific looks is that a specific resource should be different to match the platform conventions. The Override feature allows us to define resources that are specific to a given platform combination. Override resources take precedence over embedded resources thus allowing us to change the look or even behavior (when overriding a GUI builder element) for a specific platform/OS.

To override select the platform where overriding is applicable

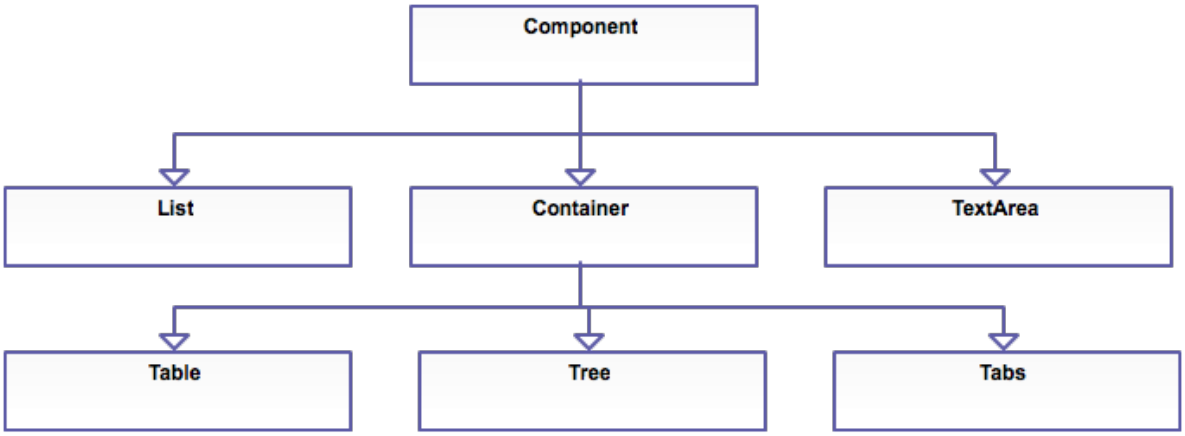
Then click the green checkbox to define that this resource is specific to this platform. All resources added at this point will only apply to the given platform. If you change your mind and are no longer interested in a particular override just delete it in the override mode and it will no longer be overridden.

In this case we just select a new image object applicable to this platform

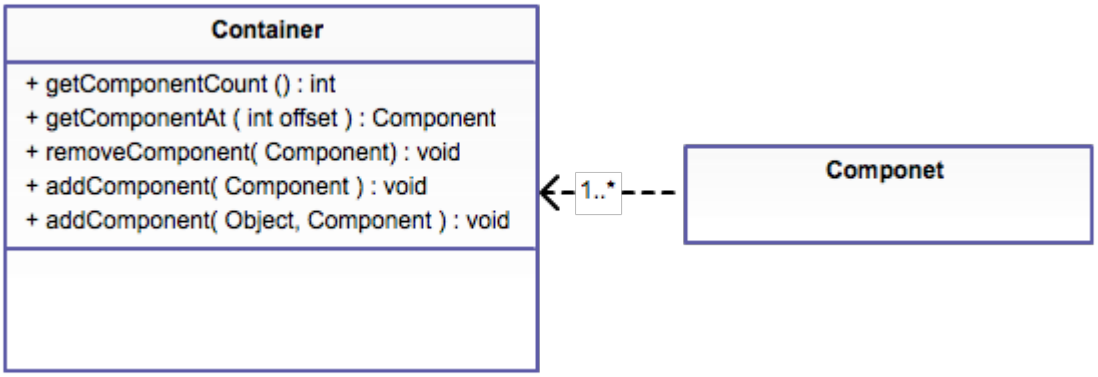


This is easily done by selecting the "..." button. We can easily do the same in the GUI builder although this is a dangerous road to start following since it might end up with a great deal of fragmentation.

Component/Container Hierarchy



The component class is the basis of all UI widgets in Codename One, to arrange multiple components together we use the Container class which itself “IS A” Component subclass. The Container is a Component that contains Components effectively allowing us to nest Containers infinitely to build any type of visual hierarchy we want by nesting Containers.



Layout Managers

Layout managers are installed on the Container class and effectively position the Components within the given container. The LayoutManager class is abstract and allows users to create their own layout managers, however Codename One ships with multiple layout managers allowing for the creation of many layout types.

The layout managers are designed to allow developers to build user interfaces that seamlessly adapt to all resolutions and thus they don't state the component size/position but rather the intended layout behavior. To install a layout manager one does something like this:

```
Container c = new Container(new BoxLayout(BoxLayout.X_AXIS));
c.addComponent(new Button("1"));
c.addComponent(new Button("2"));
c.addComponent(new Button("3"));
```

This would produce 3 buttons one next to the other horizontally.

There are two major types of layout managers: Constraint based & regular. The regular layout managers like the box layout are just installed on the container and "do their job". The constraint based layout managers associate a value with a Component sometimes explicitly and sometimes implicitly. Codename One ships with 3 such layouts BorderLayout, TableLayout & GroupLayout.
A constraint layout can/must accept a value when adding the component to indicate its position e.g.:

```
Container c = new Container(new BorderLayout());
c.addComponent(BorderLayout.CENTER, new Button("1"));
c.addComponent(BorderLayout.NORTH, new Button("2"));
c.addComponent(BorderLayout.SOUTH, new Button("3"));
```

This will stretch button 1 across the center of the container and buttons 2-3 will be stretched horizontally at the top and bottom of the container. Notice that the order to adding doesn't mean much once we have a constraint involved...

Flow Layout

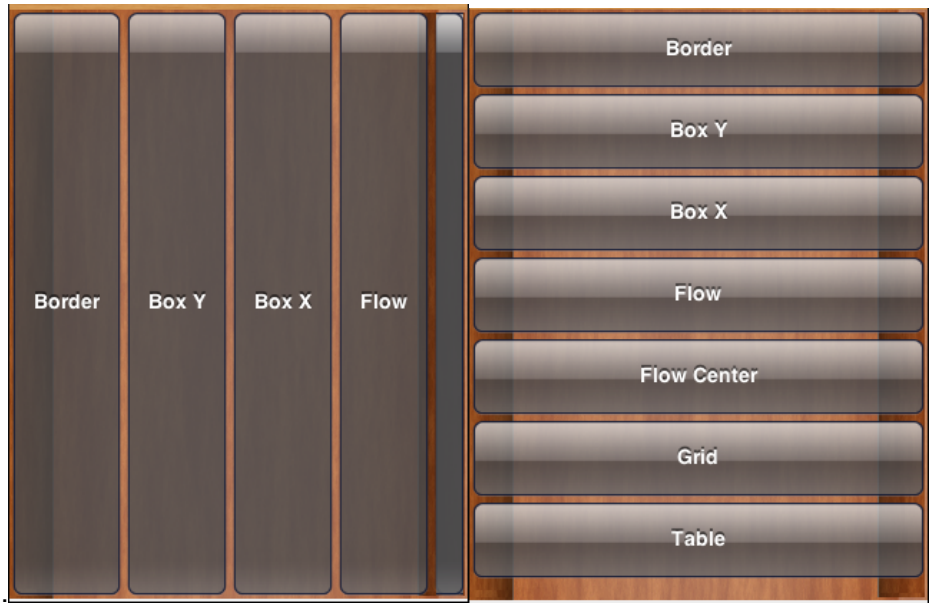
Flow layout can be used to just let the components "flow" horizontally and break a line when reaching the end of the component. It is the default layout manager for Codename One but because it is so flexible it could be problematic since it can cause the preferred size of the Container to provide false information triggering endless layout reflows (see below).

Flow layout can be aligned to the left (by default) center or to the right. Components within the flow layout get their natural preferred size by default and are not stretched in any axis.
The layout manager also supports modifying the horizontal alignment of the flow layout in cases where the container grows vertically.


```
final Container layouts = new Container();
final Button borderLayout = new Button("Border");
final Button boxYLayout = new Button("Box Y");
final Button boxXLayout = new Button("Box X");
final Button flowLayout = new Button("Flow");
final Button flowCenterLayout = new Button("Flow Center");
final Button gridLayout = new Button("Grid");
final Button tableLayout = new Button("Table");
layouts.setLayout(new FlowLayout(Component.CENTER));
layouts.addComponent(borderLayout);
layouts.addComponent(boxYLayout);
layouts.addComponent(boxXLayout);
layouts.addComponent(flowLayout);
layouts.addComponent(flowCenterLayout);
layouts.addComponent(gridLayout);
layouts.addComponent(tableLayout);
```

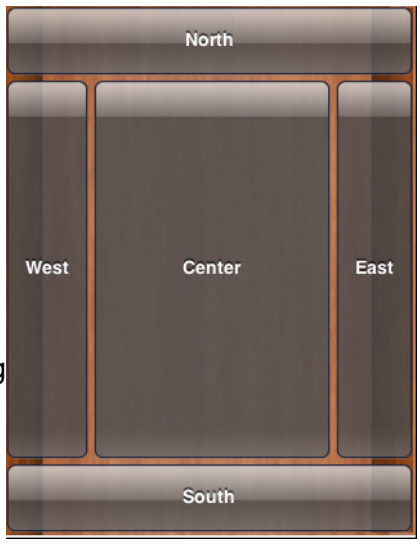
Box Layout

Box layout allows placing components in a horizontal or a vertical line that doesn't break the line. Components are stretched along the opposite axis, e.g. X axis box will flow components horizontally and stretch them vertically.



Box layout accepts the axis in its constructor, the axis can be either BoxLayout.X_AXIS or BoxLayout.Y_AXIS.

Border Layout



Border layout is quite unique, its a constraint based layout that can place up to 5 components in one of the 5 positions: North, South, East, West or Center.

The layout always stretches the North/South components on the X axis to completely fill the container and the East/West components on the Y axis. The center component is stretched to fill the remaining area by default. However, the border layout has a flag to manipulate the behavior of the center component allowing it to be placed in the absolute center without stretching.

Grid Layout

The Grid Layout accepts a predefined grid rows/columns and grants all components within it an equal size based on the dimensions of the largest component. It is an excellent layout for a set of icons in a grid.
It can also dynamically calculate the columns/rows based on available space.

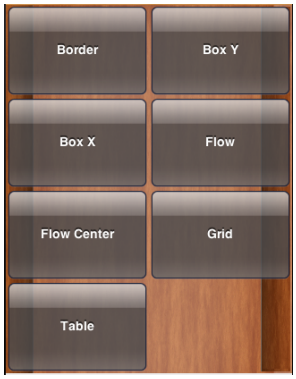


Table Layout

The table layout is far more elaborate than the grid layout and more akin to the HTML table structure. It is a constraint based layout, however it includes a default constraint if none are specified (e.g. using

```
container.addComponent(component);
```

is equivalent to using addComponent((layout.createConstraint(), cmp)).
A table generally gives elements their preferred sizes but stretches them based on column/row. There are abilities within the constraint element to define multiple behaviors such as row/column spanning, alignments and grow behavior.

The table layout will automatically size components to the largest preferred size in the row/column until running out of space, if the table is not horizontally scrollable this will happen when the edge of the parent container is reached (close to the edge of the screen) and further components will be "crammed together". Notice that all cells in the table layout are sized to fit the entire cell always. To align, or margin cell's a developer can use the methods of the component/Style appropriately.

A developer can provide hints to the table layout to enable spanning and more detailed column/row sizes using the constraint argument to the addComponent method. The constraint argument is an instance of TableLayout.Constraint that **must not** be reused for more than one cell, this will cause an exception.

A constraint can specify the absolute row/column where the entry should fit as well as spanning between cell boundaries. Notice that in the picture the "First" cell is spanned vertically while the "Spanning" cell is spanned horizontally. This is immensely useful in creating elaborate UI's,

Constraints can also specify a height/width for a column/row that will override the default, this size is indicated in percentage of the total table layout size. In the picture you can see that the "First" label is sized to 50% width while the "Forth" label is sized to 20% height.

```
Form mainForm = new Form("Table Layout");
TableLayout layout = new TableLayout(4, 3);
mainForm.setLayout(layout);
TableLayout.Constraint constraint = layout.createConstraint();
constraint.setVerticalSpan(2);
constraint.setWidthPercentage(50);
mainForm.addComponent(constraint, new Label("First"));
mainForm.addComponent(new Label("Second"));
mainForm.addComponent(new Label("Third"));

constraint = layout.createConstraint();
constraint.setHeightPercentage(20);
mainForm.addComponent(constraint, new Label("Forth"));
mainForm.addComponent(new Label("Fifth"));
constraint = layout.createConstraint();
constraint.setHorizontalSpan(3);
Label span = new Label("Spanning");
span.getStyle().setBorder(Border.createLineBorder(2));
span.setAlignment(Component.CENTER);
mainForm.addComponent(constraint, span);
mainForm.show();
```

Layered Layout

The layered layout just places the components in order one on top of the other and sizes them all to the size of the largest component. This is useful when trying to create an overlay on top of an existing component e.g. an “x” button to allow removing the component.

Understanding Preferred Size

The component class contains many useful methods, one of the most important ones is calcPreferredSize() which is invoked to recalculate the size a component “wants” when something changes (by default Codename One calls getPreferredSize() which caches the value).

The preferred size is decided by the component based on many constraints such as the font size, border sizes, padding etc. When a layout places the component it will size it based on its preferred size or ignore its preferred size either entirely or partially. Eg. FlowLayout always gives components their exact preferred size yet BorderLayout resizes the center component by default (and the other components on one of their axis).

Layout Reflow

When adding a component to a form which isn't shown on the screen there is no need to tell the UI to repaint or reflow. This happens implicitly. However, when adding a component to a UI which is already visible the component will not show by default.
The reason for this is performance based. E.g. imagine adding 100 components to an already visible form with 100 components within it. If we laid out automatically layout would have happened 100 times when it can happen only once.
That is why when you add components to a form that is already showing you should invoke revalidate or animate the layout appropriately. This also enables layout animation behavior explained below.

Layout Animations

To understand animations you need to understand a couple of things about Codename One components. When we add a component to a container it's generally just added but not positioned anywhere. A novice might notice the setX/Y/Width/Height methods on a component and just try to position it absolutely.

This won't work since these methods are meant for the layout manager which it implicitly invoked when a form is shown (internally in Codename One) and the layout manager uses these methods to position the components as it sees fit.

However, if you add components to a Codename One Form that is already shown it is your responsibility to invoke revalidate (or layoutContainer) to arrange the newly added components. Codename One doesn't "reflow" implicitly since that would be hugely expensive, imagine doing the layout calculations for every component added to the container the cost would be closer to a factorial of the original cost of adding a component.

The animateLayout() is simply a fancy form of revalidate. After changing the layout when you invoke this method it will animation the components to their new sizes and positions seamlessly.

The first example in the tipster demo shows an "interlace" effect where the components each slide from a separate direction into the screen. This is the code I used before showing the form:

```
f.revalidate();
 for(int iter = 0 ; iter < c.getComponentCount() ; iter++) {
    Component current = c.getComponentAt(iter);
    if(iter % 2 == 0) {
       current.setX(-current.getWidth());
    } else {
       current.setX(current.getWidth());
    }
 }
 c.setShouldCalcPreferredSize(true);
 c.animateLayout(1000);
```

Lets go over this line by line:

```
f.revalidate();
```

I make sure the layout is valid so I can start from the correct component positions.

```
    if(iter % 2 == 0) {
       current.setX(-current.getWidth());
    } else {
       current.setX(current.getWidth());
    }
```

I manually position every component outside of the screen, if they are odd I place them to the right and if they are even I place them to the left.

```
c.setShouldCalcPreferredSize(true);
```

I mark the UI as needing layout. This is crucial since I validated the UI earlier (by calling revalidate). Changing the X/Y/Width/Height doesn't trigger a validation! Codename One doesn't know I made that change!
By calling setShouldCalcPreferredSize I'm explicitly telling Codename One that I changed something in the UI and I want it to validate, normally this method is implicitly invoked by Codename One.

```
c.animateLayout(1000);
```

Perform the animation over the length of a second, this might seem like much but the animation starts before the form entry transition so it isn't that much.

The other animations are even simpler than this one and all follow the same basic rules, place the components wherever you want either manually (or by changing the layout) and use animateLayout() to automatically rearrange them to the new position.

Building Your Own Layout Manager

Layout managers in Codename One are a remarkably powerful tool, I won't go into all the elaborate ways in which you can modify the layout in Codename One since this is covered rather well in the tutorial and developer guide. Instead I will try to show something that is a bit under documented, mostly because its almost exactly like its Swing/AWT equivalent: building a layout manager.

A layout manager contains all the logic for positioning Codename One components, it essentially traverses a Codename One container and positions components absolutely based on internal logic. When we build our own component we need to take padding into consideration, when we build the layout we need to take margin into consideration. Building a layout manager involves two simple methods: layoutContainer & getPreferredSize.

layoutContainer is invoked whenever Codename One decides the container needs rearranging, Codename One tries to avoid calling this method and only invokes it at the last possible moment. Since this method is generally very expensive (imagine the recursion with nested layouts...), Codename One just marks a flag indicating layout is "dirty" when something important changes and tries to avoid "reflows".

getPreferredSize allows the layout to determine the size desired for the container, this might be a difficult call to make for some layout managers that try to provide both flexibility and simplicity. Most of flow layout bugs stem from the fact that this method is just impossible to implement for flow layout. The size of the final layout won't necessarily match the requested size (it probably won't) but the requested size is taken into consideration, especially when scrolling and also when sizing parent containers.

This is a layout manager that just arranges components in a center column aligned to the middle:

```
/**
* Layout manager that arranges components in a center column
*
* @author Shai Almog
*/
public class CenterLayout extends Layout {
```

```
public void layoutContainer(Container parent) {
    int components = parent.getComponentCount();
    Style parentStyle = parent.getStyle();
    int centerPos = parent.getLayoutWidth() / 2 + parentStyle.getMargin(Component.LEFT);
    int y = parentStyle.getMargin(Component.TOP);
    for(int iter = 0 ; iter < components ; iter++) {
        Component current = parent.getComponentAt(iter);
        Dimension d = current.getPreferredSize();
        current.setSize(d);
        current.setX(centerPos - d.getWidth() / 2);
        Style currentStyle = current.getStyle();
        y += currentStyle.getMargin(Component.TOP);
        current.setY(y);
        y += d.getHeight() + currentStyle.getMargin(Component.BOTTOM);
    }
}

public Dimension getPreferredSize(Container parent) {
    int components = parent.getComponentCount();
    Style parentStyle = parent.getStyle();
    int height = parentStyle.getMargin(Component.TOP) +
parentStyle.getMargin(Component.BOTTOM);
    int marginX = parentStyle.getMargin(Component.RIGHT) +
parentStyle.getMargin(Component.LEFT);
    int width = marginX;
    for(int iter = 0 ; iter < components ; iter++) {
        Component current = parent.getComponentAt(iter);
        Dimension d = current.getPreferredSize();
        Style currentStyle = current.getStyle();
        width = Math.max(d.getWidth() + marginX + currentStyle.getMargin(Component.RIGHT)
                + currentStyle.getMargin(Component.LEFT), width);
        height += currentStyle.getMargin(Component.TOP) + d.getHeight() +
                currentStyle.getMargin(Component.BOTTOM);
    }
    Dimension size = new Dimension(width, height);
    return size;
}
}
```

Here is a simple example of using it:

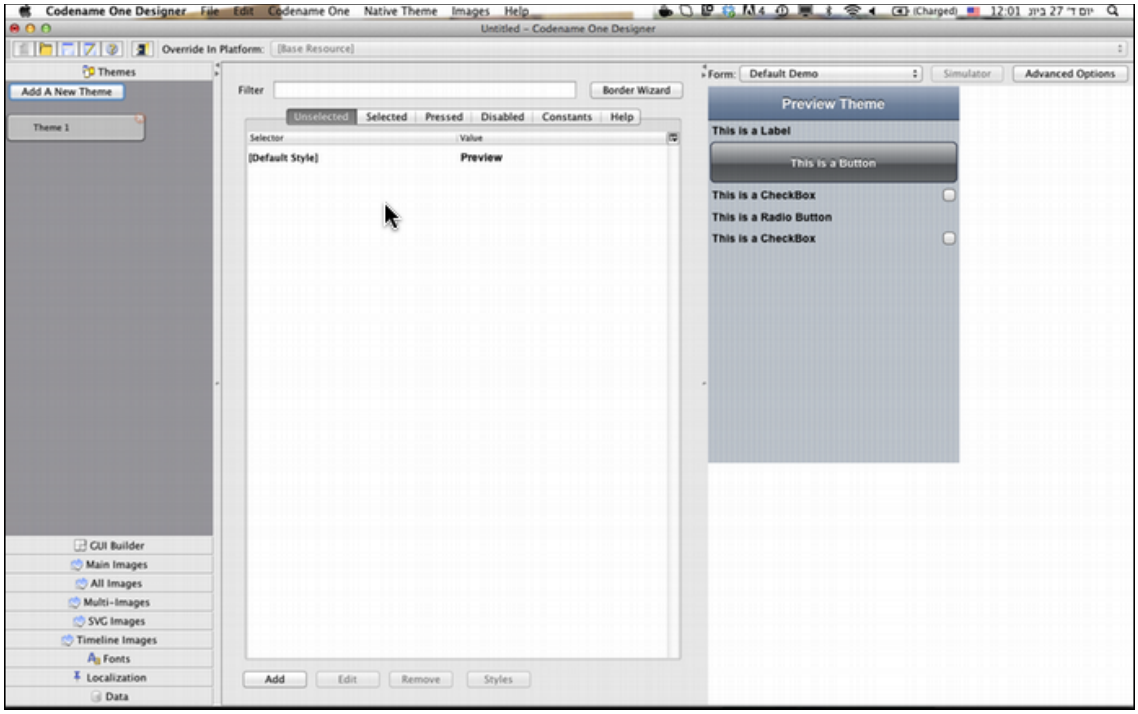
```
Form f = new Form("Centered");
```

```
f.setLayout(new CenterLayout());
for(int iter = 1 ; iter < 20 ; iter++) {
    f.addComponent(new Button("Button: " + iter));
}
f.addComponent(new Button("Really Wide Button Text!!!"));
f.show();
```

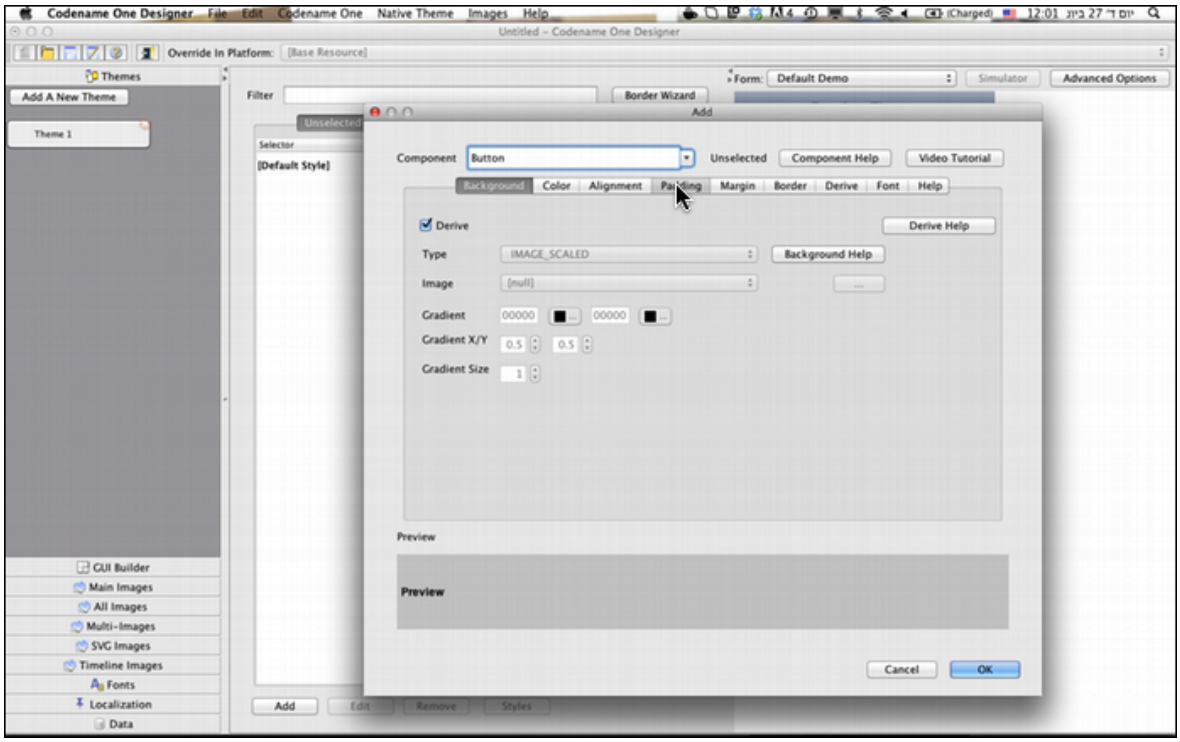
Theme Basics

This chapter covers the creation of a simple hello world style theme and how it can be customized visually. It uses the Codename One Designer tool to demonstrate basic concepts in theme creation such as 9-piece borders, selectors and style types.

Codename One themes are effectively a set of UIID's mapped to a Style object, we can create a new theme by adding it in the Designer tool and customizing the UIID values.



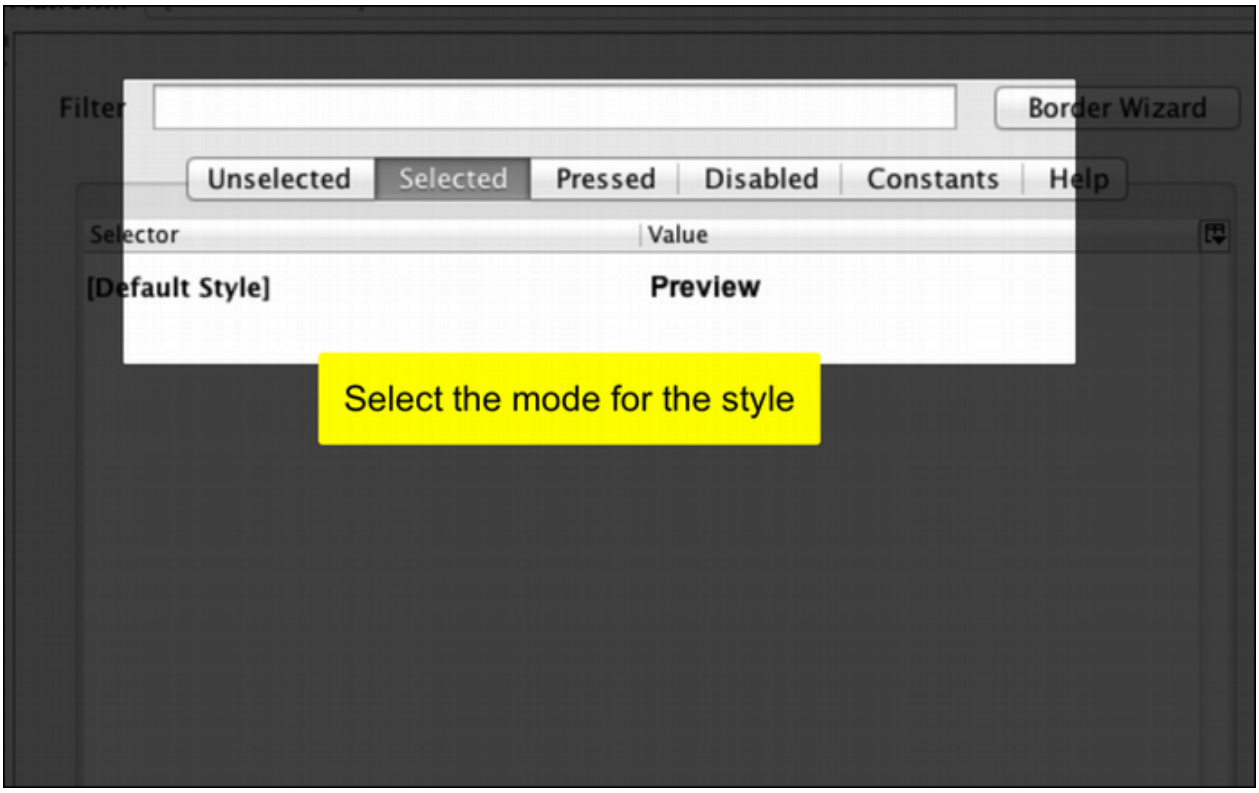
We can add a component style to a component such as Button, typically UIID's are named with the same name as the Component class. You can modify the UIID of a component by invoking setUIID(String) on an arbitrary component or changing the UIID property in the GUI builder.



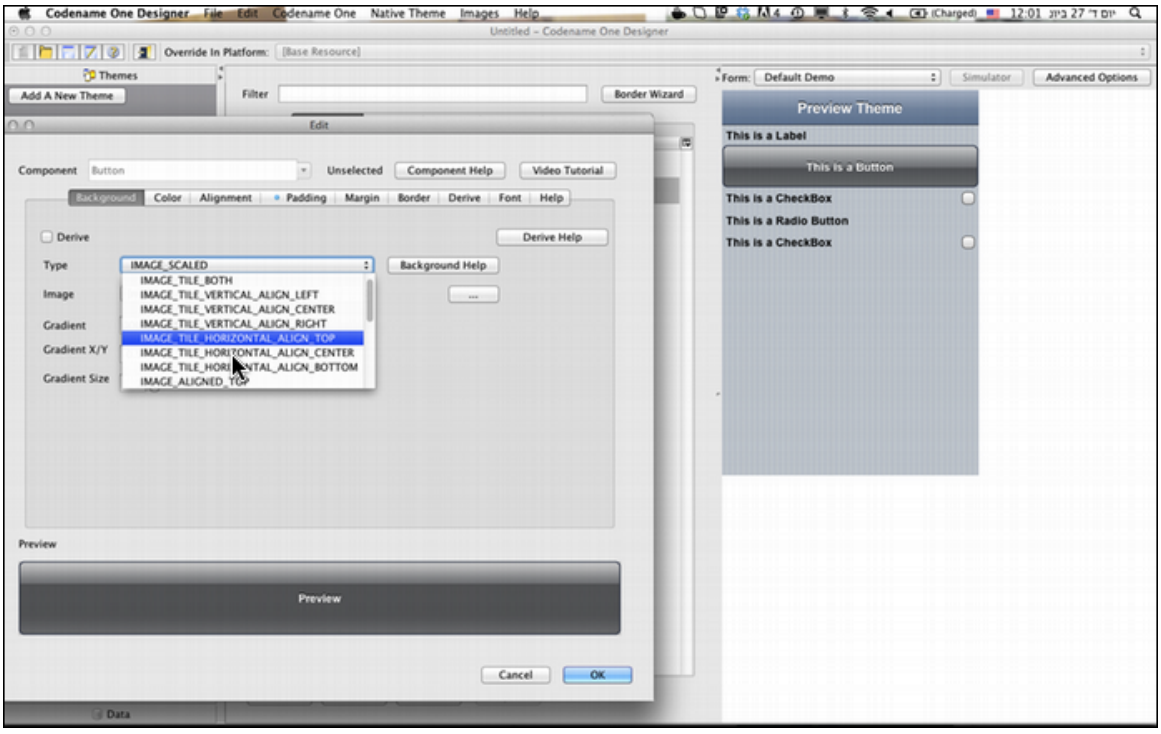
Styles can have one of 4 states:

1. Default (unselected) - the way a component appears when its in none of the other states.
2. Selected - shown when the component has focus or is active (on a touch screen device this only appears when the user interacts with the device with touch or with a physical key).
3. Pressed - shown when the component is pressed. This is only active for Button's.
4. Disabled - shown when the component is disabled.

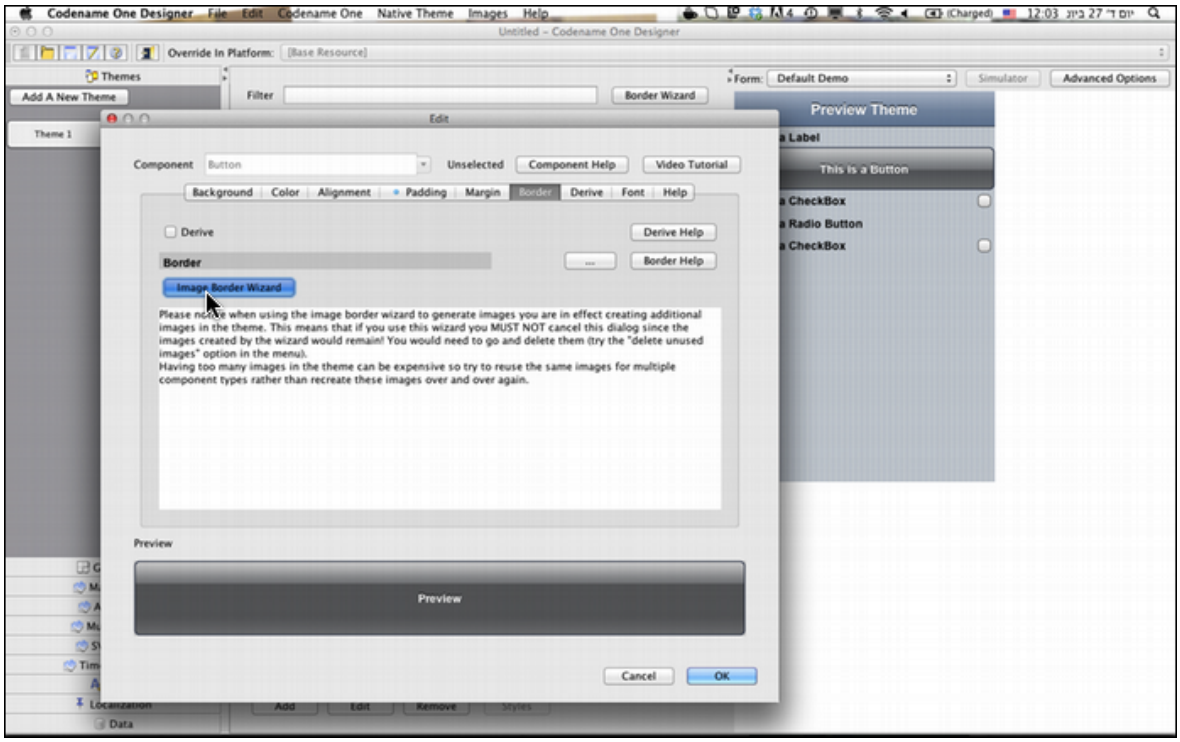
You can add a style to any one of the states in the Designer to make the component appear as expected in those cases.



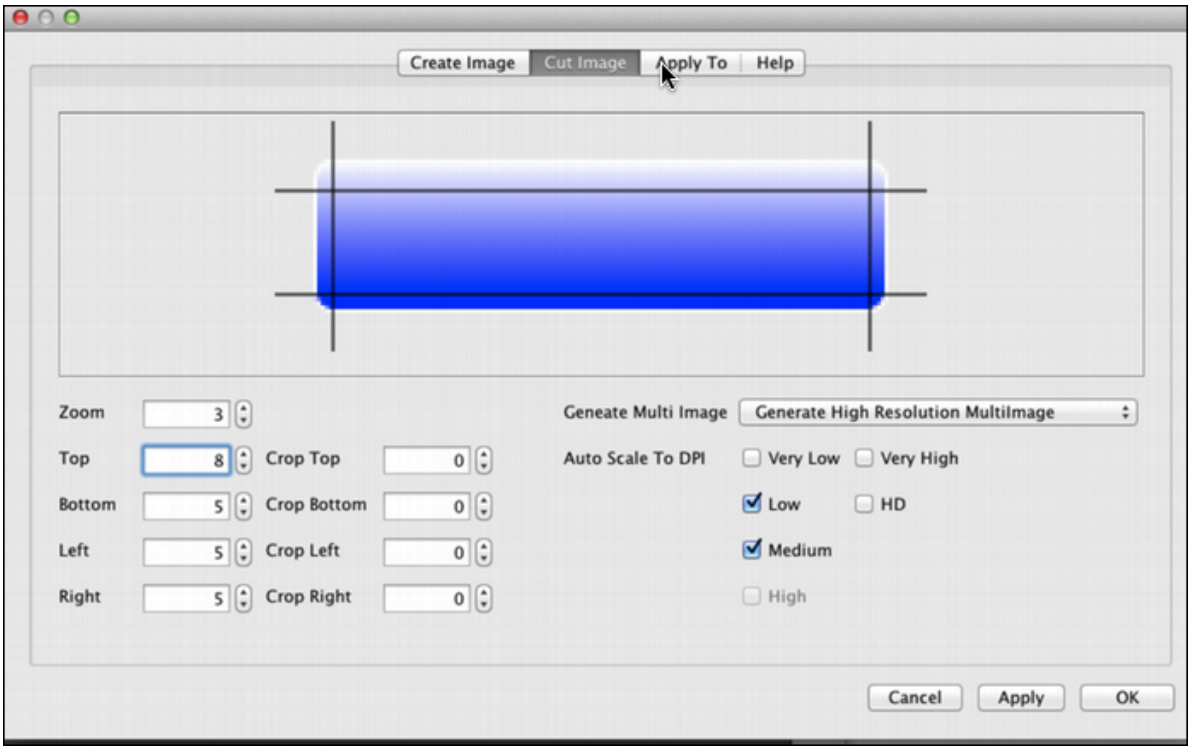
When editing the style of the component you can customize multiple things such as the background image, the way such a background image is displayed or a gradient/solid color background. You can customize colors, fonts, padding/margin, border etc.



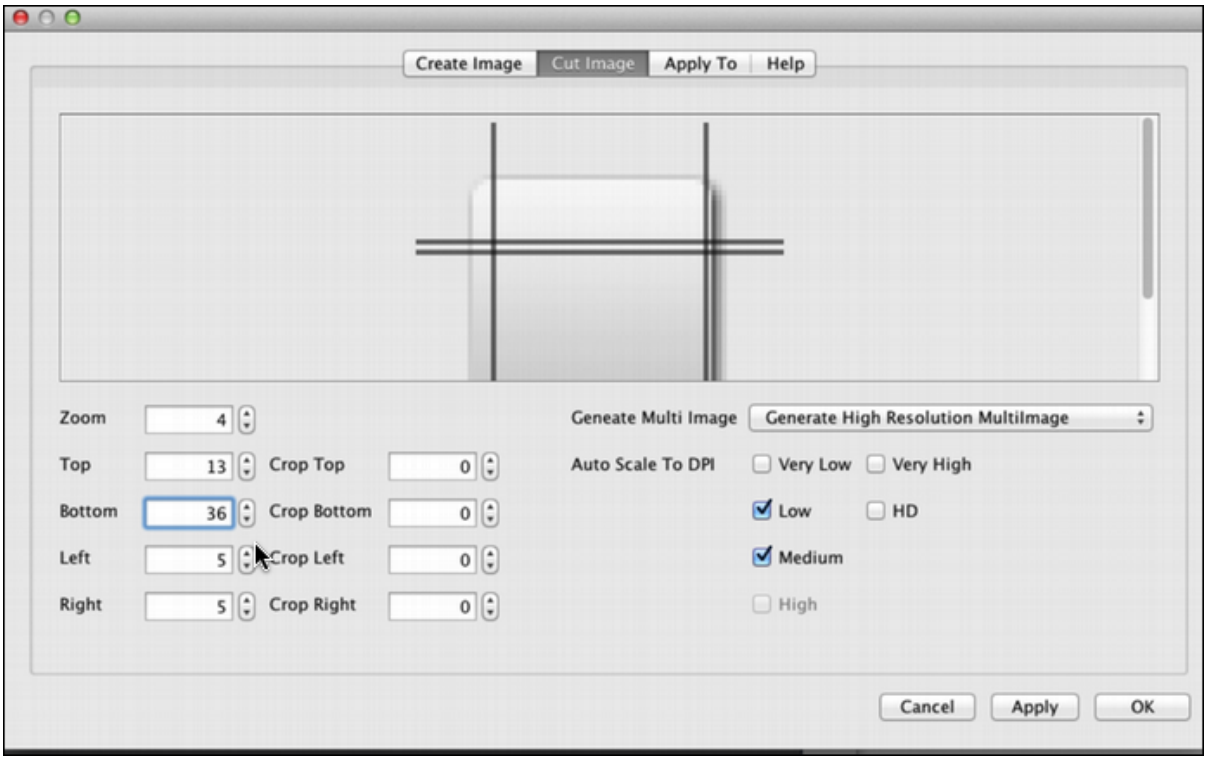
Borders are a remarkably powerful tool for customizing a Component. The most powerful approach is the 9-piece image border which is easiest to use when using the Image Border Wizard.



The image border wizard allows you to take an image and "cut it" into 9 distinct pieces: 4 corners, top, bottom, left, right & center.
The corners are placed as usual in the edges of the component and the other elements are tiled to fill up the available space.



Its important when using a gradient effect within the image border to make sure the center piece is as narrow as possible to avoid a case of a "broken" gradient.



Advanced Theming

This chapter covers the advanced concepts of theming as well as deeper understanding of how to build/design a Codename One Theme.

Understanding Codename One Themes

Codename One themes are pluggable CSS like elements that allow developers to determine/switch the look of the application in runtime. A theme can be installed via the UIManager class and themes can be layered one on top of the other (like CSS).

By default Codename One themes derive the native operating system themes although this behavior is entirely optional.

A theme initializes the Style objects which are then used by the components to render themselves or by the LookAndFee/DefaultLookAndFeel class to create the appearance of the application.

Codename One themes have some builtin defaults e.g. borders for buttons and padding/margin/opacity for various components. These are a set of “common sense” defaults that can be overridden within the theme.

Working With UIID

UIID’s (User Interface IDentifier) are effectively a unique name given to a UI component that allow associating a set of theme definitions with a specific component. The class name of the component is commonly the same as the UIID but they are separate entities for a few important reasons.

One of the biggest advantages with UIID’s is the ability to change the UIID of a component, e.g. to create a multiline label one can use something like:

```
TextArea t = …;
t.setUIID("Label");
t.setEditable(false);
```

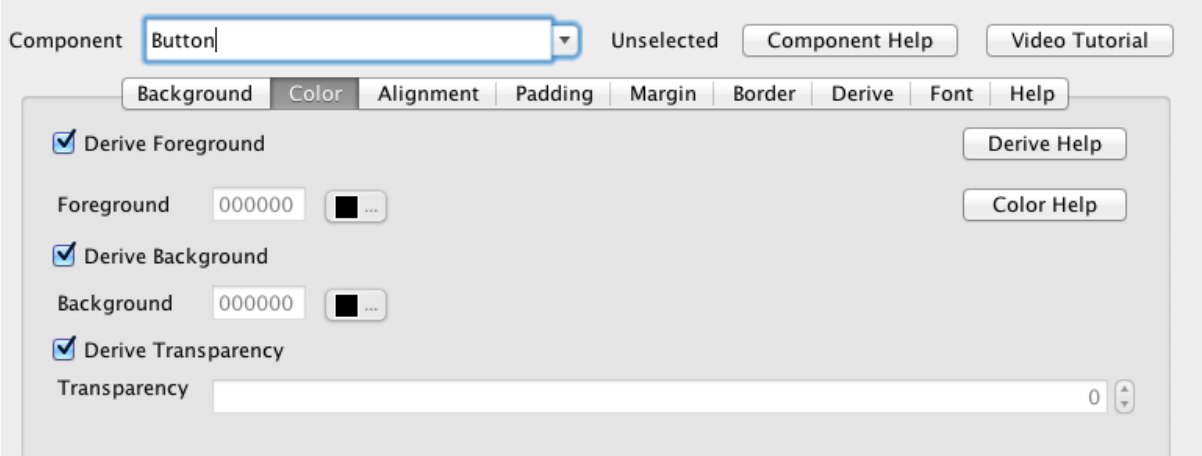
UIID can be customized via the GUI builder and allows for powerful selection of individual components.

Style Inheritance

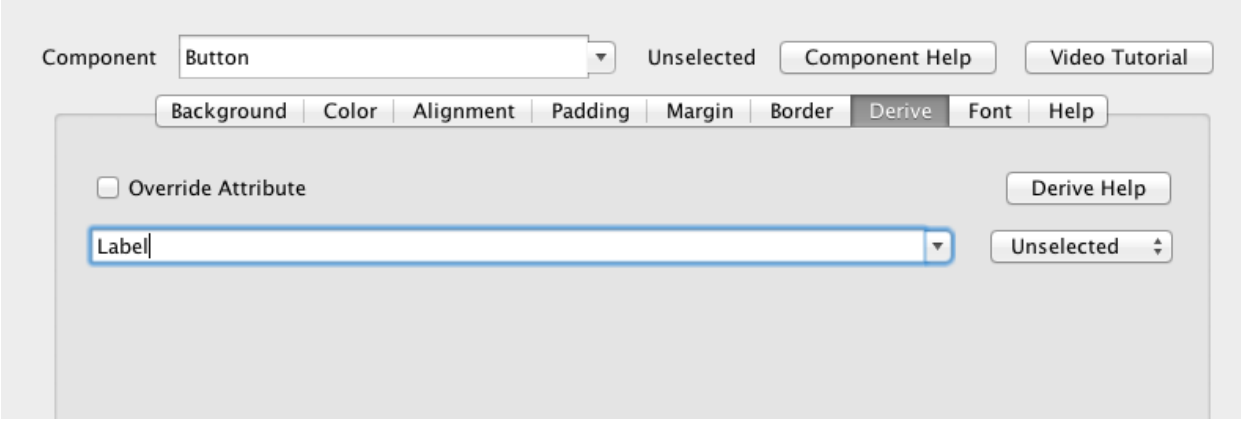
There are multiple defaults defined for elements within Codename One, e.g. a Container defaults to zero padding/margin and full transparency.

By default every style inherits from these common cross platform defaults. Other than that a theme may derive from the platform native theme providing further defaults.

In the designer we can define whether a specific entry derives from the global default or defines its own override:



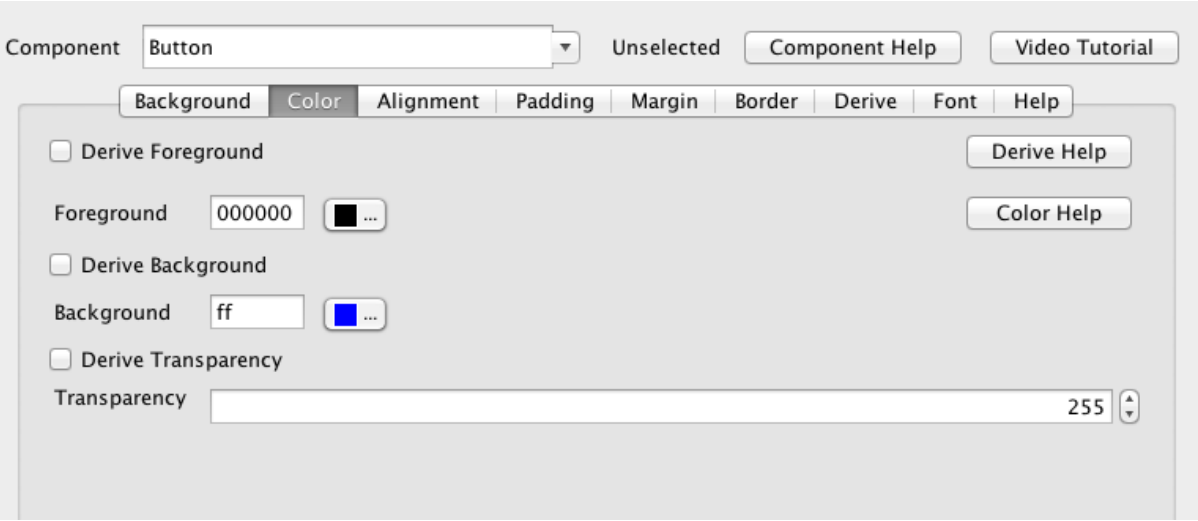
By unchecking the derive flag we can override the appearance of a specific entry within the style.



The Derive entry within the style allows a specific style to derive and extend another style, e.g. a Button selected style can derive from a Button unselected style thus removing the need to redefine style behavior for every state. This is very useful when defining commonalities among entries in a theme.

Colors & Transparency

The colors of the style are represented as web style RGB entries, the foreground color is usually used to draw text. The background color applies when a border isn't defined.
Transparency is only used to draw the background color when applicable.



Backgrounds

The background tab is one of the more elaborate entries within the style section. Notice that a background will only apply when no border is defined.

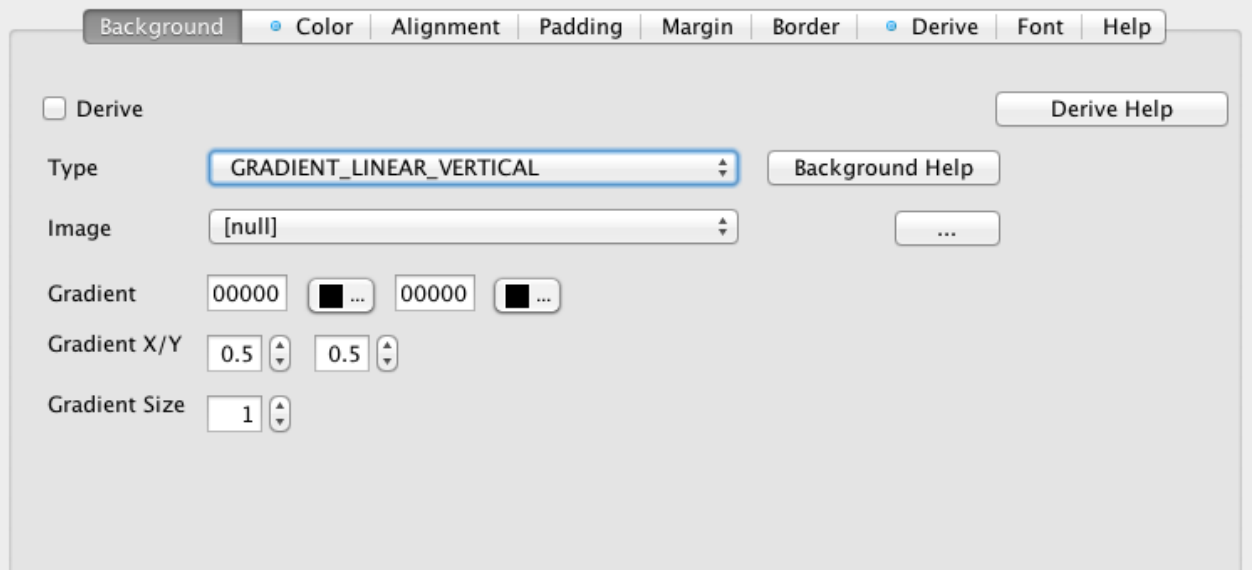
In general 2 types of background are supported: Image or Gradient. You can pick several modes for each. For the gradients you can pick one of horizontal, vertical or radial gradient.

In the case of gradients you need to define all the gradient variables mentioned within the screen to define the source/destination colors and in the case of a radial the properties relevant to that.

When using an image background there are basically three options: Scaled, Tiled or Aligned.

An image can be scaled across the background of the component, this can create a decent effect in some cases but often causes some distortion.

An image can be aligned to a specific location within the component background, in which case the component will be painted based on the transparency setting and the image will be painted in the appropriate location based on the alignment.
Finally an image can be tiled either completely over the background or tiled as a single row/column aligned to a specific area within the component.



Fonts

Codename One supports 2 major font types[2] system fonts and truetype fonts. The system fonts are very limited in selection but are highly recommended for portability. They use the built in font on the given device which is often the font the user is used to. A developer has a selection from one of 3 generic sizes small, medium or large and basic style choices: bold/plain/italic.

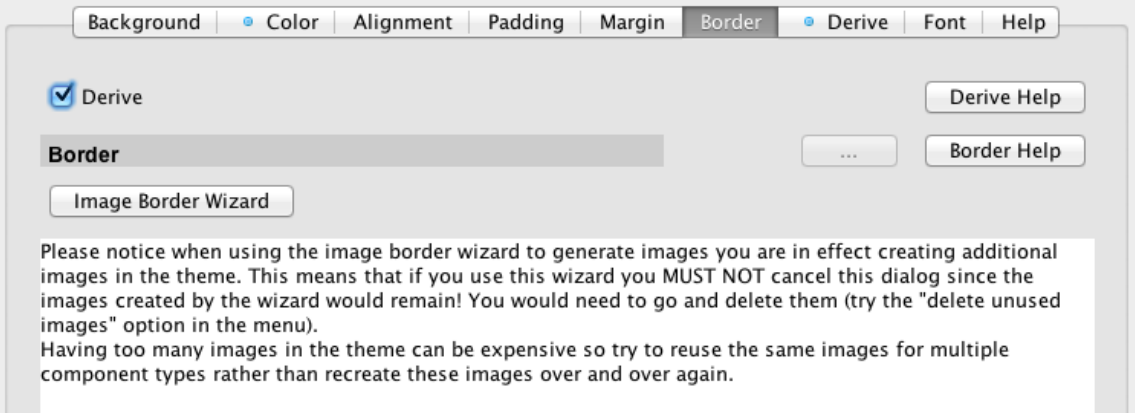
[2] Historically bitmap fonts were also supported, however this functionality is deprecated and might be removed in a future revision. We highly recommend you avoid using the bitmap font functionality.

When a specific truetype font is needed Codename One allows the developer to place a truetype font within the src directory of the project. This font will be automatically detected by the designer tool and offered as an option. Since truetype fonts are only supported on iOS, Android and RIM devices you should still define a system font which will be used as a fallback on unsupported platforms.
Truetype fonts allow specifying their sizes using one of 3 approaches:

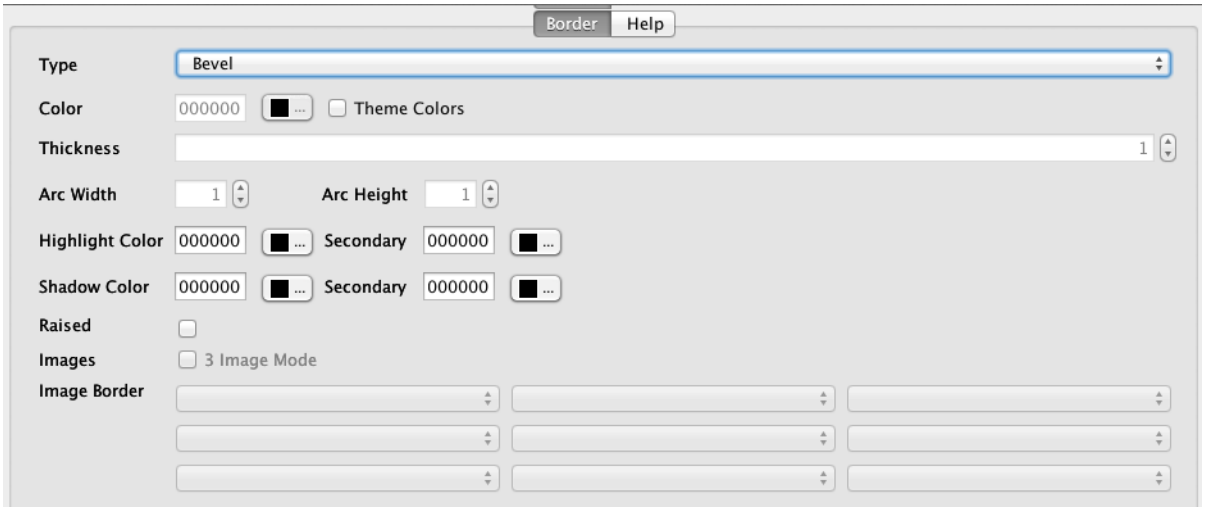
1. System font size scheme - the truetype font will have the same size as a small, medium or large system font. This allows the developer to size the font based on the device DPI.
2. Millimeter size - allows sizing the font in a more DPI aware size.
3. Pixels - a font can be sized in pixels, which is useful for some unique cases but highly problematic in multi-DPI scenarios.

Borders

See the theme basics above about cutting a 9-piece border using the image border class.
Codename One supports configuring the border according to several configurations:



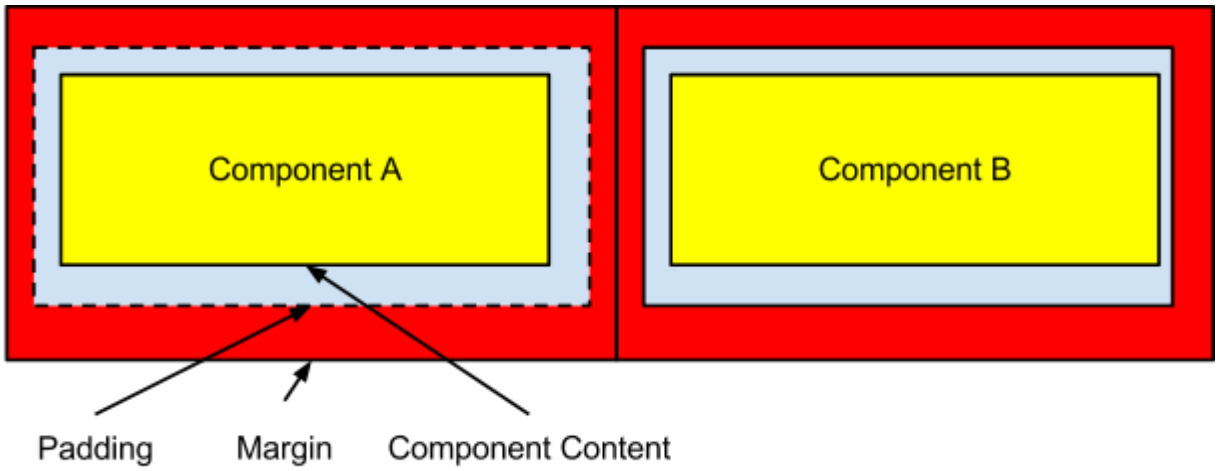
When pressing the '...' button you can customize the border using multiple configuration types:



There are several border types that can be used to customize the look of the component.

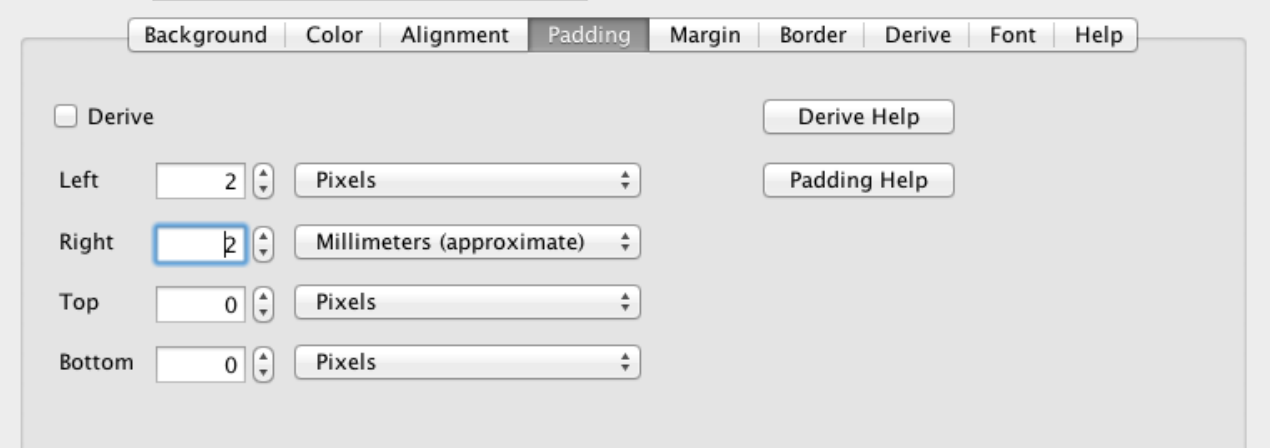
Padding/Margin

Padding and margin are concepts derived from the CSS box model. Its slightly different in Codename One where the border spacing is part of the padding but other than that they are pretty similar:



In the above diagram we can see the component represented in yellow occupying its preferred size. The padding portion in blue effectively increases the components size. The margin is an area which effectively belongs to the component but the component doesn't draw anything within that area it is represented in red.

The theme allows us to customize the padding/margin and specify them for all 4 sides of a component. They can be specified in pixels, millimeters, or screen percentage:



Padding is especially important when a border is defined and we need to space the component drawing from the edge of the border. Margin allows us to space components from one another easily and create “whitespace” within the user interface.

Theme Constants

The Codename One Designer has a tab for creating constants which can be used to add global values of various types and behavior hints to Codename One and its compoennts. Constants are always strings, they have some conventions where a constant ending with Bool is treated as a boolean true/false value and a constant ending with Int or Image (for image the string name of the image is stored but the image instance will be returned).

To use a constant one can use the UIManager's methods to get the appropriate constant type specifically:
getThemeConstant
isThemeConstant
getThemeImageConstant

Internally Codename One has several built in constants and the list is constantly growing as we add features to Codename One, we will try to keep this list up to date when possible.

Constant	Description/Argument
alwaysTensileBool	Enables tensile drag even when there is no scrolling in the container (only for scrollable containers though)

defaultCommandImage	Image to give a command with no icon
dialogButtonCommandsBool	Place commands in the dialogs as buttons
dialogPosition	Place the dialog in an arbitrary border layout position (e.g. North, South, Center etc.)
centeredPopupBool	Popup of the combo box will appear in the center of the screen checkBoxCheckDisImage CheckBox image to use instead of Codename One drawing it on its own
checkBoxCheckedImage	CheckBox image to use instead of Codename One drawing it on its own
checkBoxUncheckDisImage	CheckBox image to use instead of Codename One drawing it on its own
checkBoxUncheckedImage	CheckBox image to use instead of Codename One drawing it on its own
comboImage	Combo image to use instead of Codename One drawing it on its own
commandBehavior	Indicates how commands should act, as a touch menu, native menu etc. Possible values: SoftKey, Touch, Bar, Title, Right, Native
ComponentGroupBool	Enables component group which allows components to be logically grouped together so the UIID's of components would be modified based on their group placement. This allows for some unique styling effects where the first/last elements have different styles from the rest of the elements. Its disabled by default thus leaving its usage up to the designer.
dialogTransitionIn	Default transition for dialog
dialogTransitionInImage	Default transition Image for dialog, causes a Timeline transition effect
dialogTransitionOut	Default transition for dialog
dialogTransitionOutImage	Default transition Image for dialog, causes a Timeline transition effect
disabledColor	Color to use when disabling entries by default

dlgCommandButtonSizeInt	Minimum size to give to command buttons in the dialog
dlgCommandGridBool	Places the dialog commands in a grid for uniform sizes
dlgSlideDirection	Slide hints
dlgSlideInDirBool	Slide hints
dlgSlideOutDirBool	Slide hints
fadeScrollBarBool	Boolean indicating if the scrollbar show fade when there is inactivity
fadeScrollEdgeBool	Places a fade effect at the edges of the screen to indicate that its possible to scroll until we reach the edge (common on Android).
fadeScrollEdgeInt	Amount of pixels to fade out at the edge
firstCharRTLBool	Indicates to the GenericListCellRenderer that it should determine RTL status based on the first character in the sentence
fixedSelectionInt	Number corresponding to the fixed selection constants in List
formTransitionIn	Default transition for form
formTransitionInImage	Default transition Image for form, causes a Timeline transition effect
formTransitionOut	Default transition for form
formTransitionOutImage	Default transition Image for form, causes a Timeline transition effect
hideEmptyTitleBool	Indicates that a title with no content should be hidden even if the border for the title occupies space
ignorListFocusBool	Hide the focus component of the list when the list doesn't have focus
listItemGapInt	Builtin item gap in the list, this defaults to 2 which predated padding/margin in Codename One
menuHeightPercent	Allows positioning and sizing the menu

menuPrefSizeBool	Allows positioning and sizing the menu
menuSlideDirection	Defines menu entrance effect
menuSlideInDirBool	Defines menu entrance effect
menuSlideOutDirBool	Defines menu entrance effect
menuTransitionIn	Defines menu entrance effect
menuTransitionInImage	Defines menu entrance effect
menuTransitionOut	Defines menu exit effect
menuTransitionOutImage	Defines menu entrance effect
menuWidthPercent	Allows positioning and sizing the menu
otherPopupRendererBool	Indicates that a separate renderer UIID/instance should be used to the list within the combo box popup
PackTouchMenuBool	Enables preferred sized packing of the touch menu (true by default), when set to false this allows manually determining the touch menu size using percentages
popupCancelBodyBool	Indicates that a cancel button should appear within the combo box popup
popupTitleBool	Indicates that a title should appear within the combo box popup
pureTouchBool	Indicates the pure touch mode
radioSelectedDisImage	Radio button image
radioSelectedImage	Radio button image
radioUnselectedDisImage	Radio button image
radioUnselectedImage	Radio button image
rendererShowsNumbersBool	Indicates whether renderers should render the entry number
reverseSoftButtonsBool	Swaps the softbutton positions
slideDirection	Default slide transition settings

slideInDirBool	Default slide transition settings
slideOutDirBool	Default slide transition settings
snapGridBool	Snap to grid toggle
tabPlacementInt	The placement of the tabs in the Tabs component: TOP = 0, BOTTOM = 2, LEFT = 1, RIGHT = 3
tabsFillRowsBool	Indicates if the tabs should fill the row using flow layout
tabsGridBool	Indicates whether tabs should use a grid layout thus forcing all tabs to have identical sizes
textCmpVAlignInt	The vertical alignment of the text component: TOP = 0, CENTER = 4, BOTTOM = 2
textFieldCursorColorInt	The color of the cursor as an integer (not hex)
tickerSpeedInt	The speed of label/button etc. tickering in ms.
tintColor	The aarrggbb hex color to tint the screen when a dialog is shown
touchCommandFillBool	Indicates how the touch menu should layout the commands within
touchCommandFlowBool	Indicates how the touch menu should layout the commands within
transitionSpeedInt	Indicates the default speed for transitions
tensileDragBool	indicates that tensile drag should be enabled/disabled. This is usually set by platform themes.

One "odd" behavior of constants is that once they are set by a theme they don't get "lost" when replacing the theme. E.g. if one would set the comboImage constant to a specific value in theme A and then switch to theme B that doesn't define the comboImage, the original theme A comboImage might remain. The reason for this is simple, when extracting the constant values components keep the values in cache locally and just don't track the change in value. Furthermore, since the components allow manually setting values its impractical for them to track whether a value was set by a constant or explicitly by the user.
The solution for this is to either manually reset undesired values before replacing a theme (e.g. for the case above by calling the default look and feel method for setting the combo image with a null value) or defining a constant value to replace the existing value.

How Does A Theme Work

Codename One themes are effectively a simple hashtable containing key/value pairs. Such a hashtable is passed on to the setThemeProps() method of the UIManager (or one of its equivalents e.g. addThemeProps()) to install a theme.
When a Codename One component is rendered or laid out, style related data is requested, in which case the UIManager generates a Style object based on the theme hashtable values.

A theme hashtable key is comprised of:
[UIID.][type#]attribute

The UIID, corresponds to the component's UIID e.g. Button, CheckBox etc. It is optional and may be omitted to address the global default style.
The type is omitted for the default unselected type and may be one of sel (selected type), dis (disabled type) or press (pressed type). The attribute should be one of:

- derive - the value for this attribute should be a string representing the base component.
- bgColor - represents the background color for the component if applicable in a web hex string format RRGGBB e.g. ff0000 for red.
- fgColor - represents the foreground color if applicable.
- border - an instance of the border class, used to display the border for the component.
- bgImage - an Image object used in the background of a component
- transparency - a String containing a number between 0-255 representing the alpha value for the background. This only applies to the bgColor.
- margin - the margin of the component as a String containing 4 comma separated numbers for top,bottom,left,right.
- padding - the padding of the component, it has an identical format to the margin attribute.
- font - A Font object instance
- alignment - an Integer object containing the LEFT/RIGHT/CENTER constant values defined in Component.
- textDecoration - an Integer value containing one of the TEXT_DECORATION_* constant values defined in Style.
- backgroundType - a Byte object containing one of the constants for the background type defined in Style under BACKGROUND_*.
- backgroundGradient - contains an Object array containing 2 integers for the colors of the gradient. If the gradient is radial it contains 3 floating points defining the x, y & size of the gradient.

So to set the foreground color of a selected button to red a theme will define a property like:
Button.sel#fgColor=ff0000

This information is mostly useful for understanding how things work within Codename One, but it can also be useful in runtime.

E.g. to increase the size of all fonts in the application we can do something like:

```
Hashtable h = new Hashtable();
h.put("font", largeFont);
UIManager.getInstance().addThemeProps(h);
Display.getInstance().getCurrent().refreshTheme();
```

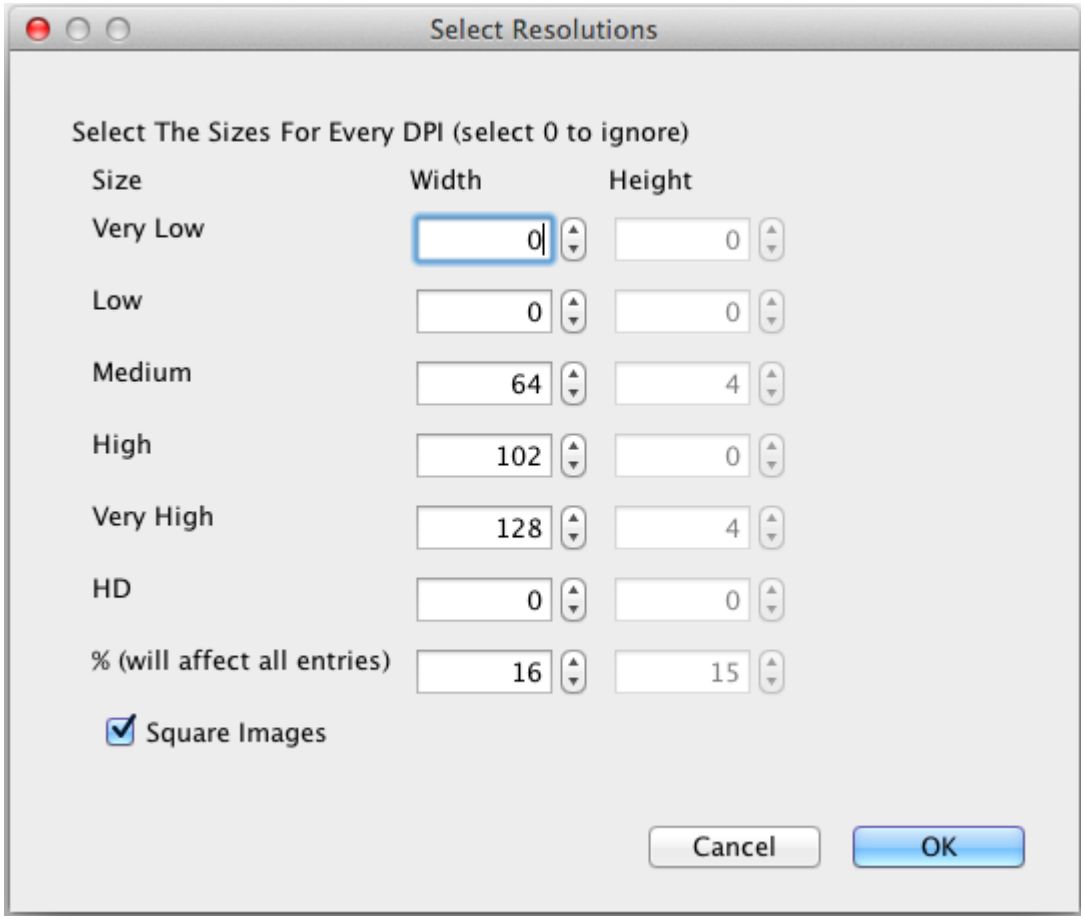
Understanding Images & Multi-Images

When working with a theme we often use images for borders or backgrounds. We also use images within the GUI for various purposes and most such images will be extracted from the resource file.
Adding a standard JPEG/PNG image to the resource file is straightforward and it can be viewed within the images section. However, due to the wide difference between device types an image that would be appropriate in size for an iPhone 3gs would not be appropriate in size for a Nexus device or an iPhone 4 (but perhaps surprisingly it will be just right for iPad 1 & iPad 2).
The reason for this is DPI or device density, the density of the devices varies significantly and Codename One tries to make matters simple by unifying everything into one set of values to indicate density. For simplicities sake density is expressed in terms of pixels but it is mapped internally to actual screen measurements where possible.

A multi-image is an image that has multiple varieties for different densities and thus looks much better on all the different resolutions. Since scaling on the device can't interpolate the data (due to performance considerations) scaling on the device becomes impractical. However, a multi-image will just provide the "right" resolution image for the given device type.
From the programming perspective this is completely seamless, a developer just access one image and has no ability to access the images in the different resolutions. Within the designer however, we can explicitly define images for multiple resolutions and perform high quality scaling so the "right" image is available.

To add a multi-image we can use two basic methods: quick add & standard add.
Both rely on understanding the source resolution of the image, e.g. you have an icon that you expect to be 128x128 pixels on iPhone 4, 102x102 on nexus one and 64x64 on iPhone 3gs. You can provide the source image as the 128 pixel image and just perform a quick add option while picking the Very High density option as an option.
This will indicate to the algorithm that your source image is designed for very high density and it will scale for the rest of the densities accordingly.

Alternatively you can use the standard add multi-image dialog and set it like this:



Notice that I selected square images essentially eliminating the height option. Setting values to 0 prevents the system from generating a multi-image entry for that resolution, which will mean a device in that category will fall on the closest alternative.

The percentage value will change the entire column and it means the percentage of the screen. E.g. We know the icon is 128 for the very high resolution, we can just move the percentage until we reach something close to 128 in the “Very High” row and the other rows will represent a size that should be pretty close in terms of physical size to the 128 figure.

Working With The GUI Builder

Basic Concepts

The basic premise is this: the designer creates a UI version and names GUI components, he can create commands as well including navigation commands, exit, minimize. He can also define a long running command which will by default trigger a thread to execute...

All UI elements can be loaded using the UIBuilder class. Why not just use the Resources API?

Since the Resources class is essential for using Codename One, adding the UIBuilder as an import would cause any application (even those that don't use the UIBuilder) to increase in size! We don't want people who aren't using the feature to pay the penalty for its existence!

The UI Builder is designed for use as a state machine carrying out the current state of the application so when any event occurs a subclass of the UIBuilder can just process it. The simplest way and most robust way for changes is to use the Codename One Designer to generate some code for you (yes we know its not a code generation tool but there is a hack...).

When using a GUI builder project it will constantly regenerate the state machine base class which is a UIBuilder subclass containing most of what you need...
The trick is not to touch that code! DO NOT CHANGE THAT CODE!

Sure you can change it and everything will be just fine, however it you will make changes to the GUI regenerating that file will obviously lose all those changes which is not something you want! To solve it you need to write your code within the State machine class which is a subclass of the state machine base class and just override the appropriate methods, then when the UI changes the GUI builder just safely overwrites the base class since you didn't change anything there...

The Components Of Codename One

This chapter covers the components of Codename One, not all are covered but it tries to go deeper than the JavaDocs.

Container

The Codename One container is a base class for many high level components, a container is basically a component that can contain other components. That is all.
Every component has a parent container which can be null if it isn't within a container at the moment or is a top level container. A container can have many children.
Components are arranged in containers using layout managers which are just algorithms to determine the flow within a specific container.
You can read more about layout managers in the section below dedicated to layout managers.
The default layout of a Container is flow layout which is useful for simple types of layouts but problematic with many scenarios. We recommend you read about this in the layout manager section.

Form

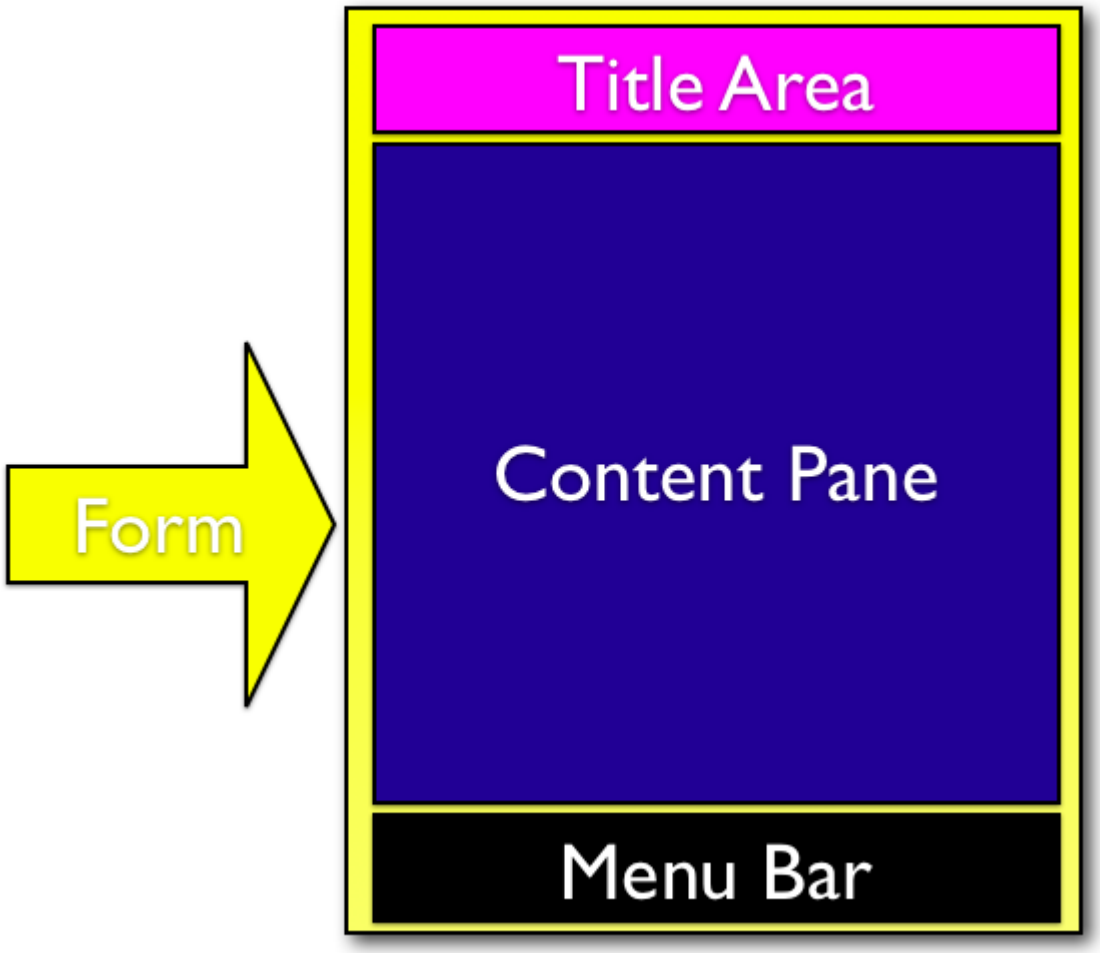
The top level container of Codename One. Form derives container and it is effectively the elemnt we "show". Only one form can be visible at any given time and we can get the currently visible form using the code:

```
Form currentForm = Display.getInstance().getCurrent();
```

A form is a unique container in the sense that it has a title, a content area and optionally a menu/menu bar area. When invoking methods such as add/removeComponent on a form you are in fact invoking something that looks like:

```
myForm.getContentPane().addComponent(...);
```

Which makes sense since a form is really just a Container that has a border layout, its north section is occupied by the title area and its south section by the optional menu bar. The center (which stretches) is the content pane. The content pane is where you place all your components.



There is one important piece missing from this image which is the glass pane (more on that soon), but basically you can see that every form has space allocated for the title/menu bar. If you don't set the title if won't show up (its size will be zero) but it will still be there. The same isn't always true for the case of the menu bar which can vary significantly. Effectively the section that matters is the content pane, so the form tries to do the "right thing" by pretending to be the content pane. However, this isn't always seamless and sometimes code needs to just invoke getContentPane() in order to work directly with the container.

Form's have an additional layer painted on top of them called the glass pane, this allows developers to overlay UI on top of existing UI and paint as they see fit. This is useful for things that provide notification but don't want to intrude with application functionality.

Dialog

A dialog is a special kind of form that can occupy only a portion of the screen, it also has the additional functionality of the modal show method. When showing a dialog we have two basic options modless and modal. Modal dialogs (the default) block the current EDT thread until the dialog is dismissed (to understand how they do it read about invokeAndBlock).
Modal dialogs are an extremely useful way to prompt the used since the code has the users response on the next line of execution promoting a very linear way of coding.
The dialog class contains multiple static helper methods to quickly show user notifications, but also allows a developer to create a dialog instance, add information to its content pane and show the dialog. When showing a dialog in this way you can either ask Codename One to position the dialog in a specific general location (taken from the BorderLayout concept for locations) or position it by spacing it (in pixels) from the 4 edges of the screen.

Label

Label allows drawing text or placing an icon, it is a single line label that might ticker and might end with "..." in some cases where appropriate. Developers can determine the placement of the label relatively to its icon in quite a few powerful ways.
Label serves as the base class for button which inherits all its functionality.

Label doesn't break lines since line break support is expensive in terms of CPU, however TextArea does break lines and a common use case for creating a multi-line label is to set the UIID of text area to "Label" and invoke its setEditable(false) method.

Button

A button is a subclass of label and so it inherits much of its functionality, specifically icon placement, tickering etc.
Besides being "pressable" which is a unique feature to Button and its subclasses, it also allows for icons based on its states (pressed, hover etc.).

CheckBox/Radio Button

CheckBox & Radio Button are both subclasses of button that allow for either a toggle state or exclusive selection state. Both of these components can be displayed as toggle buttons (see the toggle button section below) or just use the default check mark/filled circle appearance based on the type/OS.

Multi-Button

A multi button is a special component that allows button like functionality (although it technically derives from Container) with more advanced features. It supports up to 4 lines of text (it doesn't automatically wrap the text), an emblem (usually navigational arrow, or check box) and an icon.
It can be used as a button, a label, a checkbox or a radio button and allows creating richer UI's.

The MultiButton is meant to provide developers with the ability to replicate some of the UI paradigms that are common to the UITableView iOS UI's. The MultiButton doesn't include anything radically special, its just a standard container with a Lead Component based UI.

The MultiButton is mostly designed for use with the GUI builder although using it from code is similar. A common source of confusion is the difference between the icon and the emblem since both may have an icon image associated with them. The icon is an image representing the entry while the emblem is an optional visual representation of the action that will be undertaken when the element is pressed. Both may be used simultaneously or individually of one another.

TextField/TextArea

The TextField class derives from the TextArea class and both are the mechanisms for user text input in Codename One. The semantic difference between the two classes dates back to the roots of Codename One in LWUIT where feature phones don't have "proper" in-place editing capabilities.
Text field allowed input on various device types without opening the native full screen editing facilities. This is not used in any smartphone platform other than Symbian. On those platforms TextField and TextArea are practically identical. Both provide multi-line editing capabilities etc. One small difference in text field is the blinking cursor animation which doesn't appear in the text area implementation when it isn't edited.

A common alternative use case of text area is as a multi-line label, to understand more about that please read the label section above.

Toggle Button

A toggle button is a button that is pressed and then stays pressed, when pressed again it's released. Hence the button has a selected state to indicate if it's pressed or not. Just like the radio button or checkbox components in Codename One. So Codename One's new toggle buttons are really any button (checkbox & radio buttons derive from Button) that has the setToggle() method invoked with true. It thus paints itself with the toggle button style unless explicitly defined otherwise (note that the UIID will be implicitly changed to "ToggleButton").

The cool thing about this is that you can effectively take your knowledge about checkboxes & radio buttons and apply it to toggle buttons.

That's half the story though, to get the full effect of some cool toggle button UI's we would like to assign the buttons on the edges with a rounded feel like some platforms choose to do... That's pretty easy, you can just assign a different UIID to the first/last buttons and be over with it.

But what if you want your code to be generic? After all you might add/remove a button in runtime based on application state and you would like it to have the right style.

To solve this we introduced the ComponentGroup.

The ComponentGroup is a special container that can be either horizontal or vertical (Box X or Y respectively), it also does nothing else. You need to explicitly activate it in the theme by setting a theme property to true (by default you need to set ComponentGroupBool to true).

When ComponentGroupBool is set to true the component group will modify the styles of all components placed within it to match the element UIID given to it (by default GroupElement) with special caveats to the first/last/only elements. E.g.
1. If I have one element within a component group it will have the UIID: GroupElementOnly
2. If I have two elements within a component group they will have the UIID's GroupElementFirst, GroupElementLast
3. If I have three elements within a component group they will have the UIID's GroupElementFirst, GroupElement, GroupElementLast
4. If I have four elements within a component group they will have the UIID's GroupElementFirst, GroupElement, GroupElement, GroupElementLast

You get the picture... This allows you to define special styles for the sides (don't forget to use the derive attribute to generalize your theme) and provide toggle buttons that include special effects simply by placing them into this group.

You can customize the UIID set by the component group by calling setElementUIID in the component group e.g. setElementUIID("ToggleButton") for the picture above will result in: ToggleButtonFirst, ToggleButton, ToggleButtonLast

This is a short sample:

```
ComponentGroup buttons = new ComponentGroup();
buttons.setElementUIID("ToggleButton");
buttons.setHorizontal(true);
RadioButton plain = new RadioButton("Plain");
RadioButton underline = new RadioButton("Underline");
RadioButton strikeout = new RadioButton("Strikethru");
ButtonGroup bg = new ButtonGroup();
initRb(bg, buttons, listener, plain);
initRb(bg, buttons, listener, underline);
initRb(bg, buttons, listener, strikeout);
Container centerFlow = new Container(new FlowLayout(Component.CENTER));
f.addComponent(centerFlow);
centerFlow.addComponent(buttons);

private void initRb(ButtonGroup bg, Container buttons, ActionListener listener, RadioButton rb)
{
  bg.add(rb);
  rb.setToggle(true);
  buttons.addComponent(rb);
  rb.addActionListener(listener);
}
```

List, ContainerList, Renderers & Models

Warning: This is a rather complex chapter. If you are just interested in creating a simple list we suggest you skip ahead to the MultiList section.
A Codename One list doesn't contain components but rather arbitrary data, this seems odd at first but makes perfect sense... If you want a list to contain components just use a Container. The advantage of using a List in this way is that we can display it in many ways (e.g. fixed focus positions, horizontally etc.) and that we can have more than a million entries without performance overhead. We can also do some pretty nifty things like filter the list on the fly or fetch it dynamically from the internet as the user scrolls down the list.
To achieve these things the list uses two interfaces: ListModel and ListCellRenderer.

List model represents the data, its responsibility is to return the arbitrary object within the list at a given offset. Its second responsibility is to notify the list when the data changes so the list can refresh, think of it as an array of objects that can notify you when you get changes.
The list renderer is like a rubber stamp that knows how to draw an object from the model, its called many times per entry in an animated list and must be very fast. Unlike standard LWUIT components it is only used to draw the entry in the model and immediately discarded hence it has no memory overhead but if it takes too long to process a model value it can be a big bottleneck!
This is all very generic but a bit too much for most, doing a list "properly" requires some understanding. The main source of confusion for developers is the stateless nature of the list and transfer of state to the model (e.g. a checkbox list needs to listen to action events on the list and update the model in order for the renderer to display that state... Once you understand that it's easy).

Important - Lists & Layout Managers

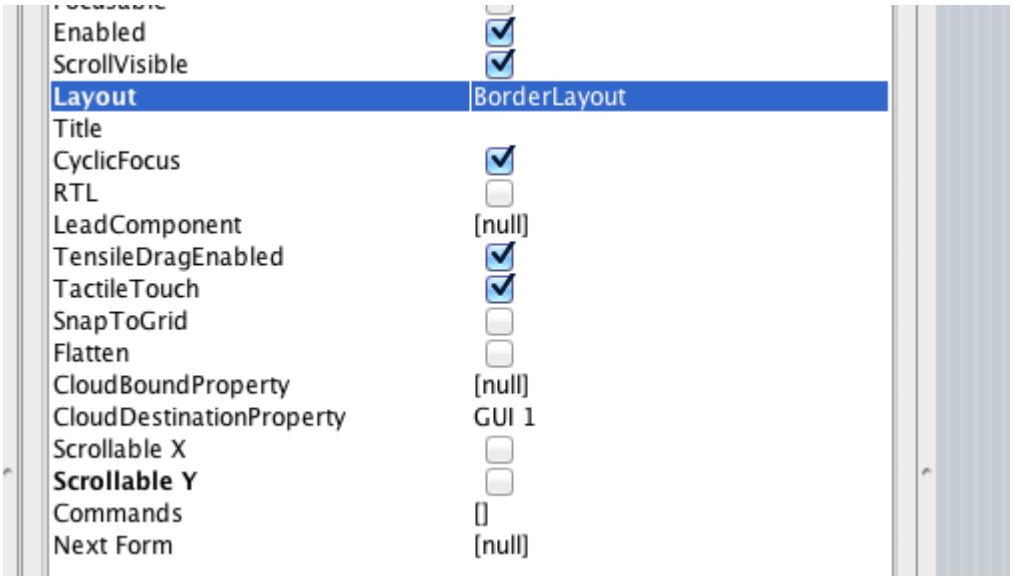
Usually when working with lists you want the list to handle the scrolling (otherwise it will perform badly). This means you should place the list in a non-scrollable container (no parent can be scrollable), notice that the content pane is scrolled by default so you should disable that.
It is also recommended to place the list in the CENTER location of a BorderLayout to produce the most effective results. e.g.:

```
form.setScrollable(false);
form.setLayout(new BorderLayout());
form.addComponent(BorderLayout.CENTER, myList);
```

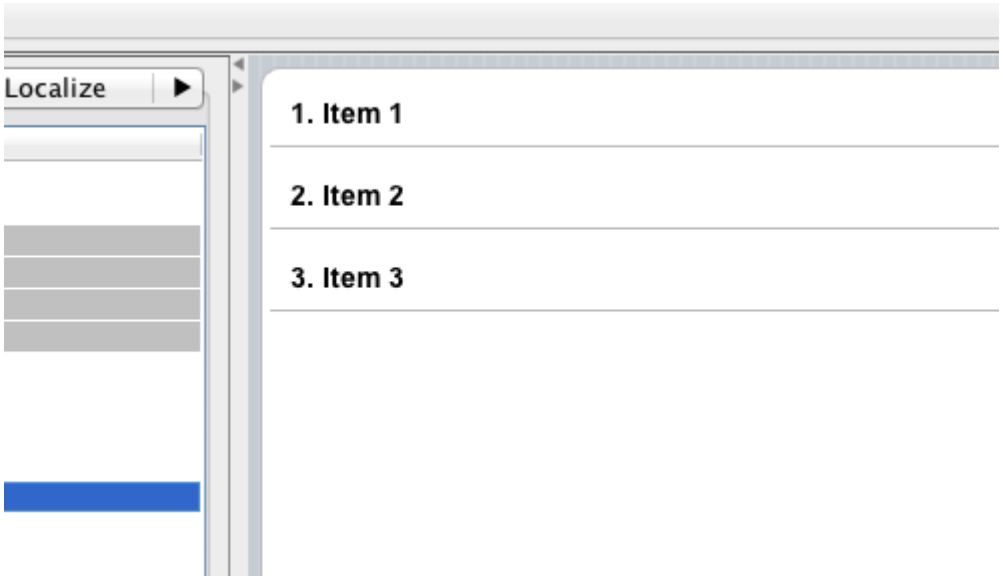
Using Lists In The GUI Builder

The Codename One GUI builder provides several simplifications to the concepts outlined below, you can build all portions of the list through the GUI builder or build portions of them using code if you so desire.
This is a step by step guide with explanations on how to achieve this. Again you might prefer using the MultiList component mentioned below which is even easier to use.

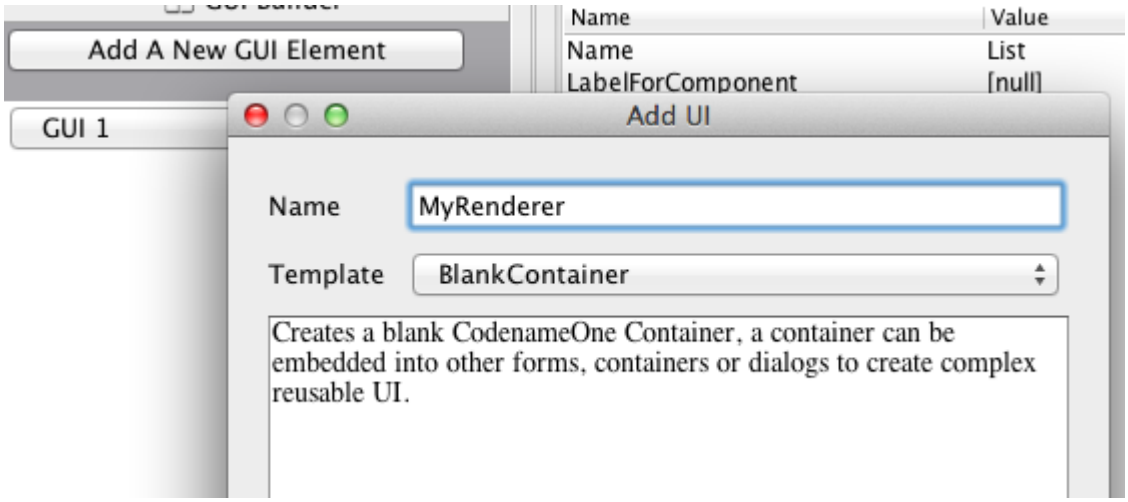
Start by creating a new GUI form, set its scrollable Y to false and set its layout to border layout



Next drag a list into the center of the layout and you should now have a functioning basic list with 3 items.

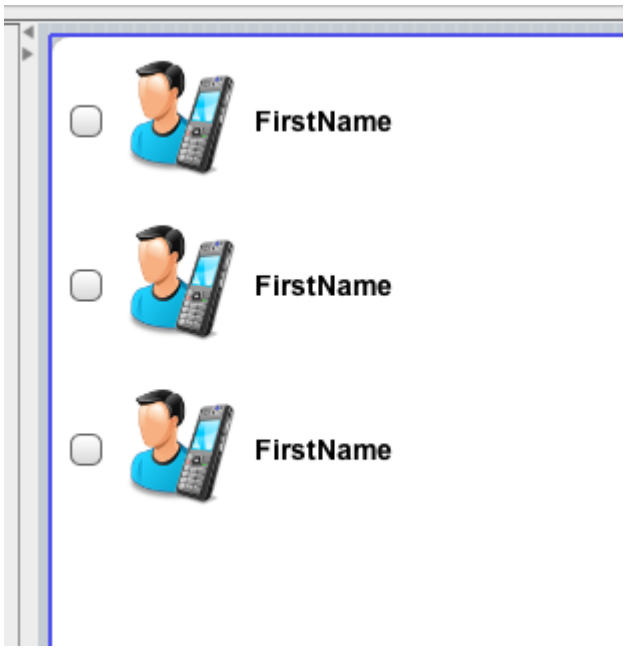


Next we will create the Renderer which indicates how an individual item in the list appears, start by pressing the Add New GUI Element button and **select the "Blank Container"** option! Fill the name as "MyRenderer" or anything like that.



Within the renderer you can drag any component you would like to appear, labels, checkboxes, etc. You can nest them in containers and layouts as you desire. Give them names that are representative of what you are trying to accomplish e.g. Firstname, selected etc.

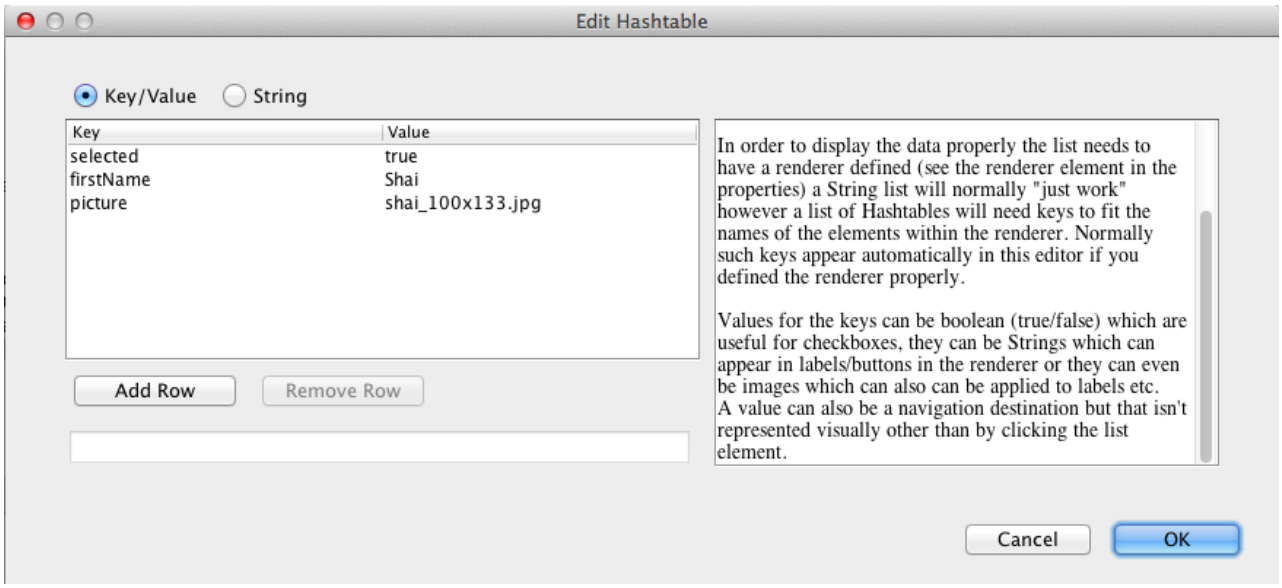
You can now go back to the list, select it and click the Renderer property in order to select the render container you created previously resulting in something like this.



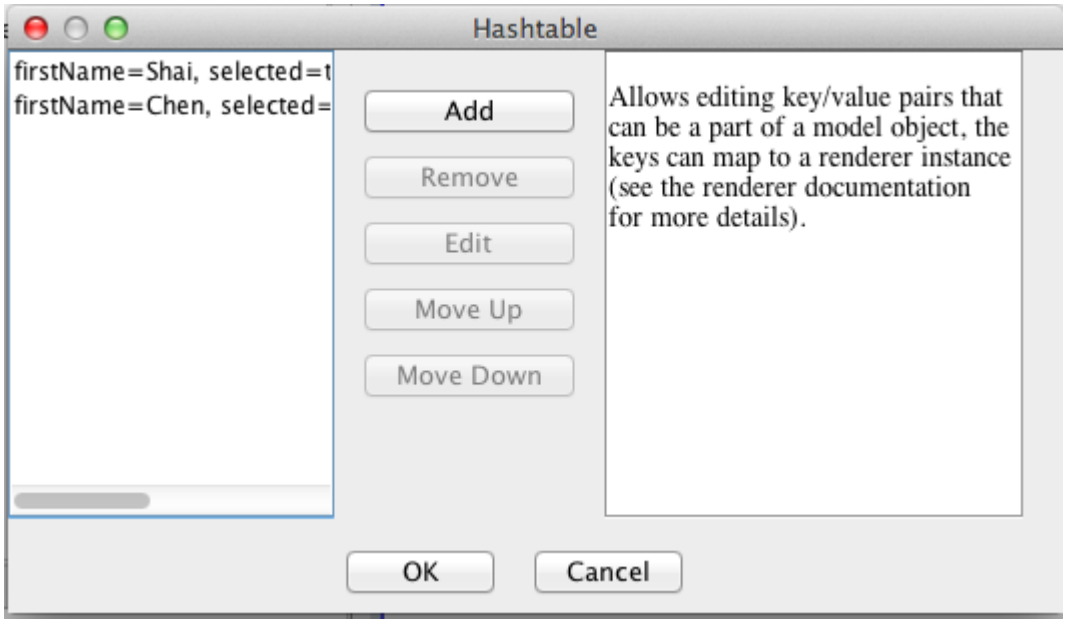
You'll notice that the data doesn't match the original content of the list, that is because the list contains strings instead of Hashtables. To fix this we must edit the list data.

You can place your own data in the list using the GUI builder which is generally desired regardless of what you end up doing since this allows you to preview your designer in the GUI builder.
If you wish to populate your list from code just click the events tab and press the ListModel button, you can fill up the model with an array of Hashtables as we explain soon enough (you can read more about the list model below).

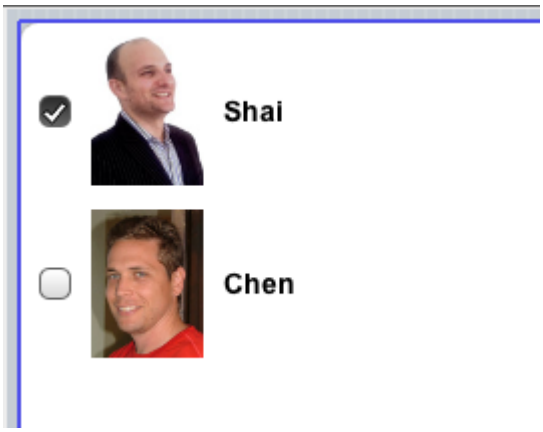
To populate the list via the GUI builder click the properties of the list and within them click the ListItems entry. The entries within can be strings or Hashtables, however in order to be customizable in the rendering stage we will need them all to be Hashtables. Remove all the current entries and add a new entry:



After adding two entries as such:



We now have a customized list that's adapted to its renderer:



Understanding MVC

Lets recap, what is MVC:

Model - Represents the data for the component (list), the model can tell us exactly how many items are in it and which item resides at a given offset within the model. This differs from a simple Vector (or array) since all access to the model is controlled (the interface is simpler) and unlike a Vector/Array the model can notify us of changes that occur within it.

View - The view draws the content of the model. It is a "dumb" layer that has no notion of what is displayed and only knows how to draw. It tracks changes in the model (the model sends events) and redraws itself when it changes.

Controller - The controller accepts user input and performs changes to model which in turn cause the view to refresh.

Codename One's List component uses the MVC paradigm to separate its implementation. List itself is the Controller (with a bit of View mixed in). The ListCellRenderer interface is a View and the ListModel is (you guessed it by now) the model.

When the list is painted it iterates over the visible elements in the model and asks for them, it then draws them using the renderer.

Why is this useful?

Since the model is a lightweight interface it can be implemented by you and replaced in runtime if so desired, this allows several very cool use cases:

1. A list can contain thousands of entries but only load the portion visible to the user. Since the model will only be queried for the elements that are visible to the user it won't need to load into

memory a very large data set until the user starts scrolling down (at which point other elements may be offloaded from memory).

2. A list can cache efficiently. E.g. a list can mirror data from the server into local RAM without actually downloading all the data. Data can also be mirrored from RMS for better performance and discarded for better memory utilization.

3. No need for state copying. Since renderers allow us to display any object type, the list model interface can be implemented by the applications data structures (e.g. persistence/network engine) which would return internal application data structures saving you the need of copying application state into a list specific data structure.

4. Using the proxy pattern (as explained in a previous post) we can layer logic such as filtering, sorting, caching etc. on top of existing models without changing the model source code.

5. We can reuse generic models for several views e.g. a model that fetches data from the server can be initialized with different arguments to fetch different data for different views. View objects in different Form's can display the same model instance in different view instances thus they would update automatically when we change one global model.

Most of these use cases work best for lists that grow to a larger size or represent complex data which is what the list object is designed to do.

List Cell Renderer

List is one of the most important widgets in Codename One, unfortunately it is also one of the most difficult widgets to understand.

The List component uses the MVC model inspired from Swing (which was inspired from SmallTalk), we created a data model to encapsulate the data and a renderer to display the data items on the screen.

Let's have a closer look at the List Renderer, the Renderer is a simple interface with 2 methods:

```
public interface ListCellRenderer {
//This method is called by the List for each item, when the List paints itself.
public Component getListCellRendererComponent(List list, Object value, int index, boolean isSelected);

//This method returns the List animated focus which is animated when list selection changes
public Component getListFocusComponent(List list);
```

```
}
```

Let's try to implement our own renderer.
The most simple/naive implementation may choose to implement the renderer as follows:

```
public Component getListCellRendererComponent(List list, Object value, int index, boolean
isSelected){
return new Label(value.toString());
}

public Component getListFocusComponent(List list){
return null;
}
```

This will compile and work, but won't give you much, notice that you won't see the List selection move on the List, this is just because the renderer returns a Label with the same style regardless if it's being selected or not.

Now Let's try to make it a bit more useful.

```
public Component getListCellRendererComponent(List list, Object value, int index, boolean
isSelected){
    Label l = new Label(value.toString());
if (isSelected) {
        l.setFocus(true);
        l.getStyle().setBgTransparency(100);
     } else {
        l.setFocus(false);
        l.getStyle().setBgTransparency(0);
    }
    return l;
}  public Component getListFocusComponent(List list){
    return null;
}
```

In this renderer we set the Label.setFocus(true) if it's selected, calling to this method doesn't really gives the focus to the Label, it simply indicates to the LookAndFeel to draw the Label with fgSelectionColor and bgSelectionColor instead of fgColor and bgColor.

Then we call to Label.getStyle().setBgTransparency(100) to give the selection semi transparency and 0 for full transparency if not selected.
OK that's a bit more functional, but not very efficient that's because we create a new Label each time the method is called.

To make it more device friendly keep a reference to the Component or extend the Widget.

```
class MyRenderer extends Label implements ListCellRenderer {

public Component getListCellRendererComponent(List list, Object value, int index, boolean
isSelected){
setText(value.toString());
if (isSelected) {
  setFocus(true);
  getStyle().setBgTransparency(100);
} else {
setFocus(false);
  getStyle().setBgTransparency(0);
}
return this;
}
}
}
```

Now Let's have a look at a more advanced Renderer

```
class ContactsRenderer extends Container implements ListCellRenderer {

 private Label name = new Label("");
 private Label email = new Label("");
 private Label pic = new Label("");

 private Label focus = new Label("");

 public ContactsRenderer() {
    setLayout(new BorderLayout());
    addComponent(BorderLayout.WEST, pic);
    Container cnt = new Container(new BoxLayout(BoxLayout.Y_AXIS));
    name.getStyle().setBgTransparency(0);
    name.getStyle().setFont(Font.createSystemFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
Font.SIZE_MEDIUM));
```

```
    email.getStyle().setBgTransparency(0);
    cnt.addComponent(name);
    cnt.addComponent(email);
    addComponent(BorderLayout.CENTER, cnt);

    focus.getStyle().setBgTransparency(100);
}

 public Component getListCellRendererComponent(List list, Object value, int index, boolean
isSelected) {

    Contact person = (Contact) value;
    name.setText(person.getName());
    email.setText(person.getEmail());
    pic.setIcon(person.getPic());
    return this;
}

 public Component getListFocusComponent(List list) {
    return focus;
 }
}
```

In this renderer we want to renderer a Contact Object to the Screen, we build the Component in the constructor and in the getListCellRendererComponent we simply updates the Labels texts according to the Contact Object.
Notice that in this renderer I return a focus Label with semi transparency, as mentioned before the focus component can be modified by using this method.
For example I can modify the focus Component to have an icon.

```
   focus.getStyle().setBgTransparency(100);
try {
  focus.setIcon(Image.createImage("/duke.png"));
  focus.setAlignment(Component.RIGHT);
} catch (IOException ex) {
    ex.printStackTrace();
}
```

Generic List Cell Renderer

Codename One is really powerful and flexible, we took the power and flexibility of Swing and went even further (styles, painters) and one such power is the cell renderer. This is a concept we derived from Swing which is both remarkably powerful and pretty hard for newbies to figure out, frankly it's pretty hard for everyone...

As part of the GUI builder work we needed a way to customize rendering for a List but the renderer/model approach seemed impossible to adapt to a GUI builder (it seems the Swing GUI builders had a similar issue). Our solution was to introduce the GenericListCellRenderer which while introducing limitations and implementation requirements still manages to make life easier both in the GUI builder and outside of it.

GenericListCellRenderer is a renderer designed to be as simple to use as a Component-Container hierarchy, we effectively crammed most of the common renderer use cases into one class. To enable that we need to know the content of the objects within the model, so the GenericListCellRenderer assumes the model contains only Hashtable objects. Since Hashtable's can contain arbitrary data the list model is still quite generic and allows storing application specific data, furthermore a Hashtable can still be derived and extended to provide domain specific business logic.
The GenericListCellRenderer accepts two container instances (more later on why at least two and not one) which it maps to individual Hashtable entries within the model by finding the appropriate components within the given container hierarchy. Components are mapped to the Hashtable entries based on the name property of the component (get/setName) and the key value within the Hashtable e.g.:
For a model that contains a Hashtable entry like this:
"Foo": "Bar"
"X": "Y"
"Not": "Applicable"
"Number": Integer(1)

A renderer will loop over the component hierarchy in the container searching for component's whose name matches Foo, X, Not and Number and assign to them the appropriate value. Notice that you can also use image objects as values and they will be assigned to labels as expected. However, you can't assign both an image and a text to a single label since the key will be taken. That isn't a big problem since two labels can be used quite easily in such a renderer.
To make matters even more attractive the renderer seamlessly supports list tickering when appropriate and if a CheckBox appears within the renderer it will toggle a boolean flag within the Hashtable seamlessly.
One issue that crops up with this approach is that if a value is missing from the hashtable it is treated as empty and the component is reset, this can pose an issue if we hardcode an image

or text within the renderer and we don't want them replace (e.g. an arrow graphic). The solution for this is to name the component with Fixed in the end of the name e.g.: HardcodedIconFixed. Naming a component within the renderer with $number will automatically set it as a counter component for the offset of the component within the list.

Styling the GenericListCellRenderer is slightly different, the renderer uses the UIID of the container passed to the generic list cell renderer and the background focus uses that same UIID with the word "Focus" appended.
It is important to notice that the generic list cell renderer will grant focus to the child components of the selected entry if they are focusable thus changing the style of said entries. E.g. a Container might have a child label that has one style when the parent container is unselected and another when its selected (focused), this can be easily achieved by defining the label as focusable. Notice that the component will never receive direct focus since it is still a par of a renderer.

Last but not least, the generic list cell renderer accepts two or four instances of a Container rather than the obvious choice of accepting only one instance. This allows the renderer to treat the selected entry differently which is especially important to tickering although its also useful for fisheye. Since it might not be practical to seamlessly clone the Container for the renderer's needs Codename One expects the developer to provide two separate instances, they can be identical in all respects but they must be separate instances for tickering to work. The renderer also allows for a fisheye effect where the selected entry is actually different from the unselected entry in its structure, it also allows for a pinstripe effect where odd/even rows have different styles (this is accomplished by providing 4 instances of the containers selected/unselected for odd/even).

The best way to learn about the generic list cell renderer and the hashtable model is by playing with them in the GUI builder, however they can be used in code without any dependency on the GUI builder and can be quite useful at that.

Here is a simple sample for a list with checkboxes that get updated automatically:

```
List list = new List(createGenericListCellRendererModelData());
list.setRenderer(new GenericListCellRenderer(createGenericRendererContainer(),
createGenericRendererContainer()));


private Container createGenericRendererContainer() {
    Container c = new Container(new BorderLayout());
    c.setUIID("ListRenderer");
    Label name = new Label();
```

```
    name.setFocusable(true);
    name.setName("Name");
    c.addComponent(BorderLayout.CENTER, name);
    Label surname = new Label();
    surname.setFocusable(true);
    surname.setName("Surname");
    c.addComponent(BorderLayout.SOUTH, surname);
    CheckBox selected = new CheckBox();
    selected.setName("Selected");
    selected.setFocusable(true);
    c.addComponent(BorderLayout.WEST, selected);
    return c;
}

private Hashtable[] createGenericListCellRendererModelData() {
    Hashtable[] data = new Hashtable[5];
    data[0] = new Hashtable();
    data[0].put("Name", "Shai");
    data[0].put("Surname", "Almog");
    data[0].put("Selected", Boolean.TRUE);
    data[1] = new Hashtable();
    data[1].put("Name", "Chen");
    data[1].put("Surname", "Fishbein");
    data[1].put("Selected", Boolean.TRUE);
    data[2] = new Hashtable();
    data[2].put("Name", "Ofir");
    data[2].put("Surname", "Leitner");
    data[3] = new Hashtable();
    data[3].put("Name", "Yaniv");
    data[3].put("Surname", "Vakarat");
    data[4] = new Hashtable();
    data[4].put("Name", "Meirav");
    data[4].put("Surname", "Nachmanovitch");
    return data;
}
```

The List Model

Swing's approach to MVC is one of the hardest concepts for people to fully grasp, which is a real shame as it is probably the most important and powerful feature in Swing. Codename One copied Swing's approach to MVC almost entirely but on a smaller scale.

To show off the power of the list model we create a list with one million entries... What I am trying to prove here is that a list and a model have a very low overhead when used properly. Most of the overhead for rendering a list is in the renderer and the model implementation, both of which you can optimize to your hearts content. This is a very small price to pay for something as flexible, powerful and customizable as the Codename One list!

```
class Contact {
 private String name;
 private String email;
 private Image pic;

 public Contact(String name, String email, Image pic) {
    this.name = name;
    this.email = email;
    this.pic = pic;
 }

 public String getName() {
    return name;
 }

 public String getEmail() {
    return email;
 }

 public Image getPic() {
    return pic;
 }
}

class ContactsRenderer extends Container implements ListCellRenderer {

 private Label name = new Label("");
 private Label email = new Label("");
 private Label pic = new Label("");

 private Label focus = new Label("");

 public ContactsRenderer() {
    setLayout(new BorderLayout());
    addComponent(BorderLayout.WEST, pic);
```

```
    Container cnt = new Container(new BoxLayout(BoxLayout.Y_AXIS));
    name.getStyle().setBgTransparency(0);
    email.getStyle().setBgTransparency(0);
    cnt.addComponent(name);
    cnt.addComponent(email);
    addComponent(BorderLayout.CENTER, cnt);
 }

 public Component getListCellRendererComponent(List list, Object value, int index, boolean
isSelected) {
    Contact person = (Contact) value;
    name.setText(index + ": " + person.getName());
    email.setText(person.getEmail());
    pic.setIcon(person.getPic());
    return this;
 }

 public Component getListFocusComponent(List list) {
    return focus;
 }
}
String[][] CONTACTS_INFO = {
 {"Nir V.","Nir.V@Sun.COM"},
 {"Tidhar G.","Tidhar.G@Sun.COM"},
 {"Iddo A.","Iddo.A@Sun.COM"},
 {"Ari S.","Ari.S@Sun.COM"},
 {"Chen F.","Chen.F@Sun.COM"},
 {"Yoav B.","Yoav.B@Sun.COM"},
 {"Moshe S.","Moshe.S@Sun.COM"},
 {"Keren S.","Keren.S@Sun.COM"},
 {"Amit H.","Amit.H@Sun.COM"},
 {"Arkady N.","Arcadi.N@Sun.COM"},
 {"Shai A.","Shai.A@Sun.COM"},
 {"Elina K.","Elina.K@Sun.COM"},
 {"Yaniv V.","Yaniv.V@Sun.COM"},
 {"Nadav B.","Nadav.B@Sun.COM"},
 {"Martin L.","Martin.L@Sun.COM"},
 {"Tamir S.","Tamir.S@Sun.COM"},
 {"Nir S.","Nir.S@Sun.COM"},
 {"Eran K.","Eran.K@Sun.COM"}
 };
```

```
int contactWidth= 36;
int contactHeight= 48;
int cols = 4;
Resources images = Resources.open("/images.res");
Image contacts = images.getImage("people.jpg");
Image[] persons = new Image[CONTACTS_INFO.length];
for(int i = 0; i < persons.length ; i++){
 persons[i] = contacts.subImage((i%cols)*contactWidth, (i/cols)*contactHeight, contactWidth,
contactHeight, true);
}

final Contact[] contactArray = new Contact[persons.length];
for (int i = 0; i < contactArray.length; i++) {
 int pos = i % CONTACTS_INFO.length;
 contactArray[i] = new Contact(CONTACTS_INFO[pos][0], CONTACTS_INFO[pos][1],
persons[pos]);
}

Form millionList = new Form("Million");
millionList.setScrollable(false);
List l = new List(new ListModel() {
 private int selection;
 public Object getItemAt(int index) {
    return contactArray[index % contactArray.length];
 }

 public int getSize() {
    return 1000000;
 }

 public int getSelectedIndex() {
    return selection;
 }

 public void setSelectedIndex(int index) {
    selection = index;
 }

 public void addDataChangedListener(DataChangedListener l) {
 }
```

```
    public void removeDataChangedListener(DataChangedListener l) {
    }

    public void addSelectionListener(SelectionListener l) {
    }

    public void removeSelectionListener(SelectionListener l) {
    }

    public void addItem(Object item) {
    }

    public void removeItem(int index) {
    }
});
l.setListCellRenderer(new ContactsRenderer());
l.setFixedSelection(List.FIXED_NONE_CYCLIC);
millionList.setLayout(new BorderLayout());
millionList.addComponent(BorderLayout.CENTER, l);
millionList.show();
```

MultiList

The MultiList is a preconfigured list that contains a ready made renderer with defaults that make sense for the most common use cases. It still retains most of the power available to the list component but reduces the complexity of one of the hardest things to grasp for most developers: rendering.

It still has the full power of the model and allows you to create a million entry list with just a few lines of code, however the objects the model returns should always be in the form of Hashtables and not any arbitrary object like the standard list allows.

You can create a MutiList by just dropping it into place in the GUI builder and just editing the list data property (see the instructions above for creating a list in the GUI builder, you won't need the renderer portion).

Slider

A slider is an empty component that can be filled horizontally or vertically to allow indicating progress/volume etc. It can be editable to allow the user to determine its value or none editable to just relay that information to the user.

It can have a thumb on top to show its current position.

The interesting part about the slider is that it has two separate style UIID's, Slider & SliderFull. The Slider UIID is always painted and SliderFull is rendered on top based on the amount the slider should be filled.

Table

Unlike list the table is a composite component, which means it is really a subclass of a Container and is effectively built from multiple components. The general thought process is that Table is an elaborate component and should include complex editing, while the list is more of a selection component designed for scalability.

Here is a minor sample of using the standard table component, it should be pretty self explanatory:

```
final Form f = new Form("Table Test");
TableModel model = new DefaultTableModel(new String[] {"Col 1", "Col 2", "Col 3"}, new
Object[][] {
{"Row 1", "Row A", "Row X"},
{"Row 2", "Row B", "Row Y"},
{"Row 3", "Row C", "Row Z"},
{"Row 4", "Row D", "Row K"},
}) {
public boolean isCellEditable(int row, int col) {
   return col != 0;
}
};
Table table = new Table(model);
table.setScrollableX(true);
f.setLayout(new BorderLayout());
f.addComponent(BorderLayout.CENTER, table);
f.show();
```

However, the more "interesting" aspect of the table is the table layout and its ability to create rather unique layouts relatively easily similarly to HTML's tables. You can use the layout constraints (also exposed in the table class) to create spanning and elaborate UI's.
In order to customize the table cell behavior you can now derive the table to create a "renderer like" widget, however unlike the list this component is "kept" and used as is. This means you

can bind listeners to this component and work with it as you would with any other component in Codename One.

Tree

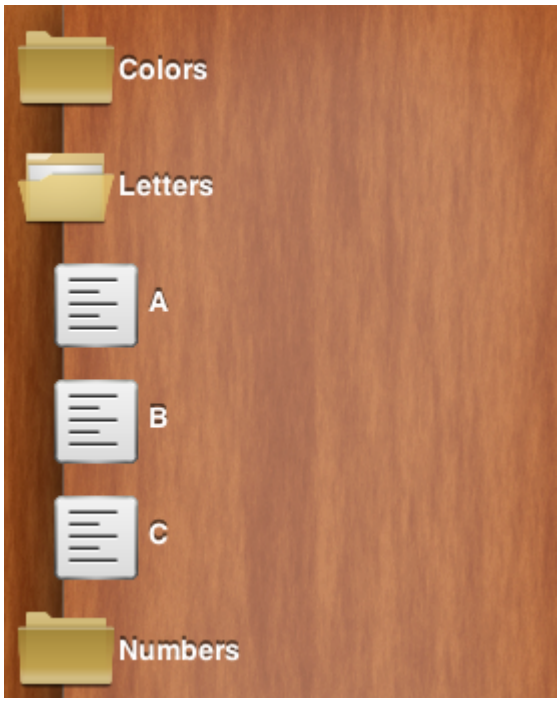
A tree allows displaying hierarchical data such as folders and files in a collapsible/expandable UI. Like the Table it is a Container derived component that works against a model to construct its user interface on the fly.
In order for the tree to have content you need to create a tree model e.g. this:

```
class StringArrayTreeModel implements TreeModel {
    String[][] arr = new String[][] {
            {"Colors", "Letters", "Numbers"},
            {"Red", "Green", "Blue"},
            {"A", "B", "C"},
            {"1", "2", "3"}
        };

    public Vector getChildren(Object parent) {
        if(parent == null) {
            Vector v = new Vector();
            for(int iter = 0 ; iter < arr[0].length ; iter++) {
                v.addElement(arr[0][iter]);
            }
            return v;
        }
        Vector v = new Vector();
        for(int iter = 0 ; iter < arr[0].length ; iter++) {
            if(parent == arr[0][iter]) {
                if(arr.length > iter + 1 && arr[iter + 1] != null) {
                    for(int i = 0 ; i < arr[iter + 1].length ; i++) {
                        v.addElement(arr[iter + 1][i]);
                    }
                }
            }
        }
        return v;
    }

    public boolean isLeaf(Object node) {
        Vector v = getChildren(node);
        return v == null || v.size() == 0;
```

```
    }
}
```

```
Tree dt = new Tree(new StringArrayTreeModel());
```

Will result in this:



Share Button

The share button allows you to help your users market your application by sharing it with their friends easily. When they press the button they will be faced with either the native sharing capability of the platform (as is available on Android) or a builtin set of sharing features such as facebook, email, SMS etc. You can customize the button to add additional sharing options.

Infinite Progress

The infinite progress indicator spins an image infinitely, it is automatically associated with a default image for infinite progress from the native theme. This can be used in several ways e.g. you can place this component into a layout to indicate work underway.
A useful function in the infinite progress component is the showInifiniteBlocking() method which modelessly shows a translucent dialog with the infinite progress indicator in it.

Tabs

The Tabs component allows arranging components into groups within containers, its a container type that allows leafing through its children using labeled buttons. The tabs can be placed in multiple different ways (top, bottom, left or right) with the default being determined by the platform. This class also allows swiping between components to leaf between said tabs (for this purpose the tabs themselves can also be hidden).

MediaPlayer

The media player allows you to control video playback e.g. to show a video one can simply use something like this:

```
final MediaPlayer mp = new MediaPlayer();
try {
    mp.setDataSource(myMediaFile);
} catch (IOException ex) {
    ex.printStackTrace();
}
player.addComponent(BorderLayout.CENTER, mp);
```

In order to run the media/video in the simulator you will need to run a version of Java 7 update 6 or newer. We rely on features that were integrated into that version in order to provide proper video codec support.

WebBrowser

The web browser component shows the native device web browser when supported by the device and the HTMLComponent when the web browser isn't supported on the given device.

To create a simple web browser component we can do something like this (assuming page.html is present in the jar):

```
WebBrowser wb = new WebBrowser();
wb.setURL("jar:///Page.html");
```

However, on devices where more elaborate HTML rendering exists we can also do things such as communicate with the HTML code using JavaScript calls (notice that opening an alert in an embedded native browser might not work). E.g. we can create HTML like this:

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Test</title>
    <script>
     function  fnc(message) {
     document.write(message);
       };
    </script>
  </head>
  <body >
    <p>Demo</p>
  </body>
</html>
```

And then communicate with the function from code like this:

```
WebBrowser web = new WebBrowser(){
  @Override
  public void onLoad(String url) {
    Component c = getInternal();
    if(c instanceof BrowserComponent) {
      BrowserComponent b = (BrowserComponent)c;
      b.execute("fnc('<p>Hello World</p>')");
    }
  }
};
f.addComponent(BorderLayout.CENTER, web);
web.setURL("jar:///page.html");
```

In order to see the native web browser in the simulator you will need to run using Java 7 update 6 or newer.

Embedded Container

EmbeddedContainer solves a problem that exists only within the GUI builder and the class makes no sense outside of the context of the GUI builder.
The necessity for EmbeddedContainer came about due to iPhone inspired designs that relied on tabs (iPhone style tabs at the bottom of the screen) where different features of the application are within a different tab.

This didn't mesh well with the GUI builder navigation logic and so we needed to rethink some of it. We wanted to reuse GUI as much as possible while still enjoying the advantage of navigation being completely managed for me.
Android does this with Activities and the iPhone itself has a view controller, we don't like both approaches and think they both suck. The problem is that you have what is effectively two incompatible hierarchies to mix and match which is why Android needed to "invent" fragments and Apple can't mix view controllers within a single application.

The Component/Container heirarchy is powerful enough to represent such a UI but we needed a "marker" to indicate to the UIBuilder where a "root" component exists so navigation occurs only within the given "root". Here EmbeddedContainer comes into play, its a simple container that can only contain another GUI from the GUI builder. Nothing else. So we can place it in any form of UI and effectively have the UI change appropriately and navigation would default to "sensible values".

Navigation replaces the content of the embedded container, it finds the embedded container based on the component that broadcast the event. If you want to navigate manually just use the showContainer() method which accepts a component, you can give any component that is under the EmbeddedContainer you want to replace and Codename One will be smart enough to replace only that component.

The nice part about using the EmbeddedContainer is that the resulting UI can be very easily refactored to provide a more traditional form based UI without duplicating effort and can be easily adapted to a more tablet oriented UI (with a side bar) again without much effort.

The Map Component

The MapComponent uses the OpenStreetMap webservice by default to display a navigatable map.
The code was contributed by Roman Kamyk and was originally used for a LWUIT application.

The screenshot above was produced using the following code:

```
Form map = new Form("Map");
 map.setLayout(new BorderLayout());
 map.setScrollable(false);
 final MapComponent mc = new MapComponent();

 try {
    //get the current location from the Location API
    Location loc = LocationManager.getLocationManager().getCurrentLocation();

   Coord lastLocation = new Coord(loc.getLatitude(), loc.getLongtitude());
    Image i = Image.createImage("/blue_pin.png");
    PointsLayer pl = new PointsLayer();
    pl.setPointIcon(i);
    PointLayer p = new PointLayer(lastLocation, "You Are Here", i);
    p.setDisplayName(true);
    pl.addPoint(p);
```

```
        mc.addLayer(pl);
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    mc.zoomToLayers();

    map.addComponent(BorderLayout.CENTER, mc);
    map.addCommand(new BackCommand());
    map.setBackCommand(new BackCommand());
    map.show();
```

The example below shows how to integrate the MapComponent with the Google Location API. make sure to obtain your secret api key from the Google Location data API at: https://developers.google.com/maps/documentation/places/


```
final Form map = new Form("Map");
map.setLayout(new BorderLayout());
map.setScrollable(false);
```

```
final MapComponent mc = new MapComponent();
Location loc = LocationManager.getLocationManager().getCurrentLocation();
//use the code from above to show you on the map
putMeOnMap(mc);
map.addComponent(BorderLayout.CENTER, mc);
map.addCommand(new BackCommand());
map.setBackCommand(new BackCommand());

ConnectionRequest req = new ConnectionRequest() {

   protected void readResponse(InputStream input) throws IOException {
      JSONParser p = new JSONParser();
      Hashtable h = p.parse(new InputStreamReader(input));
      // "status" : "REQUEST_DENIED"
      String response = (String)h.get("status");
      if(response.equals("REQUEST_DENIED")){
         System.out.println("make sure to obtain a key from "
               + "https://developers.google.com/maps/documentation/places/");
         progress.dispose();
         Dialog.show("Info", "make sure to obtain an application key from "
               + "google places api's"
               , "Ok", null);
         return;
      }

      final Vector v = (Vector) h.get("results");

      Image im = Image.createImage("/red_pin.png");
      PointsLayer pl = new PointsLayer();
      pl.setPointIcon(im);
      pl.addActionListener(new ActionListener() {

         public void actionPerformed(ActionEvent evt) {
            PointLayer p = (PointLayer) evt.getSource();
            System.out.println("pressed " + p);

            Dialog.show("Details", "" + p.getName(), "Ok", null);
         }
      });

      for (int i = 0; i < v.size(); i++) {
```

```
                Hashtable entry = (Hashtable) v.elementAt(i);
                Hashtable geo = (Hashtable) entry.get("geometry");
                Hashtable loc = (Hashtable) geo.get("location");
                Double lat = (Double) loc.get("lat");
                Double lng = (Double) loc.get("lng");
                PointLayer point = new PointLayer(new Coord(lat.doubleValue(),
lng.doubleValue()),
                        (String) entry.get("name"), null);
                pl.addPoint(point);
            }
            progress.dispose();

            mc.addLayer(pl);
            map.show();
            mc.zoomToLayers();

        }
      };
      req.setUrl("https://maps.googleapis.com/maps/api/place/search/json");
      req.setPost(false);
      req.addArgument("location", "" + loc.getLatitude() + "," + loc.getLongtitude());
      req.addArgument("radius", "500");
      req.addArgument("types", "food");
      req.addArgument("sensor", "false");

      //get your own key from https://developers.google.com/maps/documentation/places/
      //and replace it here.
      String key = "yourAPIKey";

      req.addArgument("key", key);

      NetworkManager.getInstance().addToQueue(req);
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
  }
```

Animations & Transitions

There are many ways to animate and liven the data within a Codename One application, one which we already discussed is the layout animations mechanism however there is a great more.

Low Level Animations

To understand the flow of animations in Codename One we can start by discussing the underlying low level animations and the motivations behind them. The Codename One event dispatch thread has a special animation “pulse” allowing an animation to update its state and draw itself. Code can make use of this pulse to implement repetitive polling tasks that have very little to do with drawing.
This is helpful since the callback will always occur on the event dispatch thread.

Every component in Codename One contains an animate() method that returns a boolean value, you can also implement the Animation interface in an arbitrary component to implement your own animation. In order to receive animation events you need to register yourself within the parent form, it is the responsibility of the parent for to call animate().
If the animate method returns true then the animation will be painted. It is important to deregister animations when they aren’t needed to conserve battery life. However, if you derive from a component which has its own animation logic you might damage its animation behavior by deregistering it, so tread gently with the low level API’s.

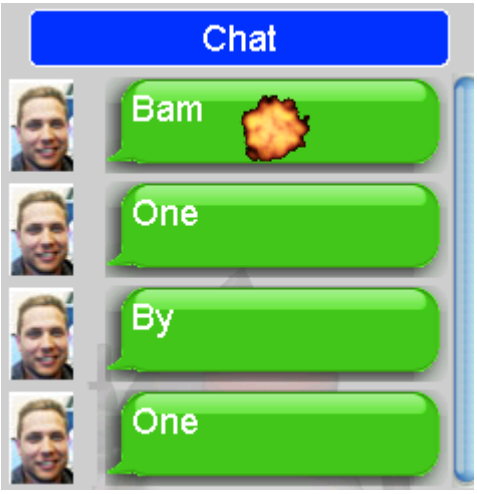
```
myForm.registerAnimated(this);

private int spinValue;
public boolean animate() {
  if(userStatusPending) {
     spinValue++;
     super.animate();
     return true;
  }
  return super.animate();
}
```

Transitions

Transitions allow us to replace one component with another, most typically forms or dialogs are replaced with a transition however a transition can be applied to replace any arbitrary component.
Developers can implement their own custom transition and install it to components by deriving the transition class, although most commonly the built in CommonTransition class is used for almost everything.
You can define transitions for forms/dialogs/menus globally either via the theme constant or via the LookAndFeel class. Alternatively you can install a transition on top level components via setter methods.
To apply a transition to a component we can just use the Container.replace method as such:

```
Container c = replace.getParent();
ta.setPreferredSize(replace.getPreferredSize());
c.replaceAndWait(replace, ta,
CommonTransitions.createSlide(CommonTransitions.SLIDE_VERTICAL, true, 500));
c.replaceAndWait(ta, replace,
CommonTransitions.createSlide(CommonTransitions.SLIDE_VERTICAL, false, 500));
```

In addition we can implement our own transitions, e.g. the following code demonstrates an explosion transition in which an explosion animation is displayed on every component as the explode one by one while we move from one screen to the next.




```
public class ExplosionTransition extends Transition {
    private int duration;
    private Image[] explosions;
    private Motion anim;
    private Class[] classes;
    private boolean done;
    private int[] locationX;
    private int[] locationY;
```

```
    private Vector components;
    private Vector sequence;
    private boolean sequential;
    private int seqLocation;
    public ExplosionTransition(int duration, Class[] classes, boolean sequential) {
        this.duration = duration;
        this.classes = classes;
        this.sequential  = sequential;
    }

    public void initTransition() {
        try {
            explosions = new Image[] {
                Image.createImage("/explosion1.png"),
                Image.createImage("/explosion2.png"),
                Image.createImage("/explosion3.png"),
                Image.createImage("/explosion4.png"),
                Image.createImage("/explosion5.png"),
                Image.createImage("/explosion6.png"),
                Image.createImage("/explosion7.png"),
                Image.createImage("/explosion8.png")
            };
            done = false;
            Container c = (Container)getSource();
            components = new Vector();
            addComponentsOfClasses(c, components);
            if(components.size() == 0) {
                return;
            }
            locationX = new int[components.size()];
            locationY = new int[components.size()];
            int w = explosions[0].getWidth();
            int h = explosions[0].getHeight();
            for(int iter = 0 ; iter < locationX.length ; iter++) {
                Component current = (Component)components.elementAt(iter);
                locationX[iter] = current.getAbsoluteX() + current.getWidth() / 2 - w / 2;
                locationY[iter] = current.getAbsoluteY() + current.getHeight() / 2 - h / 2;
            }
            if(sequential) {
                anim = Motion.createSplineMotion(0, explosions.length - 1, duration /
locationX.length);
```

```
            sequence = new Vector();
        } else {
            anim = Motion.createSplineMotion(0, explosions.length - 1, duration);
        }
        anim.start();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

private void addComponentsOfClasses(Container c, Vector result) {
    for(int iter = 0 ; iter < c.getComponentCount() ; iter++) {
        Component current = c.getComponentAt(iter);
        if(current instanceof Container) {
            addComponentsOfClasses((Container)current, result);
        }
        for(int ci = 0 ; ci < classes.length ; ci++) {
            if(current.getClass() == classes[ci]) {
                result.addElement(current);
                break;
            }
        }
    }
}

public void cleanup() {
    super.cleanup();
    explosions = null;
    if(sequential) {
        components = sequence;
    }
    if(components != null) {
        for(int iter = 0 ; iter < components.size() ; iter++) {
            ((Component)components.elementAt(iter)).setVisible(true);
        }
        components.removeAllElements();
    }
}

public Transition copy() {
    return new ExplosionTransition(duration, classes, sequential);
```

```
}

public boolean animate() {
    if(sequential) {
        if(anim != null && anim.isFinished() && components.size() > 0) {
            Component c = (Component)components.elementAt(0);
            components.removeElementAt(0);
            sequence.addElement(c);
            c.setVisible(false);
            if(components.size() > 0) {
                seqLocation++;
                anim.start();
            }

            return true;
        }
        return components.size() > 0;
    }
    if(anim != null && anim.isFinished() && !done) {
        // allows us to animate the last frame, we should animate once more when
        // finished == true
        done = true;
        return true;
    }
    return !done;
}

public void paint(Graphics g) {
    getSource().paintComponent(g);
    int offset = anim.getValue();
    if(sequential) {
        g.drawImage(explosions[offset], locationX[seqLocation], locationY[seqLocation]);
        return;
    }
    for(int iter = 0 ; iter < locationX.length ; iter++) {
        g.drawImage(explosions[offset], locationX[iter], locationY[iter]);
    }
    if(offset > 4) {
        for(int iter = 0 ; iter < components.size() ; iter++) {
            ((Component)components.elementAt(iter)).setVisible(false);
        }
```

```
        }
    }
}
```

The EDT - Event Dispatch Thread

This chapter covers the basic concepts of the EDT

What Is The EDT

Codename One allows developers to create as many threads as they want, however in order to interact with the Codename One user interface components a developer must use the EDT. The EDT is the main thread of Codename One, by using just one thread Codename One can avoid complex synchronization code and focus on simple functionality that assumes only one thread. This has huge advantages in your code as well, you can normally assume that all code will occur on a single thread, however this also comes with a price...

Normally, every call you receive from Codename One will occur on the EDT. E.g. every event, calls to paint(), lifecycle calls (start etc.) should all occur on the EDT. This is pretty powerful, however it means that as long as your code is processing nothing else can happen in Codename One... If your code takes too long to execute then no painting or event processing will occur during that time, so a call to Thread.sleep() will actually stop everything!

The solution is pretty simple, if you need to perform something that requires intensive CPU you can spawn a thread, Codename One's networking code automatically spawns a separate thread (see that NetworkManager chapter for more). However, we now run into a problem... Codename One assumes all modifications to the UI are performed on the EDT but we just spawned a separate thread. How do we force our modifications back into the EDT?

Codename One includes 3 methods in the Display class to help in these situations: isEDT(), callSerially(Runnable) & callSeriallyAndWait(Runnable).

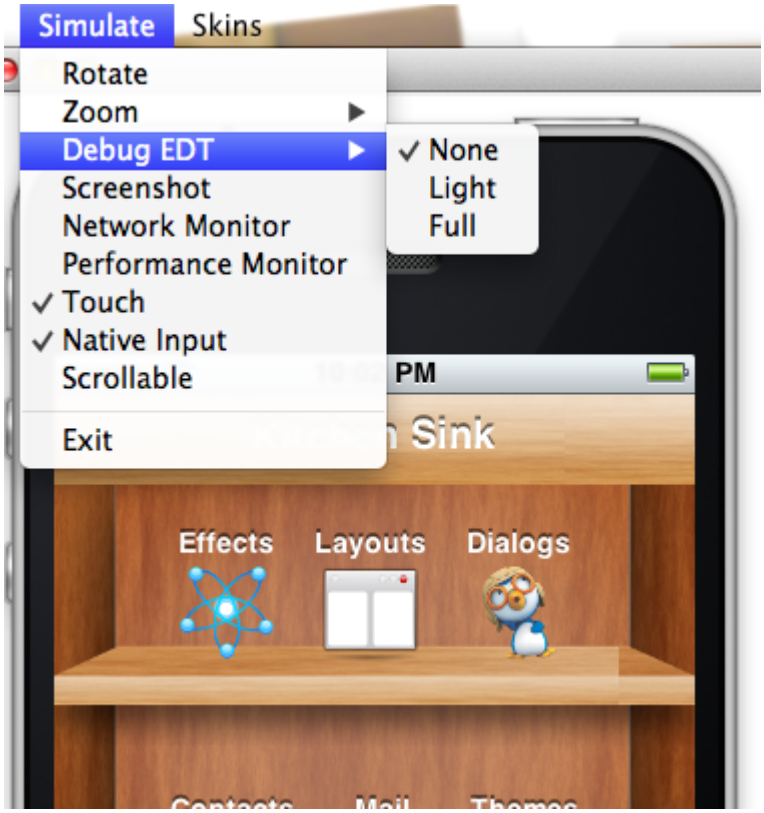
isEDT() is useful for generic code that needs to test whether the current code is executing on the EDT.

Debugging EDT Violations

There are two types of EDT violations:

1. Blocking the EDT thread so the UI performance is considerably slower.
2. Invoking UI code on a separate thread

Codename One provides a tool to help you detect some of these violations with some caveats, it is an imperfect tool. It might fire “false positives” meaning it might detect a violation for perfectly legal code and it might miss some illegal calls.
However, it is a valuable tool in the process of detecting hard to track bugs which are sometimes only reproducible on the devices (due to race condition behavior).
To activate this tool just select the Debug EDT menu option in the simulator and pick the level of output you wish to receive:



Full output will include stack traces to the area in the code that is suspected in the violation.

Call Serially (And Wait)

callSerially(Runnable) should normally be called off the EDT (in a separate thread), the run method within the submitted runnable will be invoked on the EDT. E.g.:

```
// this code is executing in a separate thread
final String res = methodThatTakesALongTime();
Display.getInstance().callSerially(new Runnable() {
    public void run() {
```

```
        // this occurs on the EDT so I can make changes to UI components
        resultLabel.setText(res);
    }
});
```

This allows code to leave the EDT and then later on return to it to perform things within the EDT.

The callSeriallyAndWait(Runnable) method blocks the current thread until the method completes, this is useful for cases such as user notification e.g.:

```
// this code is executing in a separate thread
methodThatTakesALongTime();
Display.getInstance().callSeriallyAndWait(new Runnable() {
    public void run() {
        // this occurs on the EDT so I can make changes to UI components
        globalFlag = Dialog.show("Are You Sure?", "Do you want to continue?", "Continue",
"Stop");
    }
});
// this code is executing the separate thread
// global flag was already set by the call above
if(!globalFlag) {
  return;
}
otherMethod();
```

It sometimes makes sense to invoke callSerially (but not call serially and wait) on the EDT. We sometimes want to postpone an action to the next cycle of the EDT loop, but that is a rare occurrence.

Invoke And Block

Invoke and block is the exact opposite of callSeriallyAndWait(), it blocks the EDT and opens a separate thread for the runnable call.
Codename One has some nifty threading tools inspired by Foxtrot, which is a remarkably powerful tool most Swing developers don't know enough about.

When people talk about dialog modality they often mean two separate things, the first indicates that the dialog intercepts all input and blocks the background form/window which is the true definition of modality. However, there is another aspect often associated with modality that is really important from a programmers perspective and simplified our code considerably:

```
public void actionPerformed(ActionEvent ev) {
  // will return true if the user clicks "OK"
  if(!Dialog.show("Question", "How Are You", "OK", "Not OK")) {
  // ask what went wrong...
  }
}
```

Notice that the dialog show method will block the calling thread until the user clicks OK or Not OK...
If you read a bit about Codename One you would notice that we are blocking the EDT (Event Dispatch Thread) which is also responsible for painting, how does the dialog paint itself or handle events?

The secret is invokeAndBlock, it allows us to "block" the EDT and resume it while keeping a "nested" EDT functioning. The semantics of this logic are a bit hairy so I won't try to explain them further, this functionality is also available in Swing which has the exact same modality feature however Swing doesn't expose the "engine" to developers. Foxtrot, exposes this undocumented engine to Swing developers, in Codename One we chose to expose the ability to block the EDT (without "really" blocking it) as a simple API: invokeAndBlock.

The best way to explain this is by example:

```
public void actionPerformed(ActionEvent ev) {
  label.setText("Initiating IO, please wait...");
  Display.getInstance().invokeAndBlock(new Runnable() {
      public void run() {
          // perform IO operation...
      }
  });
  label.setText("IO completed!");
  // update UI...
}
```

Notice that the behavior here is similar to the modal dialog, invokeAndBlock "blocked" the current thread despite the fact that it is the EDT and performed the run() method in a separate thread. When run() completes the EDT is resumed. All the while repaints and events occur as usual, you can have invokeAndBlock calls occurring while another invokeAndBlock is still pending there are no limitations here although we would recommend against it since invokeAndBlock does carry some overhead.

As you can see this is a very simple approach for thread programming in UI, you don't need to block your flow and track the UI thread. You can just program in a way that seems sequential (top to bottom) but really uses multi-threading correctly without blocking the EDT.

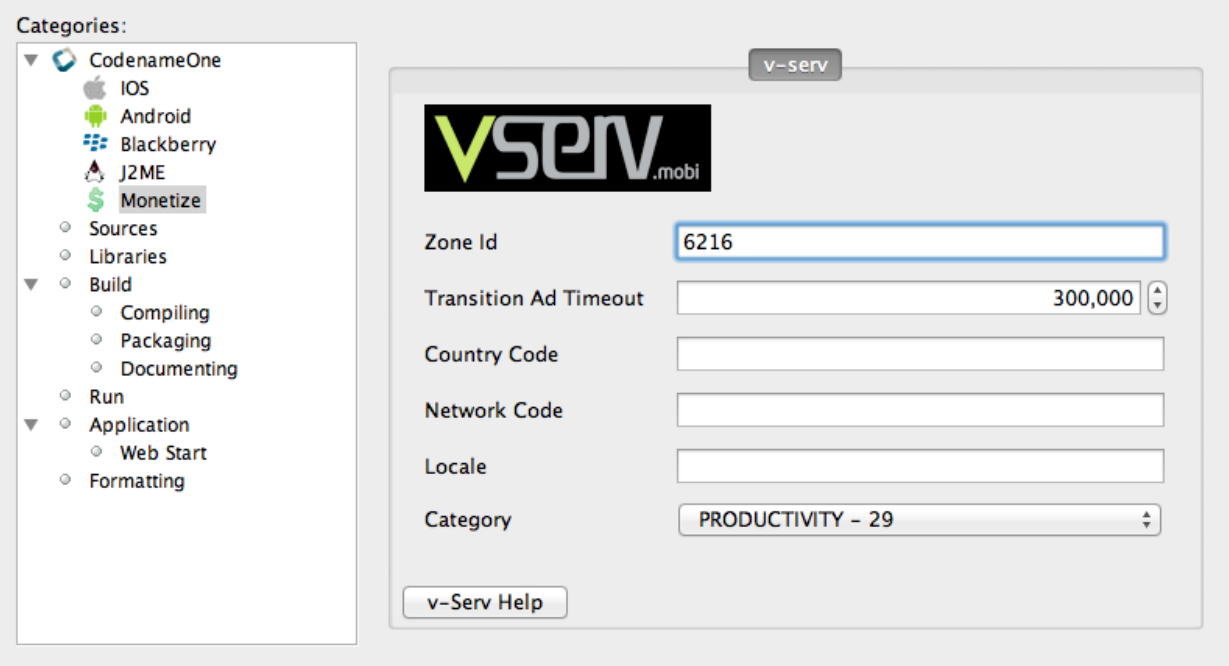
Monetization

Codename One tries to make the lives of software developers easier by integrating several forms of builtin monetization solutions such as ad network support, in-app-purchase etc. The Codename One integration is only applicable when developing the application, the actual integration is a matter of runtime relationship with the service provider[3].

Ad Networks

vserv

The vserv ad network provides a unique proposition where ads are displayed in full screen before during and sometimes after application execution. The biggest feature within vserv is its ability to seamlessly integrate with an existing application and provide value to the developers without changing a single line of application code.



[3] To Clarify: all payment and financial transactions go through the monetization provider and not through Codename One. E.g. Ad network revenue is the property of the developer and Codename One doesn't take any cut from the developers!

Codename One provides deep integration with this unique ad network the the IDE plugin where developers can input their vserv Zone Id (login to vserv to acquire a zone Id). Setting the zone ID to an empty string disables vserv ads, zone ID 6216 is useful for debugging purposes.

The transition ad timeout indicates the amount of time to wait before pushing an ad between transitions in milliseconds. You can set it to a very large number to disable that functionality.

The other properties are entirely optional and allow vserv to better target its ads to different application needs.

Inneractive

To integrate Inneractive banner ad's please register first using http://console.inner-active.com/iamp/publisher/register?ref_id=affiliate_CodenameOne.

Once registered you will be able to create an application id with which you will be able to associate ads with your account.

To initiate the inneractive ads you will need to enter the following line in your application init(Object) or start() method:

```
AdsService.setAdsProvider(InnerActive.class);
```

From here on you can just use the standard Codename One Ads Component and place it as you wish within the UI or drag it into place within the GUI builder. The Ads Component has multiple properties you can set to indicate your activity but the most important one is the setAppID() attribute (appId in the GUI builder) with which you can determine the application that will receive the payouts for the ads.

In App Purchase

Codename One's in app purchase API's try to generalize 3 different concepts for purchase:

1. Gooles in app purchase
2. Apples in app purchase
3. Mobile payments for physical goods

While all 3 approaches end up with the developer getting paid, all 3 take a different approach to the same idea. Google and Apple work with "products" which you can define and buy through their respective stores. You need to define the product in the development environment and then send the user to purchase said product.

Once the product is purchased you receive an event that the purchase was completed and you can act appropriately. On the other hand mobile payments are just a transfer of a sum of money. Both Google's and Apple's stores prohibit the sale of physical goods via the stores, so a mobile payment system needs to be used for those cases.
This is where the similarity ends between the Google & Apple approach. Google expects developers to build their own storefront and provides developers with an API to extract the data in order to construct said storefront. Apple expects the developers to open its storefront to perform everything.
We tried to encode all 3 approaches into the purchase API which means you would need to handle all 3 cases when working. Unfortunately these things are very hard to simulate and can only be properly tested on the device.

So to organize the above we have:

1. Managed payments - payments are handled by the platform. We essentially buy an item not transfer money (in app purchase).
2. Manual payments - we transfer money, there are no items involved.

```
final Purchase p = Purchase.getInAppPurchase(false);

if(p != null) {
   if(p.isManualPaymentSupported()) {
      purchaseDemo.addComponent(new Label("Manual Payment Mode"));
      final TextField tf = new TextField("100");
      tf.setHint("Send us money, thanks");
      Button sendMoney = new Button("Send Us Money");
      sendMoney.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
            p.pay(Double.parseDouble(tf.getText()), "USD");
         }
      });
      purchaseDemo.addComponent(tf);
      purchaseDemo.addComponent(sendMoney);
   } else {
      if(p.isManagedPaymentSupported()) {
         purchaseDemo.addComponent(new Label("Managed Payment Mode"));
         for(int iter = 0 ; iter < ITEM_NAMES.length ; iter++) {
            Button buy = new Button(ITEM_NAMES[iter]);
            final String id = ITEM_IDS[iter];
            buy.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent evt) {
                  p.purchase(id);
```

```
                }
            });
            purchaseDemo.addComponent(buy);
        }
    }
  }
} else {
  purchaseDemo.addComponent(new Label("Payment unsupported on this device"));
}
```

Graphics, Drawing & Images

This chapter covers the basics of drawing manually using the Codename One API, notice that drawing is considered a low level API which might introduce some platform fragmentation.

Basics - Where & How Do I Draw Manually?

The Graphics class is responsible for drawing basics, shapes, images and text, it is never instantiated by the developer and is always passed on by the Codename One API.
You can gain access to a Graphics object by doing one of the following:

- Derive Component or a subclass of Component - within Component there are several methods that allow developers to modify the drawing behavior, notice that Form is a subclass of component and thus features all of these methods. These can be overridden to change the way the component is drawn:
    - paint(Graphics) - invoked to draw the component, this can be overridden to draw the component from scratch.
    - paintBackground(Graphics)/paintBackgrounds(Graphics) - these allow overriding the way the component background is painted although you would probably be better off implementing a painter (see below).
    - paintBorder(Graphics) - allows overriding the process of drawing a border, notice that border drawing might differ based on the style of the component.
    - paintComponent(Graphics) - allows painting only the components contents while leaving the default paint behavior to the style.
    - paintScrollbars(Graphics),paintScrollbarX(Graphics),paintScrollbarY(Graphics) - allows overriding the behavior of scrollbar painting.
- Implement the painter interface, this interface can be used as a GlassPane or a background painter.
  The painter interface is a simple interface that includes 1 paint method, this is a useful way to allow developers to perform custom painting without subclassing component. Painters can be chained together to create elaborate paint behavior by using the PainterChain class.
    - Glass pane - a glass pane allows developers to paint on top of the form painting. This allows an overlay effect on top of a form.
      For a novice it might seem that a glass pane is similar to overriding the Form's paint method and drawing after super.paint(g) completed. This isn't the case. When a component repaints (by invoking the repaint() method) only that

component is drawn and Form's paint() method wouldn't be invoked. However, the glass pane painter is invoked for such cases and would work exactly as expected.
Container has a glass pane method called paintGlass(Graphics), which can be overridden to provide a similar effect on a Container level. This is especially useful for complex containers such as Table which draws its lines using such a methodology.

- Background painter - the background painter is installed via the style, by default Codename installs a custom background painter of its own. Installing a custom painter allows a developer to completely define how the background of the component is drawn.

A paint method can be implemented by deriving a Form as such:

```
public MyForm {
  public void paint(Graphics g) {
      // red color
      g.setColor(0xff0000);

      // paint the screen in red
      g.fillRect(getX(), getY(), getWidth(), getHeight());

      // draw hi world in white text at the top left corner of the screen
      g.setColor(0xffffff);
      g.drawString("Hi World", getX(), getY());
  }
}
```

Images

Codename One has quite a few image types: loaded, RGB (builtin), RGB (Codename One), Mutable, EncodedImage,SVG, Multi-Image & Timeline. There are also FileEncodedImage, FileEncodedImageAsync, StorageEncodedImage/Async which will be covered in the IO section.

Here are the pros/cons and logic behind every image type and how its created:

- Loaded Image - this is the basic image you get when loading an image from the jar or network using Image.createImage(String)/Image.createImage(InputStream)/Image.createImage(byte[], int, int).
  In some platforms (e.g. MIDP) calling getGraphics() on an image like this will throw an exception (its immutable in MIDP terms), this is true for almost all other images as well.

This restriction might not apply for all platforms.
The image is encoded based on device logic and should be reasonably efficient.

- RGB Image (internal) - close cousin of the loaded image. This image is created using the method Image.createImage(int[], int, int) and receives ARGB data forming the image. It is usually (although not always) a high color image. Its more efficient than the Codename One RGB image but can't be modified, at least not on the pixel level.

- RGBImage (Codenme One) - constructed via the RGBImage constructors this image is effectively an ARGB array that can be drawn by Codename One. On many platforms this is quite inefficient but for some pixel level manipulations there is just no other way.

- EncodedImage - created via the encoded image static methods, the encoded image is effectively a loaded image that is "hidden". When creating an encoded image only the PNG (or jpeg etc.) is loaded to an array in RAM. Normally such images are very small relatively so they can be kept in memory without much effect. When image information is needed (e.g. pixels, dimension etc.) the image is decoded into RAM and kept in a weak/sort reference.
  This allows the image to be cached for performance and allows the garbage collector to reclaim it when the memory becomes scarce.
  Encoded image is not final and can be derived to produce complex image fetching strategies such as lazily loading an image from the filesystem (read more about it in the IO section).

- SVG - SVG's can be loaded directly via Image.createSVG() if Image.isSVGSupported() returns true. When adding SVG's via the Codename One Designer fallback images are produced for devices that do not support SVG. The fallback images are effectively multi-images.

- Multi-Image - The multi-image is seamless to developers it is strictly a design time feature, during runtime an EncodedImage is returned whenever a multi-image is used. In the Codename One Designer one can add several images based on the DPI of the device (one of several predefined ranges). When loading the resource file irrelevant images are skipped thus saving the additional memory.
  Multi-images are ideal for icons or small artifacts that are hard to scale properly. They are not meant to replace things such as 9-image borders etc. since adapting them to every resolution or to device rotation isn't practical.
  9-image borders use multi-images by default internally to keep their appearance more refined on the different DPI’s.

- Timeline - Timeline's allow rudimentary animation and enable GIF importing using the Codename One Designer. Effectively a timeline is a set of images that can be moved rotated, scaled & blended to provide interesting animation effects. It can be created manually using the Timeline class.

All image types are mostly seamless to use and will just work with drawImage and various image related image API's for the most part with caveats on performance etc. For animation images the code must invoke images animate() method (this is done automatically by Codename One when placing the image as a background or as an icon! You only need to do it if you invoke drawImage in code rather than use a builtin component).
All images might also be animated in theory e.g. my gif implementation returned animated gifs from the standard Loaded Image methods and this worked pretty seamlessly (since Icons's and backgrounds just work). To find out if an image is animated you need to use the isAnimation() method, currently SVG images are animated in MIDP but most of our ports don't support GIF animations by default (although it should be easy to add to some of them).

Performance and memory wise you should read the above carefully and be aware of the image types you use. The Codename One designer tries to conserve memory and be "clever" by using only encoded images, while these are great for low memory they are not as efficient as loaded images in terms of speed. Also when scaled these images have much larger overhead since they need to be converted to RGB, scaled and then a new image is created. Keeping all these things in mind when optimizing a specific UI is very important.

Glass Pane

The GlassPane in Codename One is inspired by the Swing GlassPane & layered pane with quite a few twists. We tried to imagine how Swing developers would have implemented the glass pane knowing what they do now about painters and Swings learning curve. But I'm getting ahead of myself, what is the glass pane?

A typical Codename One application is essentially composed of 3 layers (this is a gross simplification though), the bg painters are responsible for drawing the background of all components including the main form. The component draws its own content which might overrule the painter and the glass pane paints last...

Essentially the glass pane is a painter that allows us to draw an overlay on top of the Codename One application. Initially we didn't think we need a glass pane, we used to suggest that people should override the form's paint() method to reach the same result. Feel free to try and guess why this failed before reading the explanation in the next paragraph.

Overriding the paint method of a form worked initially, when you enter a form this behaves just as you would expect. However, when modifying an element within the form only that element gets repainted not the entire form! So if I had a form with a Button and text drawn on top using the Form's paint method it would get erased whenever the button got focus.

Thats good for the forms paint method, calling the forms paint method would be REALLY expensive for every little thing that occurs in Codename One. However, we do want overlays for some things and we don't need to repaint every component in the screen to get them. The glass pane is called whenever a component gets painted, it only paints within the clipping region of the component hence it won't break the rest of the glass pane.

The painter chain is a tool that allows us to chain several painters together to perform different logistical tasks such as a validation painter coupled with a fade out painter. The sample below shows a crude validation panel that allows us to draw error icons next to components while exceeding their physical bounds as is common in many user interfaces

```
public class ValidationPane implements Painter {
 private Vector components = new Vector();
 private static Image error;
 public ValidationPane(Form parentForm) {
    try {
       if(error == null) {
          error = Image.createImage("/error.png");
       }
    } catch (IOException ex) {
       ex.printStackTrace();
    }
PainterChain.installGlassPane(parentForm, this);
 }

 public void paint(Graphics g, Rectangle rect) {
    for(int iter = 0 ; iter < components.size() ; iter++) {
       Component c = (Component) components.elementAt(iter);
       if(c == null) {
          components.removeElementAt(iter);
          continue;
       }
       Object p = c.getClientProperty(VALIDATION_PROP);
       int x = c.getAbsoluteX();
       int y = c.getAbsoluteY();
       x -= error.getWidth() / 2;
```

```
            y += c.getHeight() - error.getHeight() / 2;
            g.drawImage(error, x, y);
        }
    }

    public void addInvalid(Component c) {
        components.addElement(c);
    }

    public void removeInvalid(Component c) {
        components.removeElement(c);
    }
}
```

File System, Storage, Network & Parsing

In this chapter we cover the IO frameworks which include everything from network to storage, filesystem and parsing.

Externalizable Objects

Codename One provides the externalizable interface which is similar to the Java SE externalizable interface. This interface allows an object to declare itself as externalizable for serialization (so an object can be stored in a file/storage or sent over the network). However, due to the lack of reflection and use of obfuscation these objects must be registered with the Util class.
Codename One will probably never support the Java SE Serialization API due to the size issues and complexities related to obfuscation.

The major objects used by Codename One are externalizable by default:
String, Vector, Hashtable, Integer, Double, Float, Byte, Short, Long, Character, Boolean, Object[], byte[], int[], float[], long[], double[].

Externalizing an object such as h below should work just fine:

```
Hashtable h = new Hashtable();
h.put("Hi","World");
h.put("data", new byte[] {...});
```

However, notice that some things aren't polymorphic e.g. if I will externalize a String array I will get back an Object array since String arrays aren't supported.

So implementing the Externalizable interface is only important when we want to store a proprietary object. In this case we must register the object with the com.codename1.io.Util class so the externalization algorithm will be able to recognize it by name by invoking:

```
Util.register("MyClass", MyClass.class);
```

A externalizable objects must have a default public constructor and must implement the following 4 methods:

```
  public int getVersion();
   public void externalize(DataOutputStream out) throws IOException;
   public void internalize(int version, DataInputStream in) throws IOException;
```

```
public String getObjectId();
```

The version just returns the current version of the object allowing the algorithm to change in the future (the version is then passed when internalizing the object). The object id is a String uniquely representing the object, it usually corresponds to the class name (in the example above the Unique Name should be MyClass).

Storage vs. File System

The question of storage vs. file system is often confusing for novice mobile developers.

Generally storage is where you store information that will be deleted if the application is removed. It is private to the application and is supported by every platform although implementations sometimes differ by a great deal (e.g. in J2ME/RIM storage is really a type of byte array store called RMS and not a file system, Codename One hides that fact).

A file system can span over an SD card area and has a hierarchy/rules. Not all phones support a “proper” file system e.g. the iPhone doesn’t work well with such stepping outside of the applications boxed area.

When in doubt we always recommend using Storage which is simpler.

Storage

Storage is accessed via the com.codename1.io.Storage class. It is not a hierarchy and contains the ability to list/delete and write to named storage entries.
The Storage API also provides convenient methods to write objects to Storage and read them from Storage specifically readObject & writeObject.

File System

The file system is accessed via the com.codename1.io.FileSystemStorage class. It maps to the underlying OS’s file system API providing all the common operations on a file name from opening to renaming and deleting.
Notice that the file system API is somewhat platform specific in its behavior, all paths used the API should be absolute otherwise they are not guaranteed to work.

Cloud Storage

Notice: the cloud storage is a premium paid service. Codename One offers a free quota for all developers using its platform.

Cloud storage is an API that allows us to persist objects into Codename One's cloud servers hosted by the Google App Engine. This guarantees high reliability and speed, it also implies some constraints.

Cloud objects are stored using a distributed object database, don't think of them as you would of tables or typical SQL like storage since they are backed by Google's big table API. Each cloud object is very much like a Map allowing set/get operations on properties. An object can have a type associated with it as well as a visibility scope for the world.

A simple example might be in order:

```
CloudObject obj = new CloudObject("MyObject",
                        CloudObject.ACCESS_PUBLIC_READ_ONLY);
obj.setString("txt", title.getText());
obj.setIndexString(1, title.getText());
CloudStorage.getInstance().save(obj);
int result = CloudStorage.getInstance().commit();
if(result != CloudStorage.RETURN_CODE_SUCCESS) {
   Dialog.show("Cloud Error", "Error " + result, "OK", null);
}
```

As you can see, the code above creates and stores a cloud object, there are several things to notice about the example above:

1. Cloud object is a specific Codename One type, you cannot store arbitrary objects.
2. Values can be anything
3. We can define object visibility scope
4. There is a special index column
5. Save doesn't actually send data to the server
6. Commit is synchronous which means we know whether the commit succeeded in the next line.

Lets go over this line by line:

```
CloudObject obj = new CloudObject("MyObject",
                        CloudObject.ACCESS_PUBLIC_READ_ONLY);
```

Here we create a new cloud object, we give it a type (sort of like a class). Notice that properties we store into the object can be anything and don't have to be identical for objects that have the same type, this is mostly for your convenience as developers.
Then we give the object its visibility scope, when the visibility is defined you will only be able to work with an object that is within your scope. Cloud objects have 5 levels of visibility:

- ACCESS_PUBLIC - A world visible/modifiable object!

- ACCESS_PUBLIC_READ_ONLY - A world visible object! Can only be modified by its creator.
- ACCESS_APPLICATION - An application visible/modifiable object!
- ACCESS_APPLICATION_READ_ONLY - An application scope readable object! Can only be modified by its creator
- ACCESS_PRIVATE - An object that can only be viewed or modified by its creator

When creating an object you determine its scope and once the scope is assigned it cannot be modified, you will need to create a new object to do so. When querying you can only query one scope (more on that later).
Application scope is determined by your application package and developer email, since these are entirely unique and can't be occupied by another developer you are safe to assume that only your applications have access to that data. However, since this data is visible it isn't hacker safe. Any sensitive data should be password protected.
Some of the scopes require a user identity in order to modify or access an object, a unique user is automatically created when logging into an application. However, you can login explicitly with a specific user by using the CloudPersona API. E.g.: CloudPersona.createFromToken(String);

The creatFromToken method initializes the persona based on a token, since this method assumes binary transfer of a completed token the token isn't verified in any way and the user is considered logged in. The idea here is that when a user logs in using some other means (e.g. Facebook), identifying information e.g. the users email can be used as a "token". That way when a user logs in from another device the same token (email) would be used and he would have the same objects visible to him on both devices.
A user doesn't have to be a single person, it can be a corporation identity or any such group thus allowing the whole group to share a single token to get access to the private scope together.

```
obj.setString("txt", title.getText());
```

Here we see a string being placed into the object store, you can place Strings and numbers into the object store but not too much data. String length is limited to 500 bytes and objects can't be too large or you will get a server failure. We suggest staying well below the 100kb mark.
To store large files you will need to use the filestore API which will be explained bellow.

```
obj.setIndexString(1, title.getText());
```

Querying in an object datastore is pretty difficult, since you can define a property to be almost anything, we also want the queries to be REALLY fast even on stores containing more than 1 million entries.
To accomplish both goals we created 10 index entries in the object store (from 1 to 10) into which you can put any arbitrary data that you can query or sort. E.g. say you have an entity such as:

```
CloudObject o = ...;
o.setString("firstName", first);
o.setString("surname", last);
```

And you would like to sort them in a case insensitive way based on firstName-lastName and by lastName-firstName. To do this you will need to create two indexes:

```
o.setIndexString(1, (first + " " + last).toLowerCase());
o.setIndexString(2, (last + " " + first).toLowerCase());
```

This will allow you to order your responses either based on the first or the second index. This is a bit difficult but it guarantees ridiculously fast queries since every query is effectively an object lookup.

```
CloudStorage.getInstance().save(obj);
```

The cloud storage class is the singleton that allows you to save, delete, query and refresh your objects.

```
int result = CloudStorage.getInstance().commit();
```

Most modification operations aren't sent to the server immediately, they should be committed or rolled back which can be done synchronously on the EDT or asynchronously. Commit returns a server response which can indicate the type of failure if a failure has occurred.
Do not mistake commit/rollback to SQL type transactions, unlike SQL part of the operation can succeed while another part can fail, the commit command allows you to batch several operations into a single ordered server request which is more performant than sending multiple small requests.

A couple of other things we should keep in mind:

- Every object has a modification date which is tested against the server timestamp. This prevents two users/devices from changing a cloud object concurrently.
- Every object has a unique ID String identifying it, you can instantly access any object via that CloudId

We can now fetch the data we committed:

```
CloudObject[] objects = CloudStorage.getInstance().querySorted("MyObject", 1, true, 0, 10,
CloudObject.ACCESS_PUBLIC_READ_ONLY);
```

Notice that this query is synchronous (there is an asynchronous version of this query as well), its functionality is relatively simple though.
It searches for objects of type MyObject and returns them sorted in ascending order (the true argument) based on index number 1. It only returns the first 10 objects (start offset 0 and destination 10) and only searches the public read only scope.

Once fetched you can modify/save the objects and commit them.

This can be further simplified by binding a user interface directly to a Cloud Object using our Cloud Bind™ feature. To bind a component UI tree created in the GUI builder use the following:

```
CloudObject objectToBind = …;
objectToBind.bindTree(form, CloudObject.BINDING_IMMEDIATE, true);
```

There are several things happening here. The UI is assumed to have names associated with the relevant components, these names are used as the keys for the Cloud Object's created. The last argument indicates whether the UI should get the initial values for the entries from the cloud object or visa versa (true means the Cloud Object determines initial values).
The binding options in between has 3 different levels:

- BINDING_DEFERRED - Changes to the bound property won't be reflected into the bound cloud object until commit binding is invoked.
- BINDING_IMMEDIATE - Changes to the bound property will be reflected instantly into the cloud object
- BINDING_AUTO_SAVE - Changes to the bound property will be reflected instantly into the cloud object and the object would be saved immediately (not committed!).

These 3 essentially match typical UI use cases. Deferred binding is great for UI that has a "cancel" option, if the user presses cancel you don't have to do anything (although you can invoke cancelBinding() to cleanup). If the user presses save you will need to invoke the method commitBinding() and your changes will be applied (but not saved or committed despite the name).
Binding immediate changes the cloud object instantly which is great for UI's that don't have a save option (as is common on mobile devices and the Mac). The same is true for the last option only it goes further and invokes save for you. It still doesn't invoke commit which you will have to do at some point.

Cloud File Storage

Notice: the cloud file storage is a premium paid service. Codename One offers a free quota for all developers using its platform.

The cloud file storage is a complimentary service to the cloud storage API, it allows storing large files such as images, videos etc. It doesn't allow file modification only upload, delete and getting a URL to the file.
To use it with the Cloud Storage API just store the file key within a cloud object.
The code to work with cloud files is encoded into the CloudStorage class, simply use the methods uploadCloudFile, deleteCloudFile or getUrlForCloudFileId. The upload method returns

an id which is improbable to guess. You can use this id to delete or to get a URL which you can use to show the file.

SQL

Most newer devices contain one version of sqlite or another, sqlite is a very lightweight SQL database designed for embedding into devices. For portability **we recommend avoiding SQL** altogether since it is both fragmented between devices (different sqlite versions) and isn't supported on other devices.
In general SQL seems overly complex for most embedded device programming tasks.

If you wish to use SQL and are willing to work around the limitations just use

```
Database db = Display.getInstance().openOrCreate("databaseName");
```

Notice that db will be null if the SQL API isn't supported on the given platform. You can invoke standard queries on the database and traverse it using a Cursor object.

Network Manager & Connection Request

One of the more common problems in Network programming is spawning a new thread to handle the network operations. In Codename One this is done seamlessly and becomes unessential thanks to the NetworkManager class which effectively alleviates the need for managing network threads. The connection request class can be used to facilitate WebService requests when coupled with the JSON/XML parsing capabilities.
Currently Codename One only supports http/https connections due to limitations inherent in many devices/network operator backends. To open a connection one needs to use a ConnectionRequest object which has some similarities to the networking mechanism in JavaScript but is obviously somewhat more elaborate.

To send a get request to a URL one performs something like:

```
ConnectionRequest request = new ConnectionRequest();
request.setUrl(url);
request.setPost(false);
request.setContentType(contentType);
request.addRequestHeader(headerName, headerValue);
requestElement.addArgument(parameter, value);
request.addResponseListener(new ActionListener() {
   public void actionPerformed(ActionEvent ev) {
      NetworkEvent e = (NetworkEvent)ev;
      // … process the response
```

```
    }
});

// request will be handled asynchronously
NetworkManager.addToQueue(request);
```

Notice that you can also implement the same thing and much more by avoiding the response listener code and instead overriding the methods of the ConnectionRequest class which offers multiple points to override e.g.

```
ConnectionRequest request = new ConnectionRequest() {
  protected void readResponse(InputStream input) {
      // just read from the response input stream
  }

  protected void postResponse() {
      // invoked on the EDT after processing is complete to allow the networking code
      // to update the UI
  }

  protected void buildRequestBody(OutputStream os) {
      // writes post data, by default this “just works” but if you want to write this
     // manually then override this
  }
};
```

Debugging Network Connections

Codename One includes a Network Monitor tool which you can access via the file menu of the simulator, this tool reflects all the requests made through the connection requests and echos them all. Allowing you to track issues in your code/web service and see everything “going through the wire”.

This is a remarkably useful tool for optimizing and for figuring out what exactly is happening with your server connection logic.

Network Services

Codename One ships with a few default bindings for common network services such as downloading and caching images locally & RSS. You can find out more about these services in the services package.

UI Bindings & Utilities

Codename One provides several tools to simplify the path between networking/IO & GUI. A common task of showing a wait dialog or progress indication while fetching network data can be simplified by using the InfiniteProgress class e.g.:

```
InfiniteProgress ip = new InfiniteProgress();
Dialog dlg = ip.showInifiniteBlocking();
request.setDisposeOnCompletion(dlg);
```

The process of showing a progress bar for a long IO operation such as downloading is automatically mapped to the IO stream in Codename One using the SliderBridge class.

Logging & Crash Protection

Codename One includes a Log API that allows developers to just invoke Log.p(String) or Log.e(Throwable) to log information to storage.
As part of the premium cloud features it is possible to invoke Log.sendLog() in order to email a log directly to the developer account. Codename One can do that seamlessly based on changes printed into the log or based on exceptions that are uncaught or logged e.g.:

```
Log.setReportingLevel(Log.REPORTING_DEBUG);
DefaultCrashReporter.init(true, 2);
```

This code will send a log every 2 minutes to your email if anything was changed. You can place it within the init(Object) method of your application.
For a production application you can use Log.REPORTING_PRODUCTION which will only email the log on exception.

Codename One also supports a crash_protection: true build parameter. However, this argument causes significant performance overhead at the moment and is only recommended during development time. It allows developers to receive a stack trace for crashes and in logging. However, the stack traces are only limited to the Codename One and developer classes and don't apply to operating system classes.

Parsing: JSON, XML & CSV

Codename One has several built in parsers for JSON, XML & CSV formats which you can use to parse data from the internet or data that is shipping with your product. E.g. use the CSV data to setup default values for your application.

The parsers are all geared towards simplicity and small size, they don't validate and will fail in odd ways when faced with broken data.

CSV is probably the easiest to use, the "Comma Separated Values" format is just a list of values separated by commas (or some other character) with new lines to indicate another row in the table. These usually map well to an Excel spreadsheet or database table.

To parse a CSV just use the CSVParser class as such:

```
CSVParser parser = new CSVParser();
String[][] data = parser.read(stream);
```

The data array will contain a two dimensional array of the CSV data. You can change the delimiter character by using the CSVParser constructor that accepts a character.

The JSON "Java Script Object Notation" format is popular on the web for passing values to/from webservices since it works so well with JavaScript. Parsing JSON is just as easy but has two different variations. You can use the JSONParser class to build a tree of the JSON data as such:

```
JSONParser parser = new JSONParser();
Hashtable response = parser.parse(reader);
```

The response is a Hashtable containing a nested hierarchy of Vectors, Strings and numbers to represent the content of the submitted JSON. To extract the data from a specific path just iterate the Hashtable keys and recurs into it. Notice that there is a webservices demo as part of the kitchen sink showing the returned data as a Tree structure.

An alternative approach is to use the static data parse() method of the JSONParser class and implement a callback parser e.g.:

```
JSONParser.parse(reader, callback);
```

Notice that a static version of the method is used! The callback object is an instance of the JSONParseCallback interface which includes multiple methods. These will be invoked by the

parser to indicate internal parser states in a similar way to a traditional XML SAX event based parser.

Advanced readers might want to dig deeper into the processing language contributed by Eric Coolman which allows for xpath like expressions when parsing JSON & XML. Read about it in Eric's blog.

Last but not least is the XML parser, to use it just create an instance of the XMLParser class and invoke parse:

```
XMLParser parser = new XMLParser();
Element elem = parser.parse(reader);
```

The element contains children and attributes and represents a tag element within the XML document or even the document itself. You can iterate over the XML tree to extract the data from within the XML file.

Miscellaneous Features

This chapter covers various features of Codename One that don't quite fit in any of the other chapters.

Analytics Integration

One of the features in Codename One is builtin support for analytic instrumentation. Currently Codename One has builtin support for Google Analytics which provides reasonable enough statistics of application usage.

The infrastructure is there to support any other form of analytics solution of your own choosing.

Analytics is pretty seamless for a GUI builder application since navigation occurs via the Codename One API and can be logged without developer interaction. However, to begin the instrumentation one needs to add the line:

```
AnalyticsService.init(agent, domain);
```

To get the value for the agent value just create a Google Analytics account and add a domain, then copy and paste the string that looks something like UA-99999999-8 from the console to the agent string. Once this is in place you should start receiving statistic events for the application.

If your application is not a GUI builder application or you would like to send more detailed data you can use the Analytics.visit() method to indicate that you are entering a specific page.

Facebook Support

Facebook uses the Graph API[4] which is a JSON based web protocol that allows developers to traverse the information within facebook and update it. To work with Facebook you need to read about the process of creating a facebook application. A Facebook application identifies your application to Facebook and allows them to associate invocations/changes made by your application with you.
The main issue with Facebook is the authentication process which requires the OAuth standard to validate against the website. OAuth forces the user to login to the website and approve the

[4] See http://developers.facebook.com/docs/reference/api/

permissions requested by the application, once these credentials are given in the web browser the application is given a token which it can use for all its calls. This token can be reused between invocations so the user doesn't need to re-enter his password. However, the token needs to be revalidated in case the user is logged out or changed his settings.

The main class of interest is FaceBookAccess with which we obtain the token, notice that in the following code you should probably update all the strings to match your actual needs:

```
FaceBookAccess.setClientId("132970916828080");
FaceBookAccess.setClientSecret("6aaf4c8ea791f08ea15735eb647becfe");
FaceBookAccess.setRedirectURI("http://www.codenameone.com/");
FaceBookAccess.setPermissions(new String[]{"user_location", "user_photos",
"friends_photos", "publish_stream", "read_stream", "user_relationships", "user_birthday",
        "friends_birthday", "friends_relationships", "read_mailbox", "user_events",
"friends_events", "user_about_me"});
FaceBookAccess.getInstance().showAuthentication(new ActionListener() {

    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() instanceof String) {
            String token = (String) evt.getSource();
            String expires = Oauth2.getExpires();
            System.out.println("recived a token " + token + " which expires on " + expires);
            Storage.getInstance().writeObject("authenticated", "true");
            if(main != null){
                main.showBack();
            }
        } else {
            Exception err = (Exception) evt.getSource();
            err.printStackTrace();
            Dialog.show("Error", "An error occurred while logging in: " + err, "OK", null);
        }
    }
});
```

You can then get access to the wall by doing something like:

```
FaceBookAccess.getInstance().getWallFeed("me", (DefaultListModel) wall.getModel(), null);
```

Performance & Size

Reducing Resource File Size

Its easy to lose track of size/performance when you are working within the comforts of a visual tool like the Codename One Designer. When optimizing resource files you need to keep in mind one thing: it's all about image sizes.
Images will take up 95-99% of the resource file size, everything else is peanuts in comparison.

Like every optimization the first rule is to reduce the size of the biggest images which will provide your biggest improvements, for this purpose I introduced the ability to see image sizes in KB (see the menu option Images -> Image Sizes (KB).
This produces a list of images sorted by size with the amount of KB each takes. Often the top entries will be multi-images which include HD resolution values that can be pretty large. These very high resolution images take up a significant amount of space! Just going to the multi-images, selecting the unnecessary resolutions & deleting these HUGE images (note you can see the size in KB at the top right side in the image viewer) saves a HUGE amount of space.

Next you should probably use the "Delete Unused Images" menu option (it's also under the Images menu). This tool allows detecting and deleting images that aren't used within the theme/GUI.

If you have a very large image that is opaque you might want to consider converting it to JPEG and replacing the built in PNG's. JPEG's work on most devices and are much smaller.

You can use the excellent OptiPng tool to optimize image files right from the Codename One designer. To use this feature you need to install OptiPng then select "Images -> Launch OptiPng" from the menu. Once you do that the tool will automatically optimize all your PNG's.

When faced with size issues make sure to check the size of your res file, if your JAR file is large open it with a tool such as 7-zip and sort elements by size. Start reviewing which elements justify the size overhead.

Improving Performance

There are quite a few things you can do as a developer in order to improve the performance and memory footprint of a Codename One application. This sometimes depends on specific device behaviors but some of the tips here are true for all devices.
The simulator contains some tools to measure performance overhead of a specific component and also detect EDT blocking logic. Other than that follow these guidelines to create more performance code:

- Avoid round rect borders - they have a huge overhead on all platforms. Use image borders instead (counter intuitively they are MUCH faster).
- Bitmap fonts are pretty slow on many platforms, we recommend avoiding them. Methods such as stringWidth, can also be very slow on some platforms. This means that reflowing the UI (preferred size calls string width) can become very expensive.
- Read carefully the Image section and make sure to make conscious choices regarding the image types you choose.
- Some older devices (symbian mostly) perform very badly with translucent images.
- Use larger images when tiling or building image borders, using a 1 pixel (or event a few pixels) wide or high image and tiling it repeatedly can be very expensive.

Performance Monitor

The performance monitor tool is accessible via the menu option in the simulator and it pops up a dialog showing some information that can help you in debugging slow performing UI's.
You will be able to see the amount of time and amount of paint operations that occur for every component as well as printouts about every image allocation and RAM statistics for said allocations.

Advanced Topics/Under The Hood

This chapter covers the more advanced topics explaining how Codename One actually works.

Sending Arguments To The Build Server

When sending a build to the server you can provide additional parameters to the build which will be incorporated into the build process on the server to hint on multiple different build time options.

Here is the current list of supported arguments, keep up with this page since we intend to update it frequently with new options:

Name	Description
android.min_sdk_version	defaults to '7". Used in the manifest to indicate the android:minSdkVersion property.
android.xapplication	defaults to an empty string. Allows developers of native Android code to add text within the application block to define things such as widgets etc.
android.xpermissions	additional permissions for the Android manifest
android.stack_size	Size in bytes for the Android stack thread
block_server_registration	true/false flag defaults to false. By default Codename One applications register with our server, setting this to true blocks them from sending information to our cloud. We keep this data for statistical purposes and intend to provide additional installation stats in the future.
ios.project_type	one of ios, ipad, iphone (defaults to ios). Indicates whether the resulting binary is targeted to the iphone only or ipad only.
ios.statusbar_hidden	true/false defaults to false. Hides the iOS status bar if set to true.
ios.prerendered_icon	true/false defaults to false. The iOS build process adapts the submitted icon for iOS conventions (adding an overlay) that might not be appropriate on some icons. Setting this to true leaves the

	icon unchanged (only scaled).
ios.application_exits	true/false (defaults to true). Indicates whether the application should exit immediately on home button press. The default is to exit, leaving the application running is only partially tested at the moment.
ios.interface_orientation	UIInterfaceOrientationPortrait by default. Indicates the default initial orientation, one of: UIInterfaceOrientationPortrait, UIInterfaceOrientationPortraitUpsideDown, UIInterfaceOrientationLandscapeLeft, UIInterfaceOrientationLandscapeRight
rim.askPermissions	true/false defaults to true. Indicates whether the user is prompted for permissions on RIM devices.
crash_protect	true/false defaults to true. Only applicable to paying users. Instruments an application with on device exception logging which allows the application to send a crash log to the server when it fails.
rim.ignor_legacy	true/false defaults to false. When set to true the RIM build targets only 5.0 devices and newer and doesn't build the 4.x version.
ios.plistInject	entries to inject into the iOS plist file during build.

The Architecture Of The GUI Builder

The Codename One GUI builder has several unique underlying concepts that aren't as common among such tools, in this article I will try to clarify some of these basic ideas.

Basic Concepts

The Codename One Designer isn't a standard code generator, the UI is saved within the resource file and can be designed without the source files available. This has several advantages:

1. No fragile generated code to break.
2. Designers who don't know Java can use the tool.
3. The "Codename One LIVE!" application can show a live preview of your design as you build it.
4. Images and theme settings can be integrated directly with the GUI without concern.
5. The tool is consistent since the file you save is the file you run.
6. GUI's/themes can be downloaded dynamically without replacing the application (this can reduce download size).

7 It allows for control over application flow. It allows preview within the tool without compilation.

This does present some disadvantages and oddities:

1 Its harder to integrate custom code into the GUI builder/designer tool.
2 The tool is somewhat opaque, there is no "code" you can inspect to see what was accomplished by the tool.
3 If the resource file grows too large it can significantly impact memory/performance of a running application.
4 Binding between code and GUI isn't as intuitive and is mostly centralized in a single class.

In theory you don't need to generate any code, you can load any resource file that contains a UI element as you would normally load a Resource file:

```
Resources r = Resources.open("/myFile.res");
```

Then you can just create a UI using the UIBuilder API:

```
UIBuilder u = new UIBuilder();
Container c = u.createContainer(r, "uiNameInResource");
```

(Notice that since Form & Dialog both derive from Container you can just downcast to the appropriate type).

This would work for any resource file and can work completely dynamically! E.g. you can download a resource file on the fly and just show the UI that is within the resource file... That is what Codename One LIVE! is doing internally.

IDE Bindings

While the option of creating a Resource file manually is powerful, its not nearly as convenient as modern GUI builders allow. Developers expect the ability to override events and basic behavior directly from the GUI builder and in mobile applications even the flow for some cases.

To facilitate IDE integration we decided on using a single Statemachine class, similar to the common controller pattern. We considered multiple classes for every form/dialog/container and eventually decided this would make code generation more cumbersome.

The designer effectively generates one class "StatemachineBase" which is a subclass of UIBuilder (you can change the name/package of the class in the Codename One properties file

at the root of the project). StatemachineBase is generated every time the resource file is saved assuming that the resource file is within the src directory of a Codename One project. Since the state machine base class is always generated, all changes made into it will be overwritten without prompting the user.

User code is placed within the Statemachine class, which is a subclass of the Statemachine Base class. Hence it is a subclass of UIBuilder!

When the resource file is saved the designer generates 2 major types of methods into Statemachine base:
1. Finders - findX(Container c). A shortcut method to find a component instance within a hierarchy of containers. Effectively this is a shortcut syntax for UIBuilder.findByName(), its still useful since the method is type safe. Hence if a resource component name is changed the find() method will fail in subsequent compilations.

2. Callback events - these are various callback methods with common names e.g.: onCreateFormX(), beforeFormX() etc. These will be invoked when a particular event/behavior occurs.

Within the GUI builder, the event buttons would be enabled and the GUI builder provides a quick and dirty way to just override these methods. To prevent a future case in which the underlying resource file will be changed (e.g formX could be renamed to formY) a super method is invoked e.g. super.onCreateFormX();

This will probably be replaced with the @Override annotation when Java 5 features are integrated into Codename One.

Working With The Generated Code

The generated code is rather simplistic, e.g. the following code from the tzone demo adds a for the remove button toggle:

```
protected void onMainUI_RemoveModeButtonAction(Component c, ActionEvent event) {
  // If the resource file changes the names of components this call will break notifying you that
you should fix the code
  super.onMainUI_RemoveModeButtonAction(c, event);
  removeMode = !removeMode;
  Container friendRoot = findFriendsRoot(c.getParent());
  Dimension size = null;
  if(removeMode) {
    if(Display.getInstance().getDeviceDensity() > Display.DENSITY_LOW) {
```

```
            findRemoveModeButton(c.getParent()).setText("Finish");
        }
    } else {
        size = new Dimension(0, 0);
        if(Display.getInstance().getDeviceDensity() > Display.DENSITY_LOW) {
            findRemoveModeButton(c.getParent()).setText("Remove");
        }
    }
    for(int iter = 0 ; iter < friendRoot.getComponentCount() ; iter++) {
        Container currentFriend = (Container)friendRoot.getComponentAt(iter);
        currentFriend.setShouldCalcPreferredSize(true);
        currentFriend.setFocusable(!removeMode);
        findRemoveFriend(currentFriend).setPreferredSize(size);
        currentFriend.animateLayout(800);
    }
}
```

As you can see from the code above implementing some basic callbacks within the state machine is rather simple. The method findFriendsRoot(c.getParent()); is used to find the "FriendsRoot" component within the hierarchy, notice that we just pass the parent container to the finder method. If the finder method doesn't find the friend root under the parent it will find the "true" root component and search there.
The friends root is a container that contains the full list of our "friends" and within it we can just work with the components that were instantiated by the GUI builder.
Implementing Custom Components There are two basic approaches for custom components:

1. Override a specific type - e.g. make all Form's derive a common base class.
2. Replace a deployed instance.

The first uses a feature of UIBuilder which allows overriding component types, specifically override createComponentInstance to return an instance of your desired component e.g.:

```
protected Component createComponentInstance(String componentType, Class cls) {
    if(cls == Form.class) {
        return new MyForm();
    }
    return super.createComponentInstance(componentType, cls);
}
```

This code allows me to create a unified global form subclass. That's very useful when I want so global system level functionality that isn't supported by the designer normally.

The second approach allows me to replace an existing component:

```
protected void beforeSplash(Form f) {
  super.beforeSplash(f);

  splashTitle = findTitleArea(f);

  // create a "slide in" effect for the title
  dummyTitle = new Label();
  dummyTitle.setPreferredSize(splashTitle.getPreferredSize());
  f.replace(splashTitle, dummyTitle, null);
}

protected void postSplash(Form f) {
  super.postSplash(f);

  f.replace(dummyTitle, splashTitle,
CommonTransitions.createSlide(CommonTransitions.SLIDE_VERTICAL, true, 1000));
  splashTitle = null;
  dummyTitle = null;
}
```

Notice that we replace the title with an empty label, in this case we do this so we can later replace it while animating the replace behavior thus creating a slide-in effect within the title. It can be replaced though, for every purpose including the purpose of a completely different custom made component. By using the replace method the existing layout constraints are automatically maintained.

Native Interfaces

Low level calls into the Codename One system, including support for making platform native API calls. Notice that when we say "native" we do not mean C/C++ always but rather the platforms "native" environment. So in the case of J2ME the Java code will be invoked with full access to the J2ME API's, in case of iOS an Objective-C message would be sent and so forth. Native interfaces are designed to only allow primitive types, Strings, arrays (single dimension only!) of primitives and PeerComponent values. Any other type of parameter/return type is prohibited. However, once in the native layer the native code can act freely and query the Java layer for additional information.

Furthermore, native methods should avoid features such as overloading, varargs (or any Java 5+ feature for that matter) to allow portability for languages that do not support such features (e.g. C).
Important! Do not rely on pass by reference/value behavior since they vary between platforms.

Implementing a native layer effectively means:

1. Creating an interface that extends NativeInterface and only defines methods with the arguments/return values declared in the previous paragraph.
2. Creating the proper native implementation hierarchy based on the call conventions for every platform within the native directory

E.g. to create a simple hello world interface do something like:

```
package com.my.code;
public interface MyNative extends NativeInteface {
    String helloWorld(String hi);
}
```

Then to use that interface use MyNative my = (MyNative)NativeLookup.create(MyNative.class);
Notice that for this to work you must implement the native code on all supported platforms!
To implement the native code use the following convention. For Java based platforms (Android, RIM, J2ME):
Just create a Java class that resides in the same package as the NativeInterface you created and bares the same name with Impl appended e.g.: MyNativeImpl. So for these platforms the code would look something like this:

```
package com.my.code;
public class MyNativeImpl implements MyNative {
    public String helloWorld(String hi) {
        // code that can invoke Android/RIM/J2ME respectively
    }
}
```

Notice that this code will only be compiled on the server build and is not compiled on the client. These sources should be placed under the appropriate folder in the native directory and are sent to the server for compilation.
For Objective-C, one would need to define a class matching the name of the package and the class name combined where the "." elements are replaced by underscores. One would need to provide both a header and an "m" file following this convention e.g.:

```
@interface com_my_code_MyNative : NSObject {
```

```
}
- (id)init;
- (NSString*)helloWorld:(NSString *)param1;
@end
```

Notice that the parameters in Objective-C are named which has no equivalent in Java. That is why the native method in Objective-C MUST follow the convention of naming the parameters "param1", "param2" etc. for all the native method implementations. Java arrays are converted to NSData objects to allow features such as length indication.

PeerComponent return values are automatically translated to the platform native peer as an expected return value. E.g. for a native method such as this: PeerComponent createPeer();
Android native implementation would need: View createPeer();
While RIM would expect: Field createPeer()
The iphone would need to return a pointer to a view e.g.: - (UIView*)createPeer; J2ME doesn't support native peers hence any method that returns a native peer would always return null.

Drag & Drop

Unlike other platforms who tried to create overly generic catch all API's we tried to make things as simple as possible. We always drag a component and always drop it onto another component, if something else is dragged to some other place it must be wrapped in a component, the logic of actually performing the operation indicated by the drop is the responsibility of the person implementing the drop.

There is a minor sample of this in the KitchenSink demo whose drag and drop behavior is implemented using this API. However, the KitchenSink demo relies on built in drop behavior of container specifically designed for this purpose.

To enable dragging a component it must be flagged as draggable using setDraggable(true), to allow dropping the component onto another component you must first enable the drop target with setDropTarget(true) and override some methods (more on that later).

Notice that is a drop target is a container that has children, dropping a component on the child will automatically find the right drop target. You don't have to make "everything" into a drop target.

You can override these methods in the draggable components:
getDragImage - this generates an image preview of the component that will be dragged. This automatically generates a sensible default so you don't need to override it.

drawDraggedImage - this method will be invoked to draw the dragged image at a given location, it might be useful to override it if you want to display some drag related information such an additional icon based on location etc. (e.g. a move/copy icon).

In the drop target you can override the following methods:
draggingOver - returns true is a drop operation at this point is permitted. Otherwise releasing the component will have no effect.
dragEnter/Exit - useful to track and cleanup state related to draging over a specific component.
drop - the logic for dropping/moving the component must be implemented here!

Notice that the Container class has a simple sample drop implementation you can use to get started.

Physics - The Motion Class

The motion class represents a physics operation that starts at a fixed time bound to the system current time millis value. The use case is entirely for UI animations and so many of its behaviors are simplified for this purpose.
The motion class can be replaced in some of the builting classes to provide a slightly different feel to some of the transition effects.

BiDi/RTL Language Support

BiDi is the term referring to bi-directional language support, generally RTL languages (right to left). There is plenty of information about RTL languages (Arabic, Hebrew, Syriac, Thaana) on the internet but as a brief primer here is a minor summary.

Most western languages are written from left to right (LTR), however some languages are normally written from right to left (RTL) speakers of these languages expect the UI to flow in the opposite direction otherwise it seems weird just like reading this word would be to most English speakers: "drieW".

The problem posed by RTL languages is known as BiDi (Bi-directional) and not as RTL since the "true" problem isn't the reversal of the writing/UI but rather the mixing of RTL and LTR together. E.g. numbers are always written from left to right (just like in English) so in an RTL language the direction is from right to left and once we reach a number or English text embedded in the middle of the sentence (such as a name) the direction switches for a duration and is later restored.

Codename One's support for bidi includes the following components:

- Bidi algorithm - allows converting between logical to visual representation for rendering
- Globabl RTL flag - default flag for the entire application indicating the UI should flow from right to left
- Individual RTL flag - flag indicating that the specific component/container should be presented as an RTL/LTR component (e.g. for displaying English elements within a RTL UI).
- RTL text field input
- RTL bitmap font rendering

Most of Codename One's RTL support is under the hood, the LookAndFeel global RTL flag can be enabled using:

```
UIManager.getInstance().getLookAndFeel().setRTL(true);
```

(Notice that setting the RTL to true implicitly activates the bidi algorithm).

Once RTL is activated all positions in Codename One become reversed and the UI becomes a mirror of itself. E.g. A softkey placed on the left moves to the right, padding on the left becomes padding on the right, the scroll moves to the left etc.
This applies to the layout managers (except for group layout) and most components. Bidi is mostly seamless in Codename One but a developer still needs to be aware that his UI might be mirrored for these cases.

Signing, Certificates & Provisioning

While Codename One can simplify allot of the grunt work in creating cross platform mobile applications, signing is not something that can be significantly simplified since it represents the developers individual identity in the markets. In this section we attempt to explain how to acquire certificates for the various platforms and how to set them up.

The good news is that this is usually a "one time issue" and once its done the work becomes easier (except for the case of iOS where a provisioning profile should be maintained).

iOS (iPhone/iPad)

iOS signing has two distinct modes: App Store signing which is only valid for distribution via iTunes (you won't be able to run the resulting application without submitting it to Apple) and development mode signing.

You have two major files to keep track of:
Certificate - your signature
Provisioning Profile - details about the application and who is allowed to execute it

You need two versions of each file (4 total files) one pair is for development and the other pair is for uploading to the itunes App Store.

Important: **You need to use a Mac in order to create a certificate file for iOS, methods to achieve this without a Mac produce an invalid certificate that fails on the server and leaves hard to remove residue[5].**

Adobe wrote a pretty decent signing tutorial on this subject for Air which you can pretty much follow to get the files we need (p12 and mobileprovision).

The first step you need to accomplish is signing up as a developer to Apple's iOS development program, even for testing on a device this is required! This step requires that you pay Apple 99 USD on a yearly basis.

[5] Seriously. An invalid name is stuck in the servers and you won't be able to build later even if you have a valid certificate. Its a pain for us so please follow the instructions properly and use a Mac for this stage.

The Apple website will guide you through the process of applying for a certificate at the end of this process you should have a distribution and development certificate pair. After that point you can login to the iOS provisioning portal where there are plenty of videos and tutorials to guide you through the process. Within the iOS provisioning portal you need to create an application ID and register your development devices.

You then create a provisioning profile which comes in two flavors: distribution (for building the release version of your application) and development. The development provisioning profile needs to contain the devices on which you want to test.

You can then configure the 4 files in the IDE and start sending builds to the Codename One cloud.

Android

Its really easy to sign Android applications if you have the JDK installed. Find the keytool executable (it should be under the JDK's bin directory) and execute the following command:
keytool -genkey -keystore Keystore.ks -alias [alias_name] -keyalg RSA -keysize 2048 -validity 15000 -dname "CN=[full name], OU=[ou], O=[comp], L=[City], S=[State], C=[Country Code]" -storepass [password] -keypass [password]

The elements in the brackets should be filled up based on this:
Alias: [alias_name] (just use your name/company name without spaces)
Full name: [full name]
Organizational Unit: [ou]
Company: [comp]
City: [City]
State: [State]
CountryCode: [Country Code]
Password: [password] (we expect both passwords to be identical)

Executing the command will produce a Keystore.ks file in that directory which you need to keep since if you lose it you will no longer be able to upgrade your applications! Fill in the appropriate details in the project properties or in the CodenameOne section in the Netbeans preferences dialog.

For more details see http://developer.android.com/guide/publishing/app-signing.html

RIM/BlackBerry

You can now get signing keys for free from RIM by going here. After obtaining the certificates and installing them on your machine (you will need the RIM development environment for this). You will have two files: sigtool.db and sigtool.csk on your machine (within the JDE directory hierarchy). We need them and their associated password to perform the signed build for Blackberry application.

J2ME

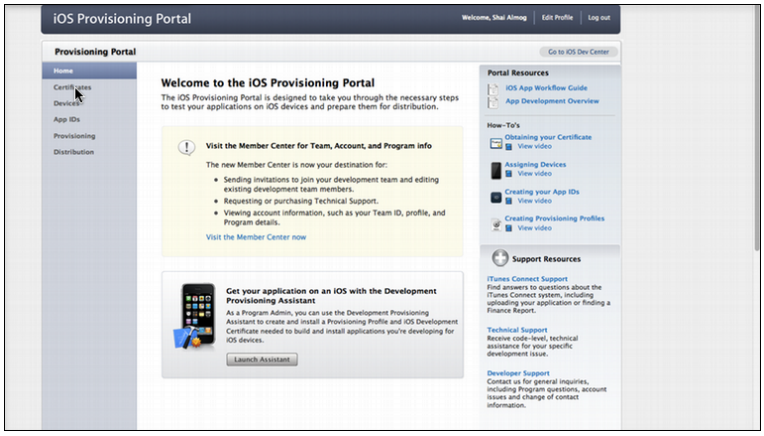
Currently signing J2ME application's isn't supported. You can use tools such as the Sprint WTK to sign the resulting jad/jar produced by Codename One.

Appendix: Working With iOS

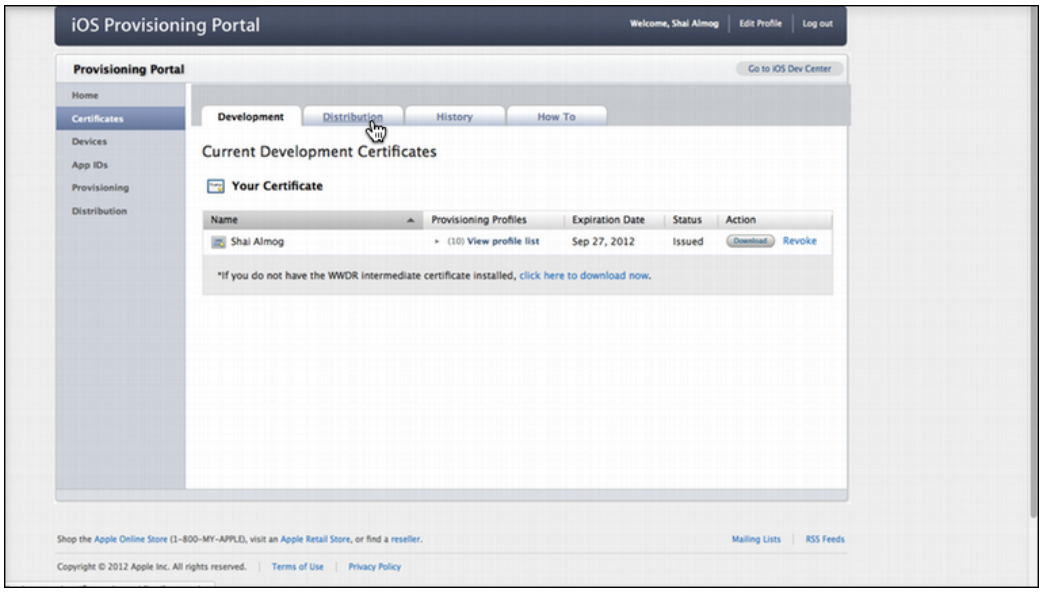
Provisioning Profile & Certificates

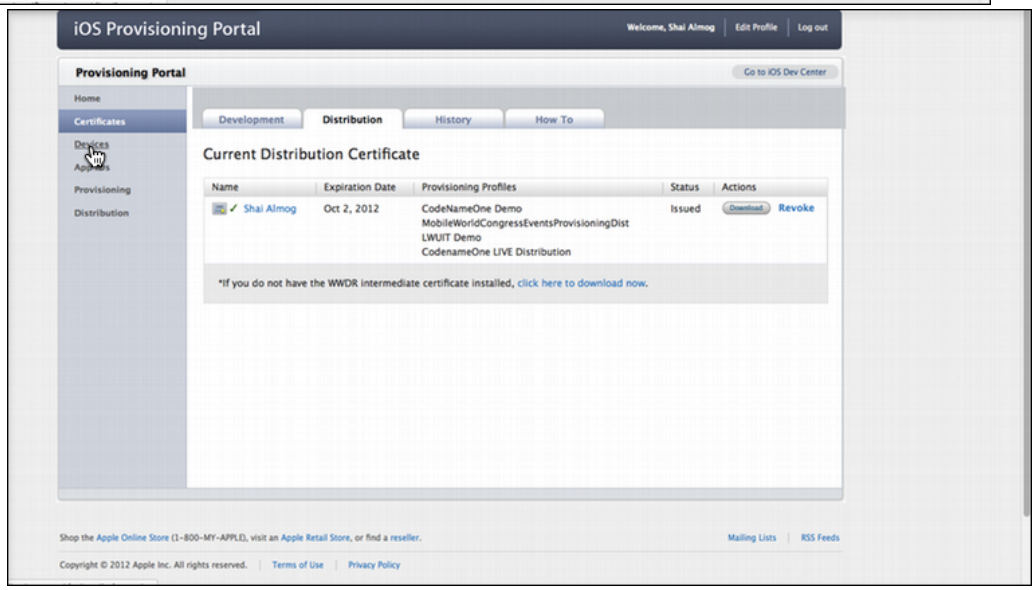
One of the hardest parts in developing for iOS is the total mess they made with their overly complex certificate/provisioning process. Relatively for the complexity the guys at Apple did a great job of hiding allot of the crude details but its still difficult to figure out where to start.

Start by logging in to the iOS provisioning portal

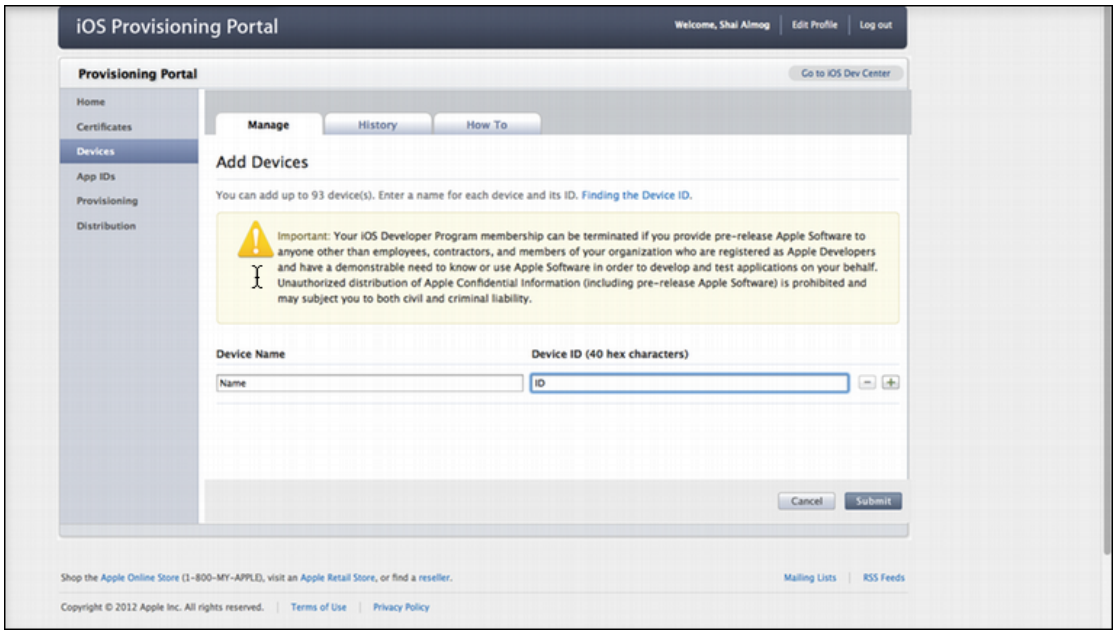


In the certificates section you can download your development and distribution certificates.

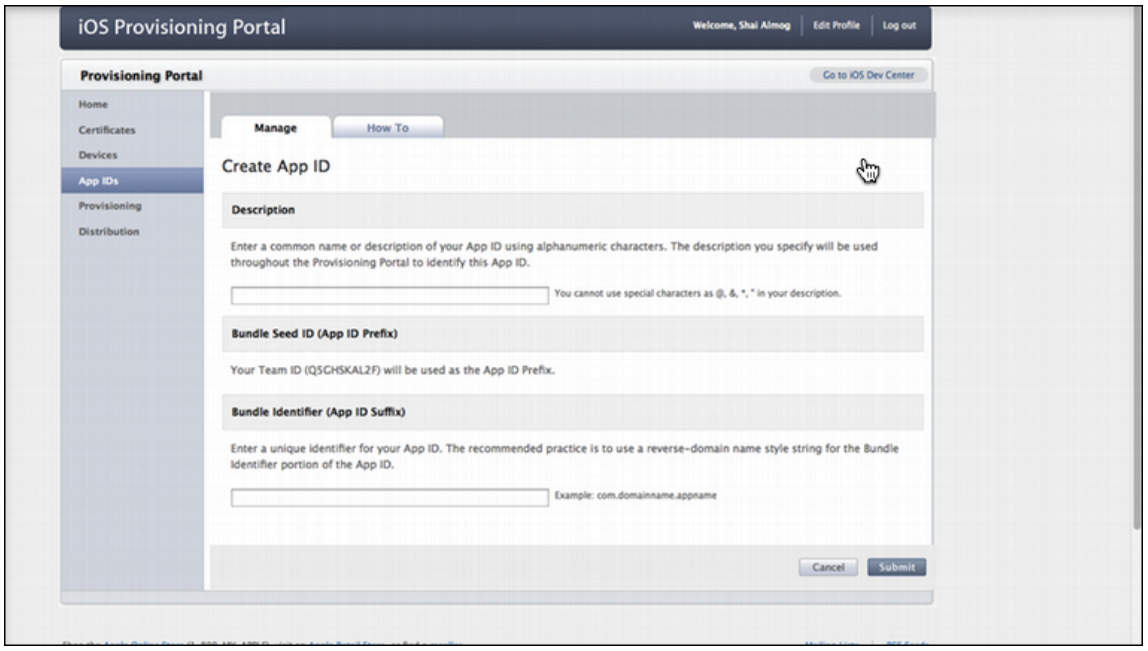




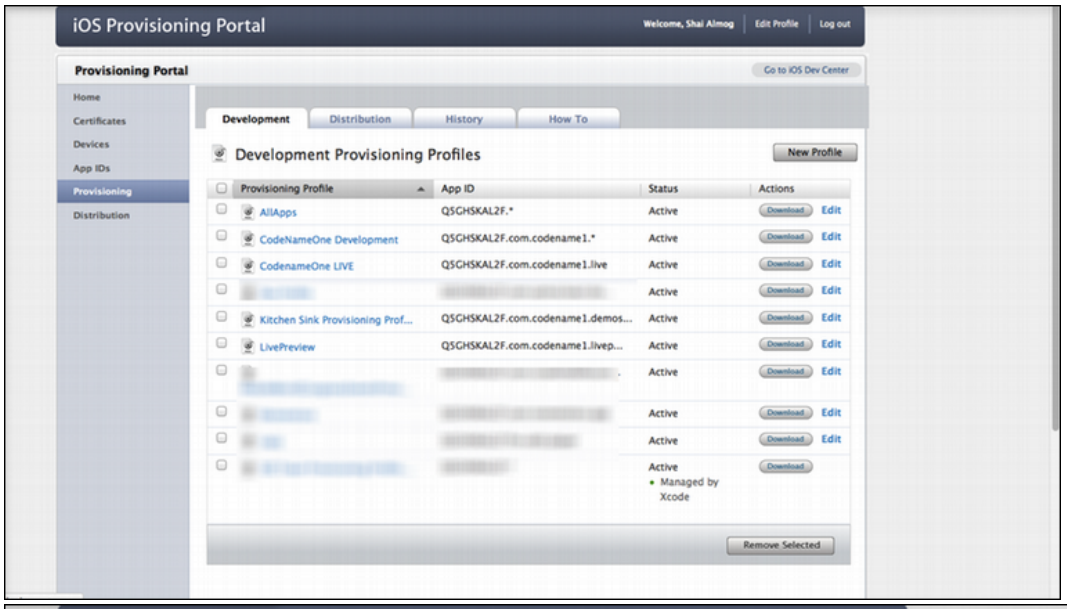
In the devices section add device id's for the development devices you want to support. Notice no more than 100 devices are supported!

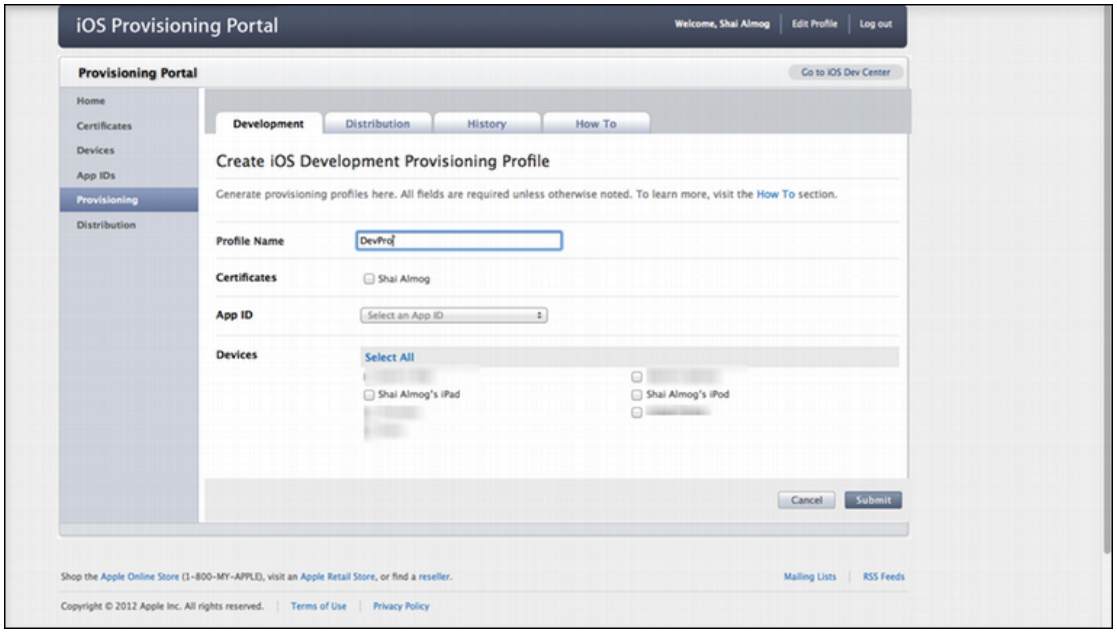


Create an application id, it should match the package identifier of your application perfectly!

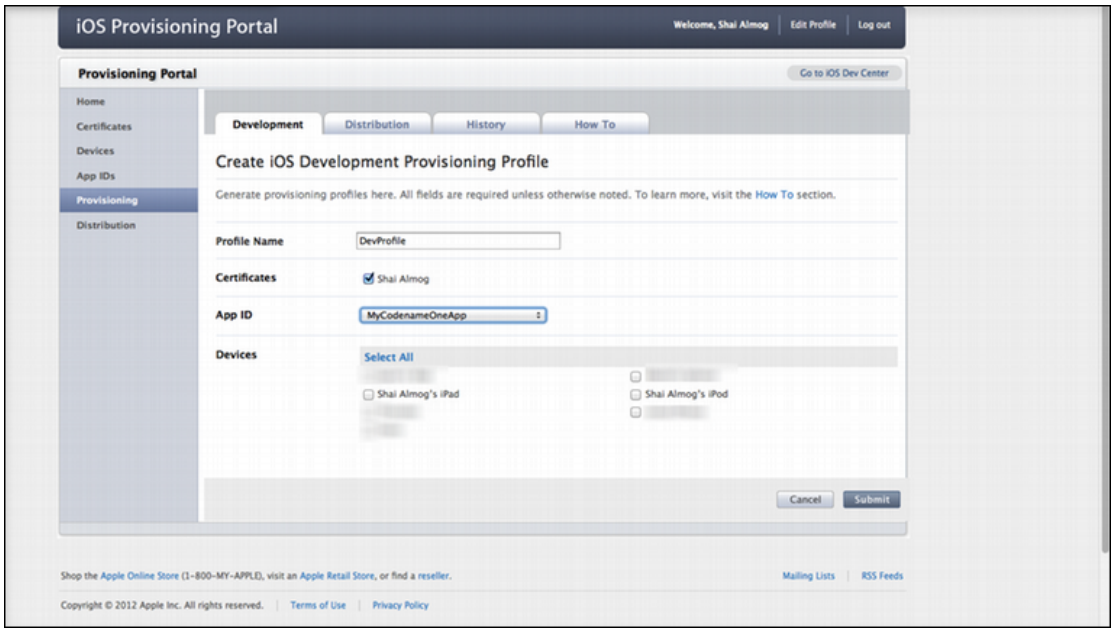


Create a provisioning profile for development, make sure to select the right app and make sure to add the devices you want to use during debug.

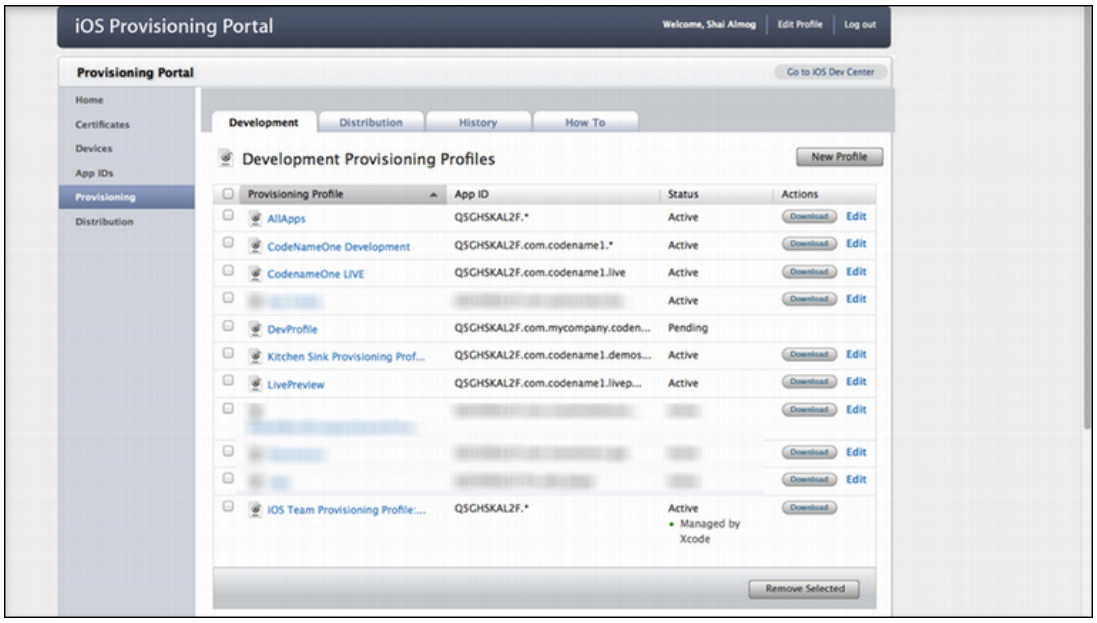




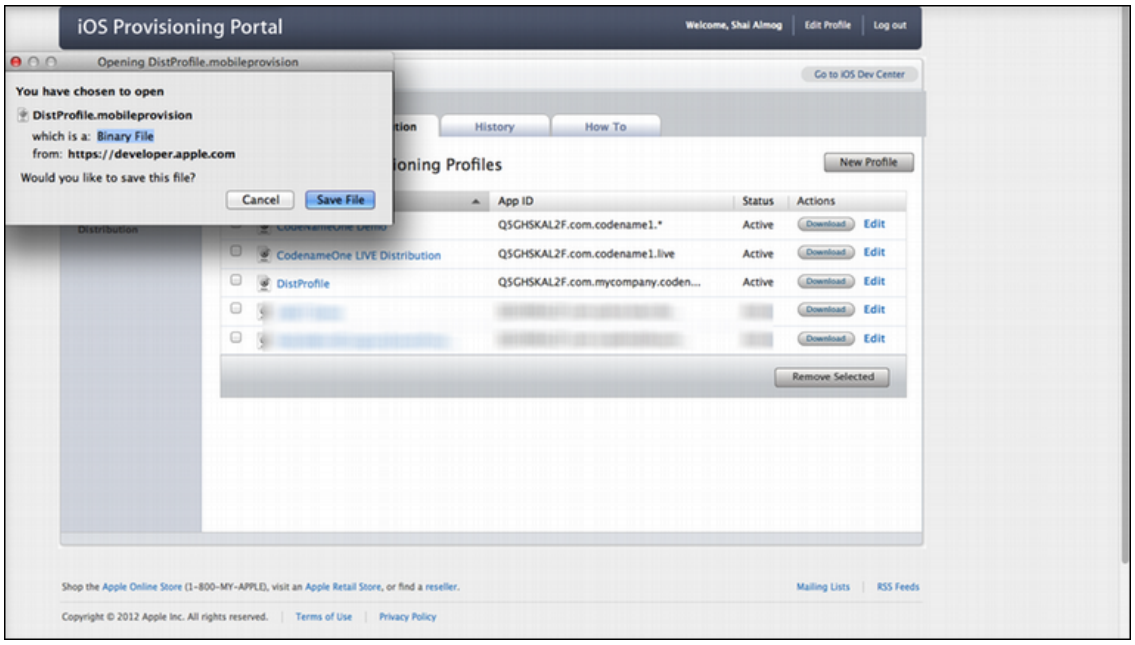
Refresh the screen to see the profile you just created and press the download button to download your development provisioning profile.



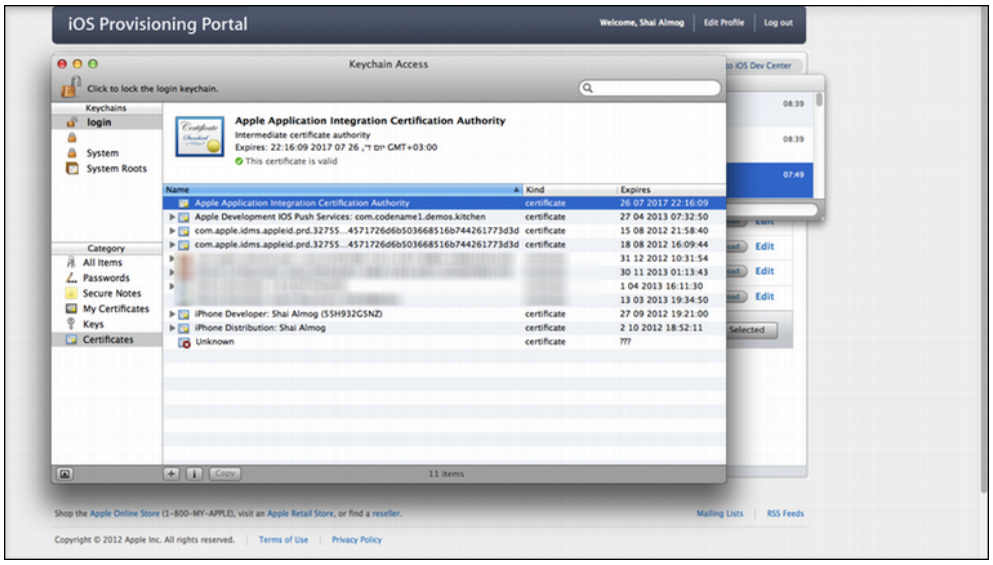
Create a distribution provisioning profile, it will be used when uploading to the app store. There is no need to specify devices here.



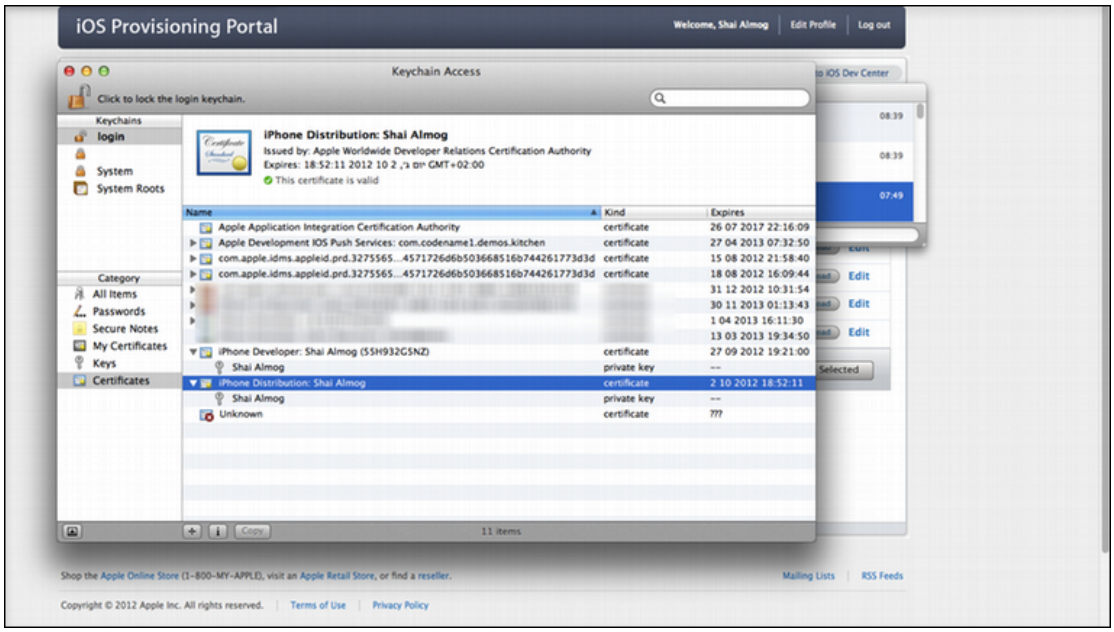
Download the distribution provisioning profile.



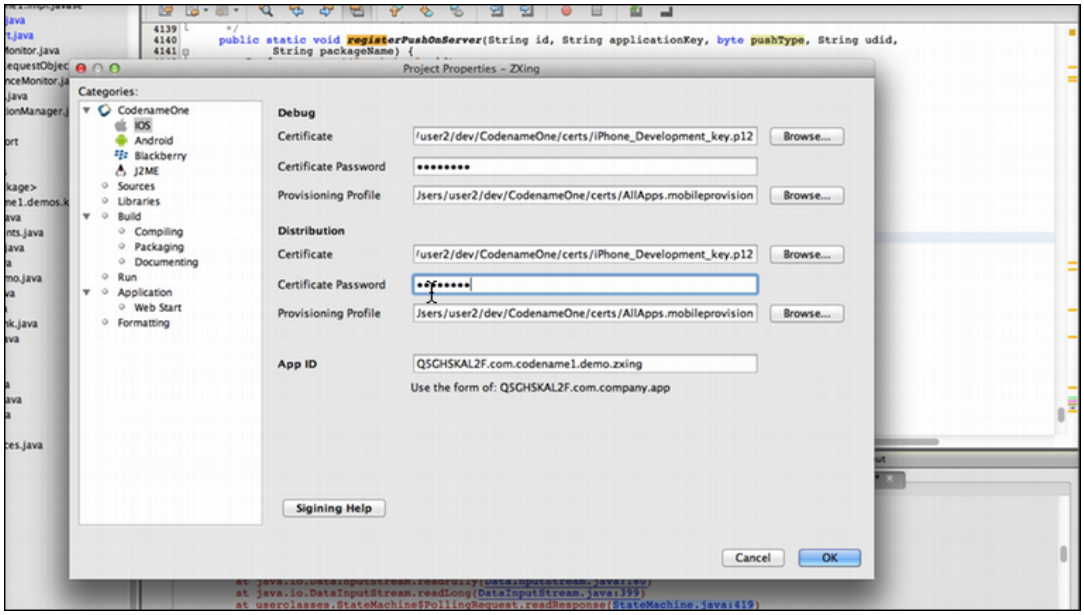
We can now import the cer files into the key chain tool on a Mac by double clicking the file, on Windows the process is slightly more elaborate



We can export the p12 files for the distribution and development profiles through the keychain tool



In the IDE we enter the project settings, configure our provisioning profile, the password we typed when exporting and the p12 certificates. It is now possible to send the build to the server.



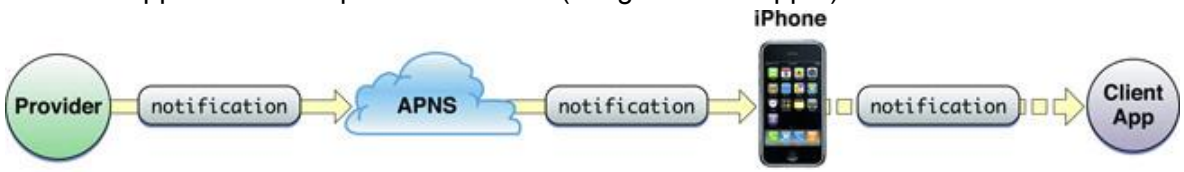
Push Notifications

Please notice that this code and implementation is highly experimental and we are still working out the kinks. It is also only enabled for paid accounts in Codename One due to some of the server side infrastructure we need to maintain to support this feature.

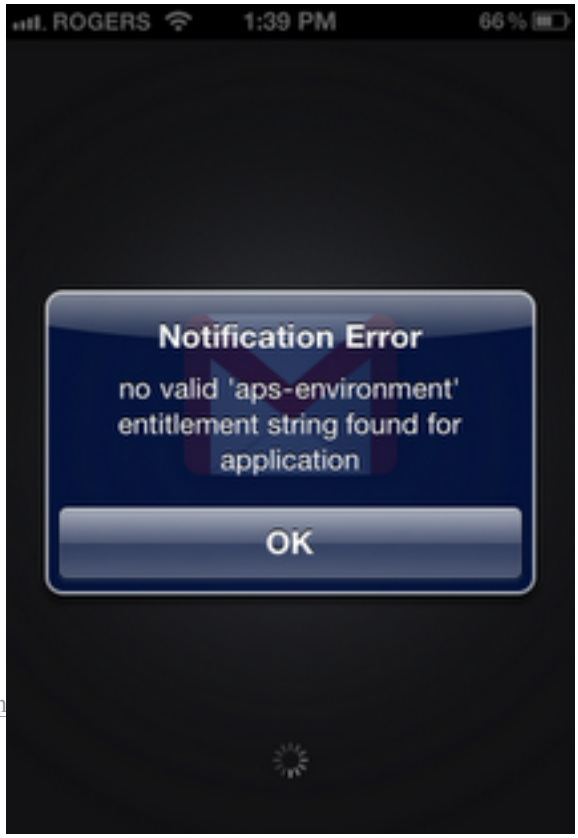
Push notification allows you to send a message to a device, usually a simple text message which is sent to the application or prompted to the user appropriately. When supported by the device it can be received even when the application isn't running and doesn't require polling which can drain device battery life.
The keyword here is "when supported" unfortunately not all devices support push notification e.g. Android device that don't have the Google Play application (formerly Android Market) don't support push and must fall back to polling the server... Not ideal.

Currently we support pushing to Google authorized Android devices: GCM (Google Cloud Messaging) and to iOS devices: Push Notification.
For other devices we will fallback to polling the server in a given frequency, not an ideal solution by any means so we give you the option to not fallback.

This is how Apple describes push notification (image source Apple):



The "provider" is the server code that wishes to notify the device. It needs to ask Apple to push to a specific device and a specific client application.
There are many complexities not mentioned here such as the need to have a push certificate or how the notification to APNS actually happens but the basic idea is identical in iOS and Android's GCM.



Codename One hides some but not all of the complexities involved in the push notification process. Instead of working between the differences of APNS/GCM & falling back to polling, we effectively do everything that's involved.

Push consists of the following stages on the client:

1. Local Registration - an application needs to register to ask for push. This is done by invoking:

2 Display.getInstance().registerPush("auth keyfor GCM", fallback);
3 The fallback flag indicates whether the system should fallback to polling if push isn't supported.
4 On iOS this stage prompts the user indicating to him that the application is interested in receiving push notification messages.
5 Remote registration - once registration in the client works, the device needs to register to the cloud. This is an important step since push requires a specific device registration key (think of it as a "phone number" for the device). Normally Codename One registers all devices that reach this stage so you can push a notification for everyone, however if you wish to push to a specific device you will need to catch this information! To get push events your main class (important, this must be your main class!) should implement the PushCallback interface. The registeredForPush(String) callback is called with the unique device ID that can be used later on when pushing to this particular device.
6 In case of an error during push registration you will receive the dreaded: pushRegistrationError.
7 This is a very problematic area on iOS, where you must have a package name that matches EXACTLY the options in your provisioning profile which is setup to support push. It is also critical that you do not use a provisioning profile containing a * character in it.
8 You will receive a push callback if all goes well.

Sending the push requires two different things for Google and iOS. In GCM (Google) you send a push via a key that is generated for you in the Android market you can read more about it here http://developer.android.com/google/gcm/gs.html.

For iOS developers must download a push certificate from the iOS provisioning portal for the given application. Notice that by enabling push you will probably need to regenerate the provisioning profile as well and download/replace your existing provisioning profile.
The certificate must be converted to a p12 certificate as I explained in the provisioning profile tutorial.

Currently we don't have the option to send the iOS certificate as part of the push request so you will have to host it somewhere and provide us with the link.

To send a push message use the following web API:
https://codename-one.appspot.com/sendPushMessage?auth=googleAuthKey&
packageName=packageNameOfApplication&
email=emailAccount&
device=theDeviceKey&
type=1&
production=true/false&

certPassword=pass&
cert=CertificateURL

Notice the following arguments in particular:

- device - this is an optional argument, to send to all devices just omit this argument. To send to a particular device you need to provide its key which was given in the registeredForPush() callback.
- type - this must currently be set to 1, there will be additional types in the future
- production - iOS has production and development push services each of which requires a different certificate. Its important to pick the right one. Pass true for production and false for development.
- cert - the URL for the iOS push certificate which you are hosting.
- certPassword - the password of the certificate

Cloud Object API On The Desktop

Using the cloud object API is remarkably powerful on the devices, however it becomes even more powerful when you can import data into the cloud server or communicate with it from a dedicated client (e.g. push data to the server).

To do so you can use the JavaSE.jar within any Java SE application make sure to invoke Display.init(new java.awt.Container()); before starting at which point you will be able to use all the standard methods of the cloud storage API to batch import or export data into/from the cloud storage.