

Codename One Developer Guide

Authors

Shai Almog, Chen Fishbein, Eric Coolman

Table of contents

Table of contents	2
Introduction	6
<i>History.....</i>	6
<i>How Does It Work.....</i>	6
Limitations & Capabilities.....	7
<i>Lightweight UI.....</i>	8
Installation	8
<i>Installing Codename One In NetBeans</i>	8
<i>Installing Codename One In Eclipse.....</i>	10
GUI Builder Hello World.....	11
Manual Hello World	15
Basics: Themes, Styles, Components & Layouts	17
What Is A Theme, What Is A Style & What Is a Component?.....	17
Creating A Native Theme	17
Component/Container Hierarchy	20
Layout Managers	21
<i>Flow Layout.....</i>	22
<i>Box Layout</i>	23
<i>Border Layout.....</i>	23
<i>Grid Layout.....</i>	24
<i>Table Layout</i>	24
<i>Layered Layout</i>	25
<i>Understanding Preferred Size</i>	26
<i>Layout Reflow.....</i>	26
Layout Animations.....	27
Building Your Own Layout Manager	29
Porting a Swing/AWT Layout Manager.....	31
Theme Basics.....	32
Advanced Theming.....	38
Understanding Codename One Themes	38
Working With UIID	38
Style Inheritance.....	39
Colors & Transparency	40
Backgrounds.....	40
Fonts	41
Borders	42
Padding/Margin.....	43
Theme Constants	44
How Does A Theme Work.....	50
Understanding Images & Multi-Images	51
Working With The GUI Builder.....	54
Basic Concepts	54
The Components Of Codename One	55

Container.....	55
Composite Components.....	55
Form.....	55
Dialog.....	57
Label.....	57
Button.....	58
CheckBox/Radio Button.....	58
Multi-Button	58
Span Button.....	59
Span Label	59
On Off Switch.....	59
TextField/TextArea.....	59
Toggle Button.....	59
List, ContainerList, Renderers & Models.....	61
<i>Important - Lists & Layout Managers</i>	62
<i>Using Lists In The GUI Builder</i>	62
<i>Understanding MVC</i>	67
<i>List Cell Renderer</i>	68
<i>Generic List Cell Renderer</i>	72
<i>The List Model</i>	75
MultiList	79
Slider.....	79
Table	80
Tree	81
Share Button.....	82
Infinite Progress	83
Tabs.....	84
MediaPlayer	84
ImageViewer	85
WebBrowser.....	85
Auto Complete	86
Spinner & Picker	87
Embedded Container	87
The Map Component.....	88
Animations & Transitions	94
Low Level Animations.....	94
Transitions	95
The EDT - Event Dispatch Thread	100
What Is The EDT	100
Debugging EDT Violations.....	100
Call Serially (And Wait)	101
Invoke And Block.....	102
Monetization.....	105
Ad Networks	105
<i>vserv</i>	105
<i>Inneractive</i>	106
Google Play Ads.....	106

In App Purchase	107
Graphics, Drawing, Images & Fonts	110
Basics - Where & How Do I Draw Manually?	110
Images	111
Understanding Encoded Images & Image Locking	114
Glass Pane	114
File System, Storage, Network & Parsing	117
Externalizable Objects	117
Storage vs. File System	119
Storage	119
File System	119
Cloud Storage	120
Cloud File Storage	124
SQL	124
Network Manager & Connection Request	125
Debugging Network Connections	126
Network Services	126
UI Bindings & Utilities	127
Logging & Crash Protection	127
Parsing: JSON, XML & CSV	128
Cached Data Service	129
GZIP	130
Miscellaneous Features	132
SMS, Dial (Phone) & E-Mail	132
Contacts API	132
Localization & Internationalization (L10N & I18N)	133
<i>Localization Manager</i>	135
<i>RTL/Bidi</i>	135
Location - GPS	136
Capture - Photos, Video, Audio	137
Codescan - Barcode & QR code scanner	137
Analytics Integration	139
Facebook Support (legacy)	139
SideMenuBar - Hamburger Sidemenu	143
Pull To Refresh	148
Infinite Scroll Adapter	148
Performance, Size & Debugging	150
Reducing Resource File Size	150
Improving Performance	151
Performance Monitor	151
Network Speed	151
Debugging Codename One Sources	152
Device Testing Framework/Unit Testing	153
EDT Error Handler and sendLog	153
Advanced Topics/Under The Hood	155
Sending Arguments To The Build Server	155

The Architecture Of The GUI Builder	158
<i>Basic Concepts</i>	158
<i>IDE Bindings</i>	159
Native Interfaces	162
<i>Native Permissions</i>	165
Libraries - cn1lib	165
Drag & Drop	166
Physics - The Motion Class	167
Signing, Certificates & Provisioning	168
iOS (iPhone/iPad)	168
<i>iOS Code Signing Fail Checklist</i>	169
Android	173
RIM/BlackBerry	174
J2ME	174
Appendix: Working With iOS	175
Provisioning Profile & Certificates	175
Push Notifications	182
Appendix: Creating Codename One Maker Plugins	189
Appendix: Cloud Object API On The Desktop	195
Appendix: Casual Game Programming	196
The Game	197
Getting Started	197
Handling Multiple Device Resolutions	197
Resources	198
The Splash Screen	199
The Game UI	200
Summary	201

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 LWUIT¹ 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.

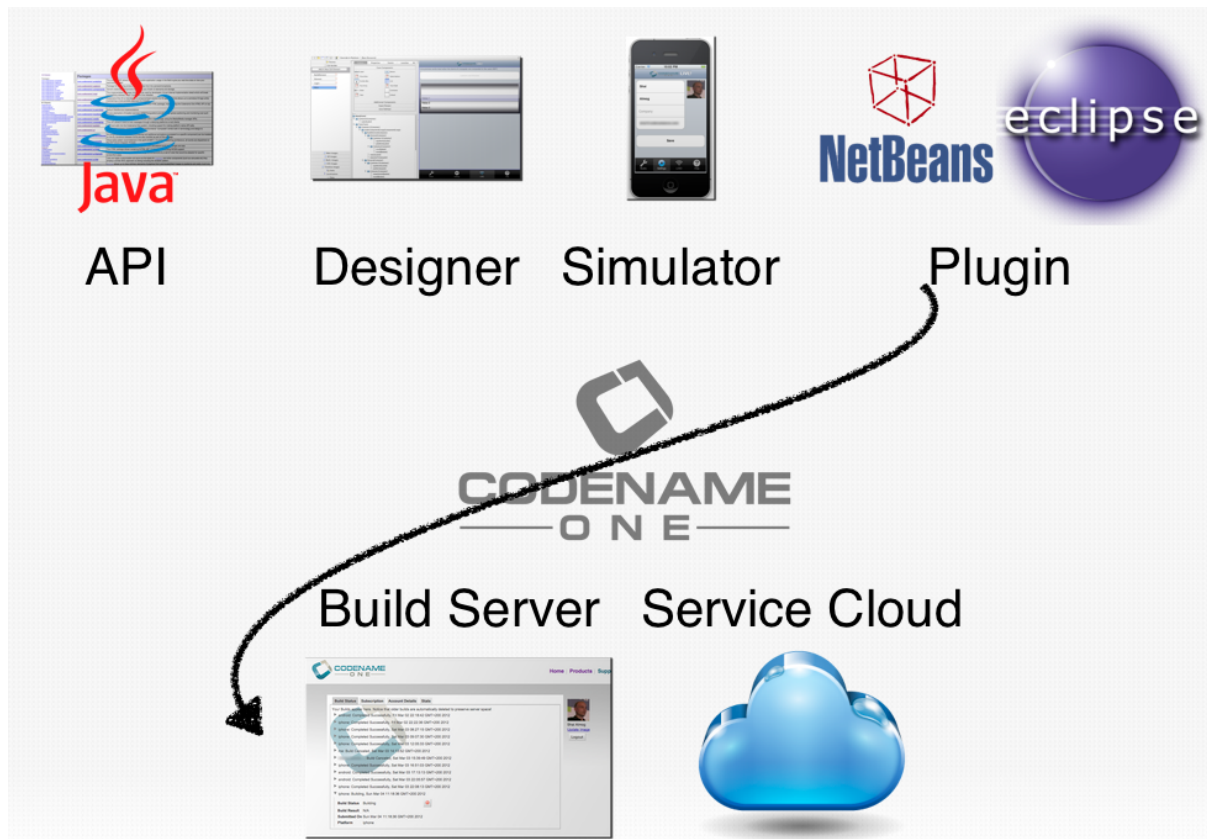


How Does It Work

Codename One has 4 major parts: API, Designer, Simulator and 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 that 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.

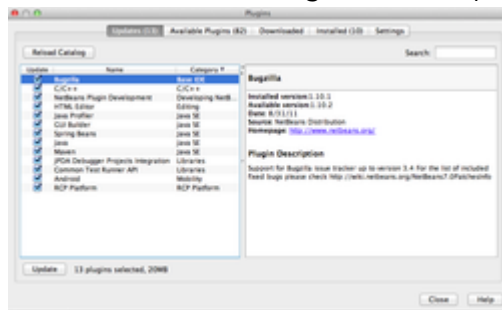
Installation

Installing Codename One In NetBeans

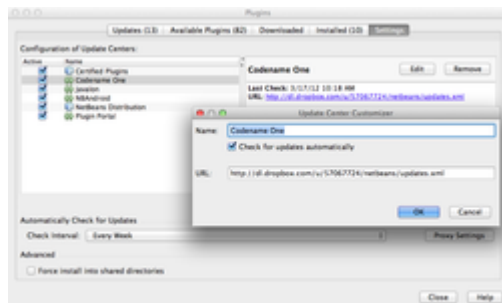
For the purpose of this document we will focus mostly on the NetBeans IDE for development, however most operations are almost identical in the Eclipse IDE as well. For instructions specific for Eclipse please go to the following section.

These instructions assume you have downloaded [NetBeans](#) 7.x, installed and launched it.

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



Click the "Add" button & enter the details below

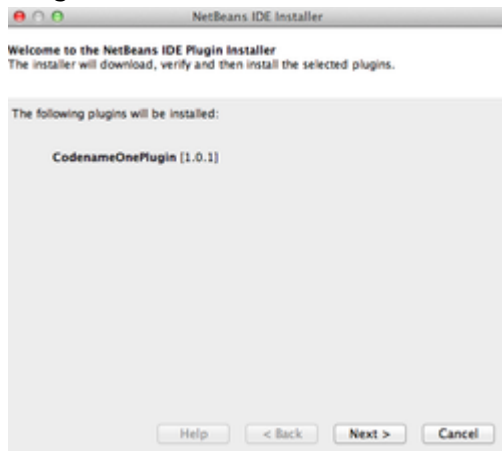


For the name enter: Codename One

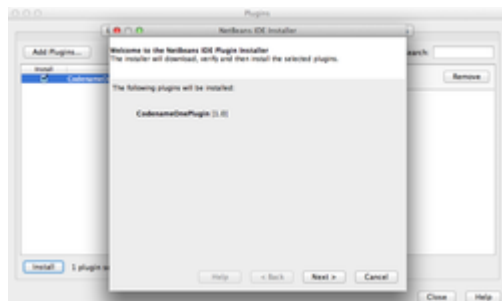
For the URL enter:

<https://codenameone.googlecode.com/svn/trunk/CodenameOne/repo/netbeans/updates.xml>

In The Available Plugins Tab Click "Reload Catalog" Then Check The CodenameOne Plugin



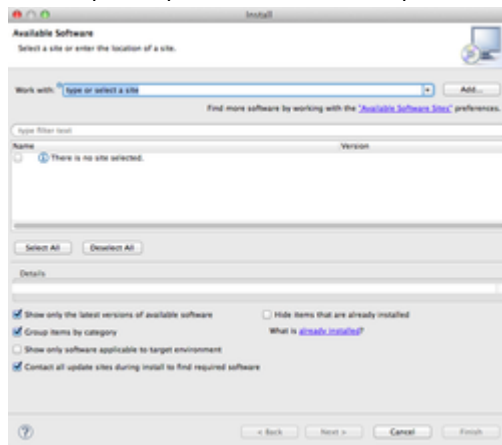
After that click the install button below. Follow the Wizard instructions to install the plugin



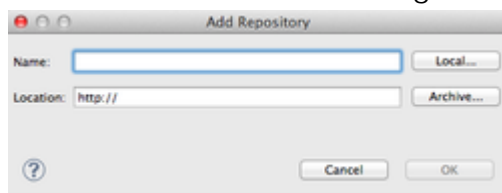
You will be informed that the plugin is unsigned which is indeed true, you just continue anyway.

Installing Codename One In Eclipse

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



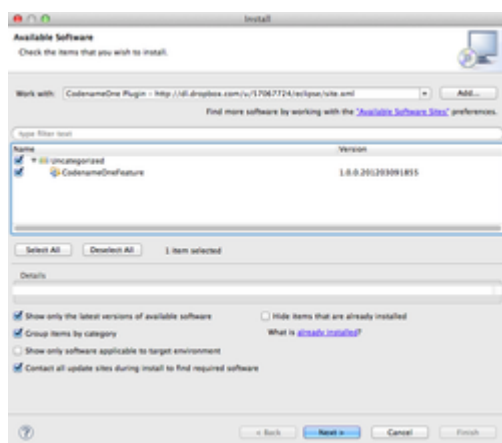
Click the Add button on the right side & fill out the entries



Enter Codename One for the name and

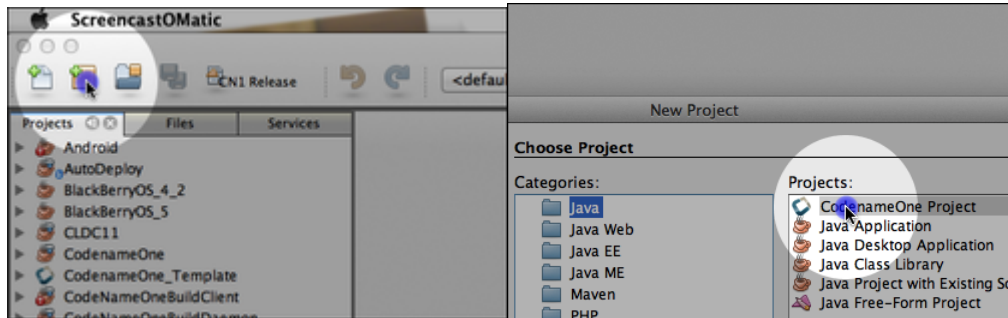
<https://codenameone.googlecode.com/svn/trunk/CodenameOne/repo/eclipse/site.xml> for the location.

Select the entries & follow the wizard to install

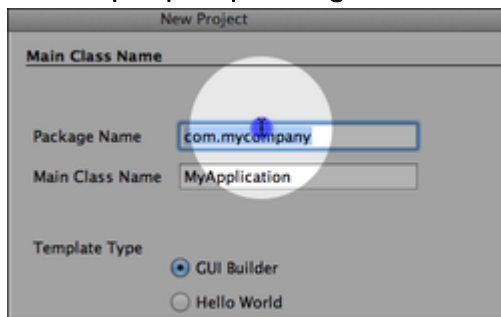


GUI Builder Hello World

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

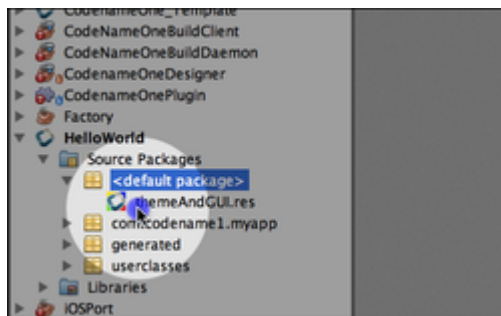


Use a proper package name for the project!

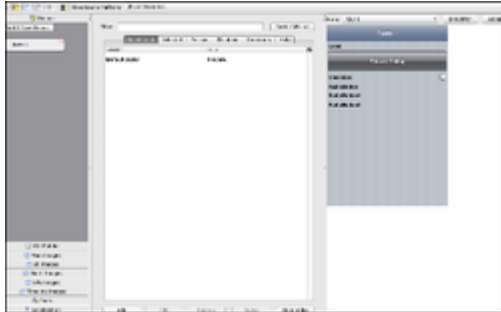


It is important to use proper package names for the project since platforms such as iOS & Android rely on these names for future updates. They are thus painful to change! The convention is the common Java convention of reverse domain names (e.g. com.codenameone for the owner of codenameone.com etc.).

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

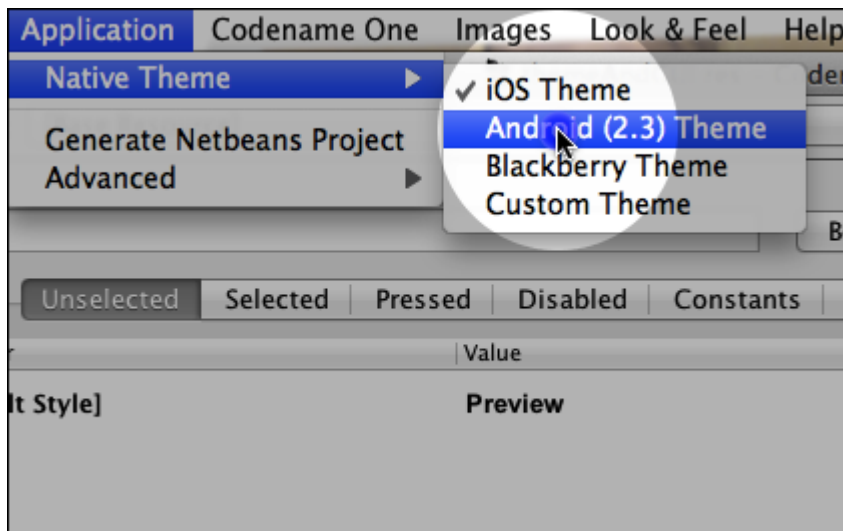


Since this is a GUI builder project you can open the Codename One Designer with the content of the theme.res file by double clicking the file. FYI: If you choose to rename this file in the future it is important to update the resource file name in the project properties settings.

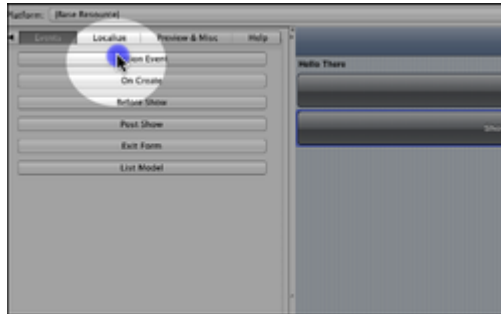


Within the Codename One Designer we can see several categories the most interesting of which are theme & GUI. We will start by opening the theme section and clicking the entry there.

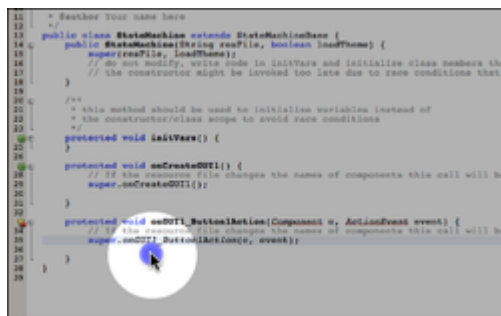
The default Codename One theme, derives from the platform native theme. We can easily change the "base theme" in the preview to see how the theme will act in different devices by clicking the native theme option in the application menu as such.



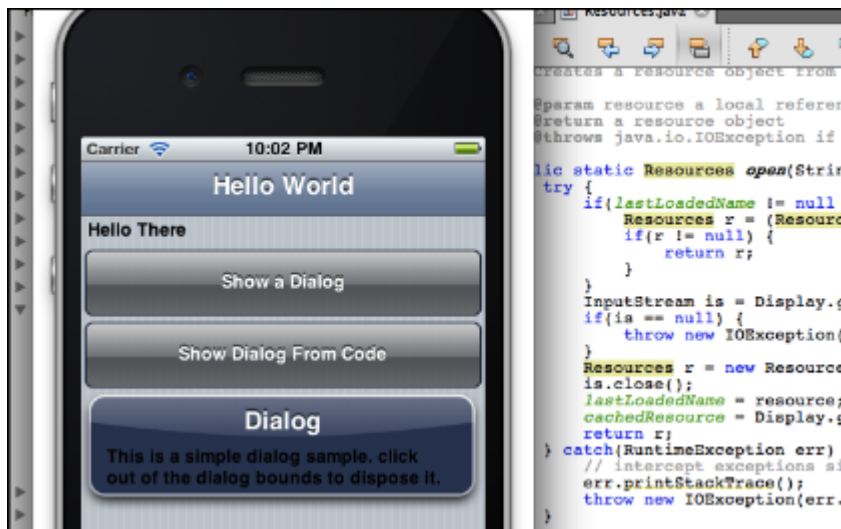
To edit the UI for the application we need to select the GUI section and click "GUI 1" within that section. We are then faced with a drag and drop interface to manipulate the GUI of the application we saw within the theme preview. All changes made here are reflected instantly to the theme preview.



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 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);`



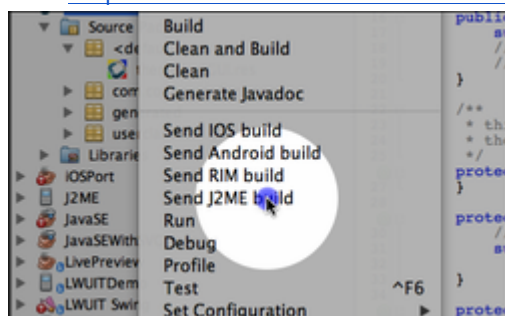
If we inspect the Codename One project to some degree we will notice that it is pretty close to a standard Java project hence it can be modified/used in very similar ways. You

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 sign up 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.

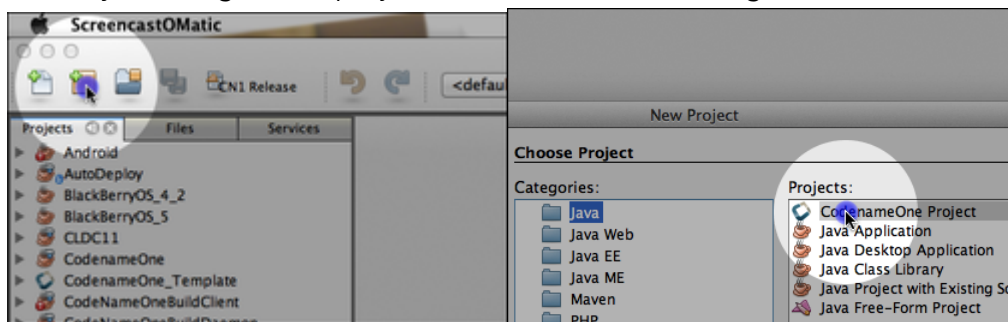


In the build server we can follow the progress of the build and the results for the current build. We can email a link to the deployment files, download them or use a QR reader to install them to the phone. The Codename One LIVE! application allows following the status and installing applications directly from the build server.

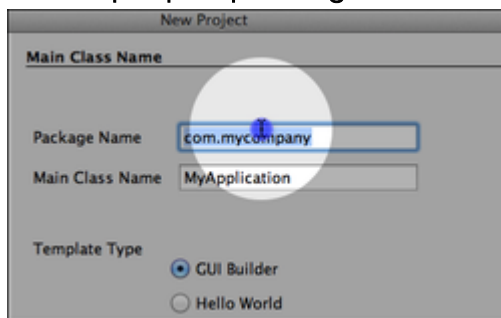
Manual Hello World

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 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.

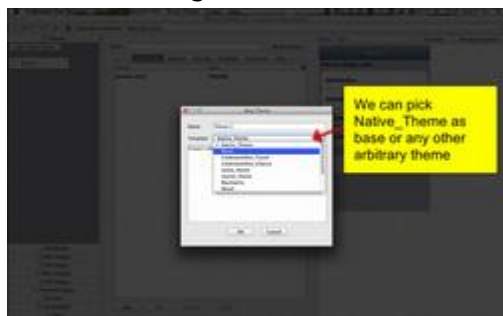
Several style objects determine the appearance of the component, every component has 4 style objects associated with it: Selected, Unselected, Disabled & Pressed.

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 and 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



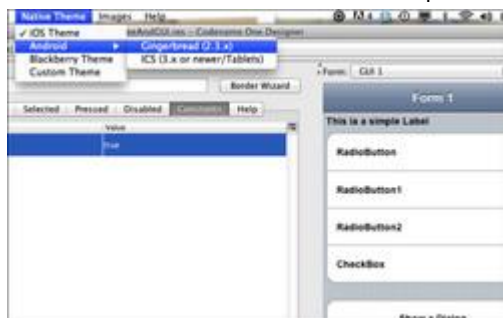
You can easily create a theme with any look you desire or you can "inherit" the platform native theme and start from that point. When adding a new theme you are given the option.

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 UUID of a label and created a style for the new UUID. 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



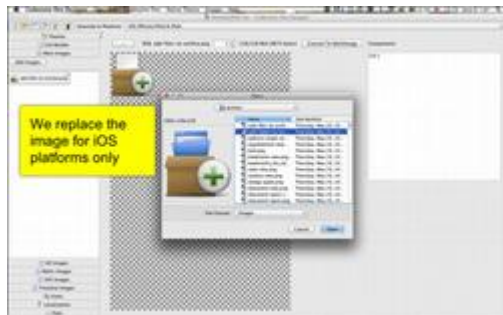
A common case we run into when adapting for platform specific looks is that a specific resource should be different to match the platform conventions. The Override feature allows us to define resources that are specific to a given platform combination. Override resources take precedence over embedded 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



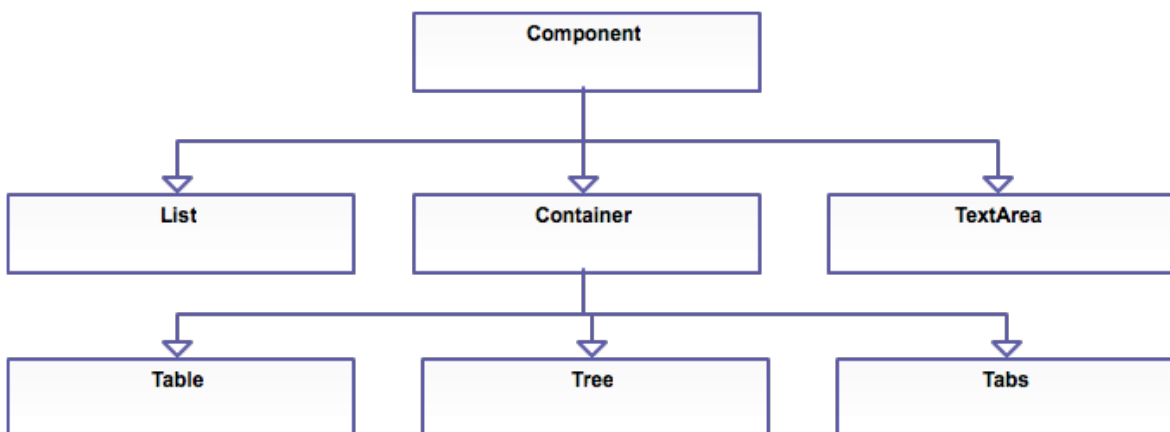
Then click the green checkbox to define that this resource is specific to this platform. All resources added at this point will only apply to the given platform. If you change your mind and are no longer interested in a particular override just delete it in the override mode and it will no longer be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

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

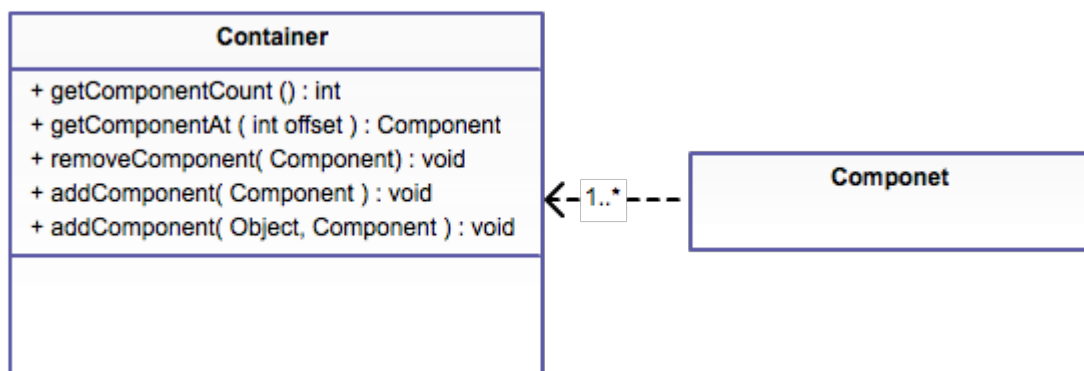


Selecting the “...” button easily does this. We can easily do the same in the GUI builder although this is a dangerous road to start following since it might end up with a great deal of fragmentation.

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, it's 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.

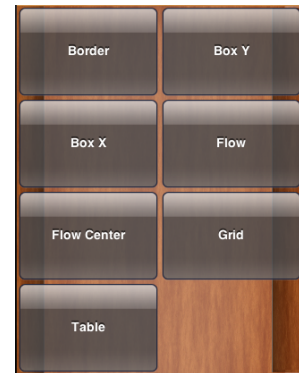


Table Layout

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 since it will trigger 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.

E.g. this code will place a green “online” icon at the bottom right position on top of an icon for instance to indicate that a friend is currently online:

```
Label pictureOfFriend = new Label(..);
Label onlineGreenImage = new Label(greenImage);
Container layered = new Container(new LayeredLayout());
layered.addComponent(pictureOfFriend);

// assuming you want bottom right position
Layout flow = new FlowLayout(Component.RIGHT);
flow.setValign(Component.BOTTOM);
Container bottomRight = new Container(flow);
bottomRight.addComponent(onlineGreenImage);
layered.addComponent(bottomRight);
```

Form’s have a built in layered layout that you can access via `getLayeredPane()`, this will allow you to overlay elements on top of the content pane.

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 that 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 is 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 in which the components slide from separate directions into the screen. This is the code we 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

Codename One LayoutManagers are a remarkably powerful tool. We 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:

```
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();
```

Porting a Swing/AWT Layout Manager

We recently ported GridBagLayout to Codename One and while that was pretty easy considering the complexity of that layout there are still some tips you should take into account when evaluating this:

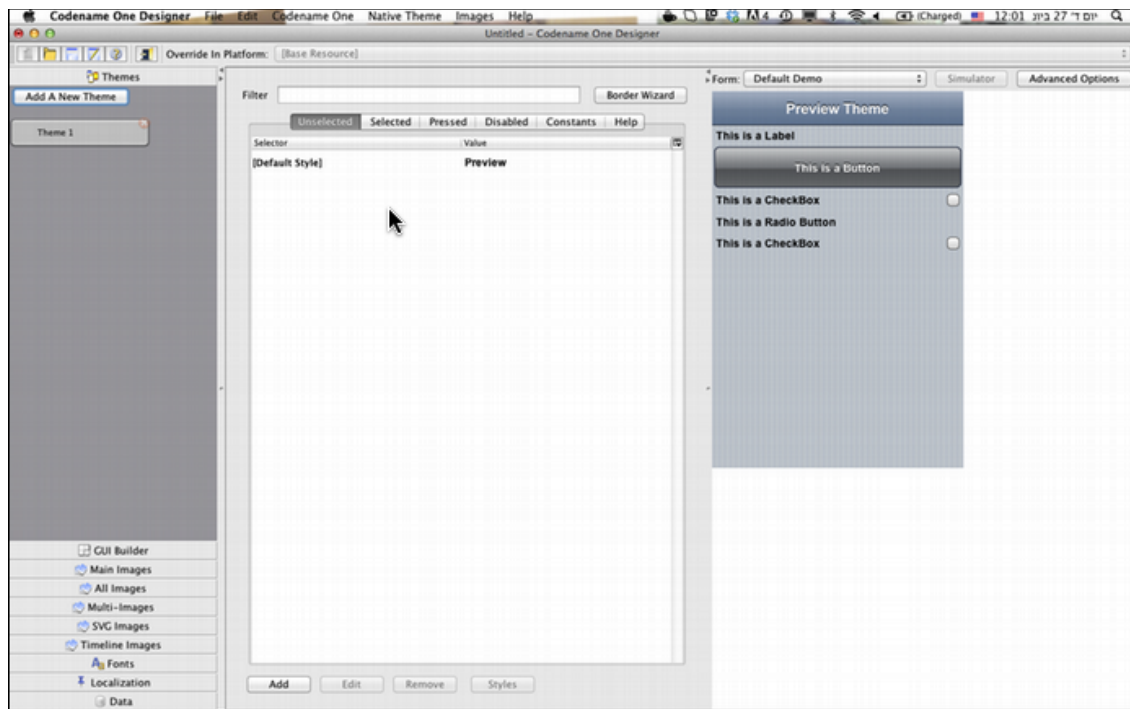
1. Codename One doesn't have Insets, we added some support for them in order to port gridbag but components in Codename One have a Margin they need to consider instead of the insets (the padding is in the preferred size).
2. AWT layout managers also synchronize a lot on the AWT thread. This is no longer necessary since Codename One is single threaded like Swing.
3. Components are positioned relatively to container so the layout code can start at 0, 0 (otherwise it will be slightly offset).

Other than those things it's mostly just fixing method signatures and import statements, which are slightly different. Pretty trivial stuff and GridBagLayout from project Harmony is now working on Codename One.

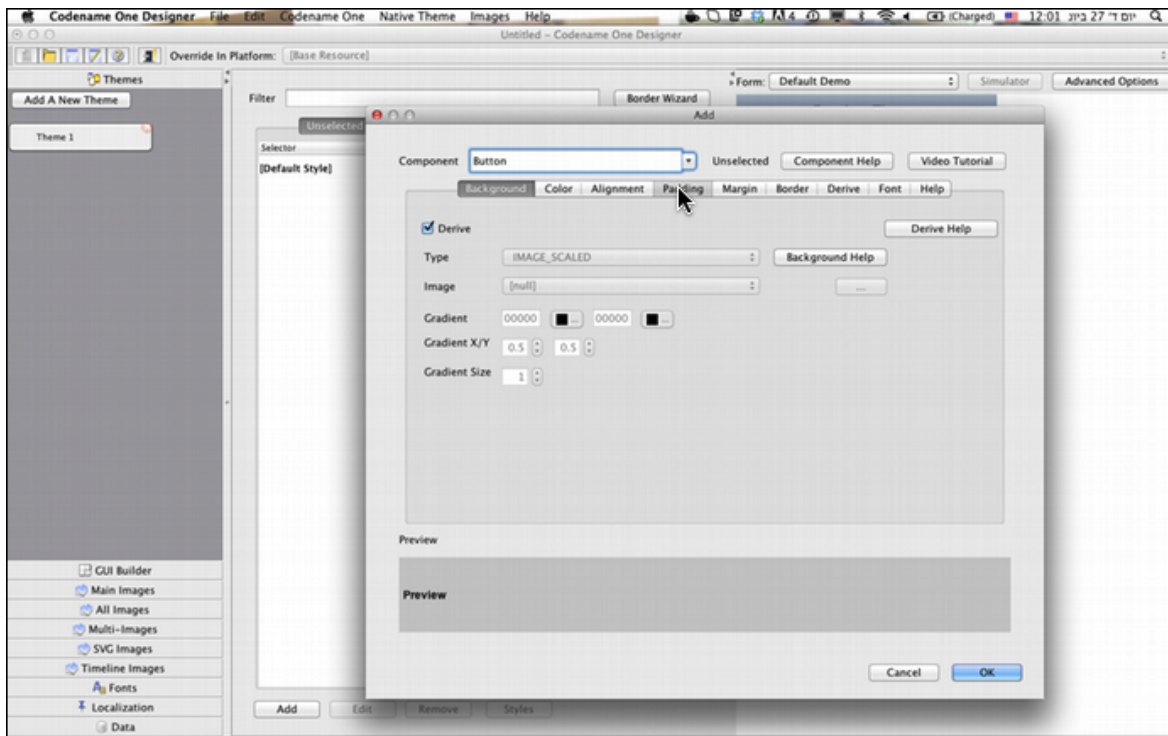
Theme Basics

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.



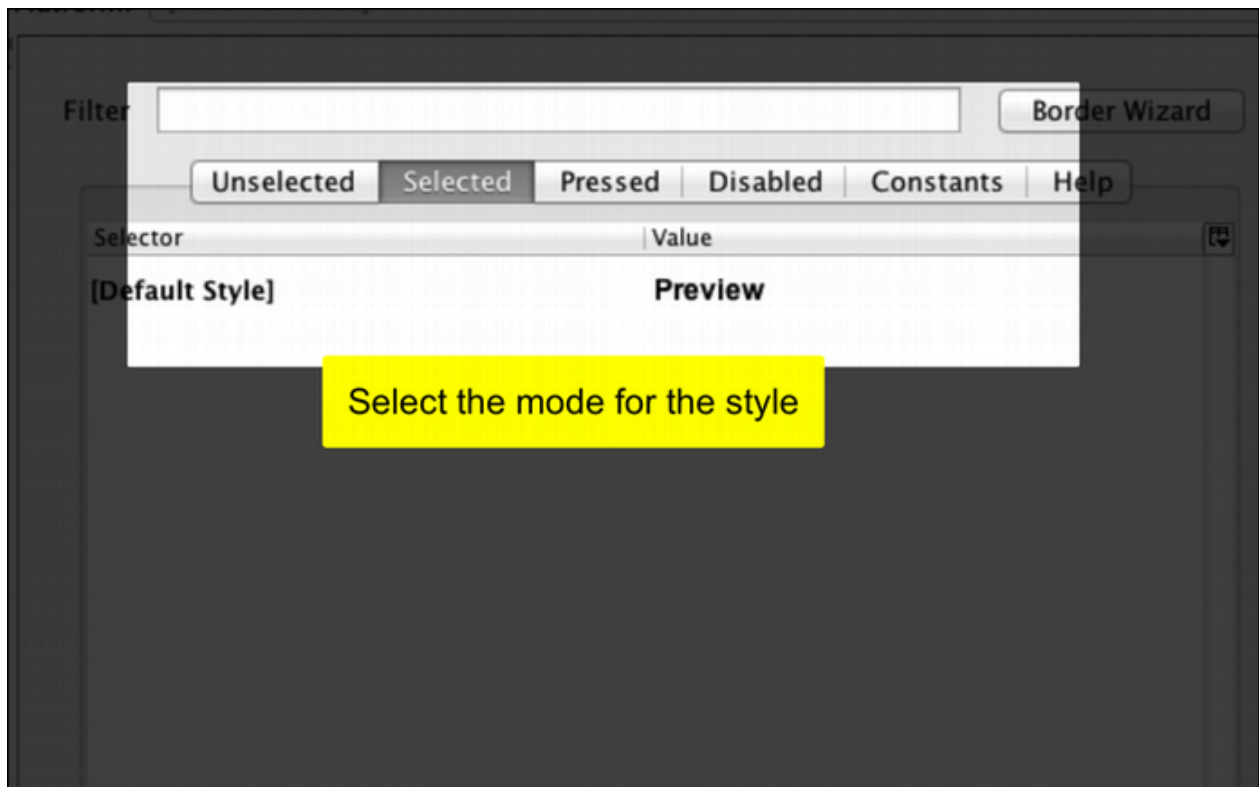
We can add a component style to a component such as Button; typically UIID's are named with the same name as the Component class. You can modify the UIID of a component by invoking `setUIID(String)` on an arbitrary component or changing the UIID property in the GUI builder.



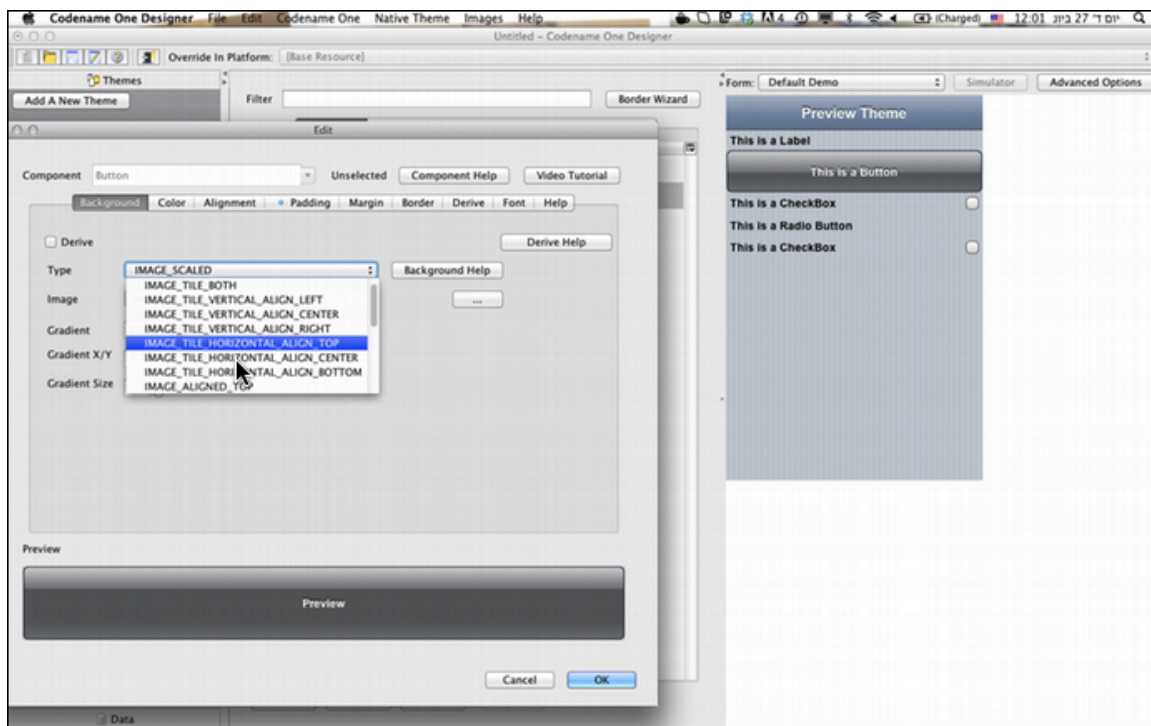
Styles can have one of 4 states:

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.

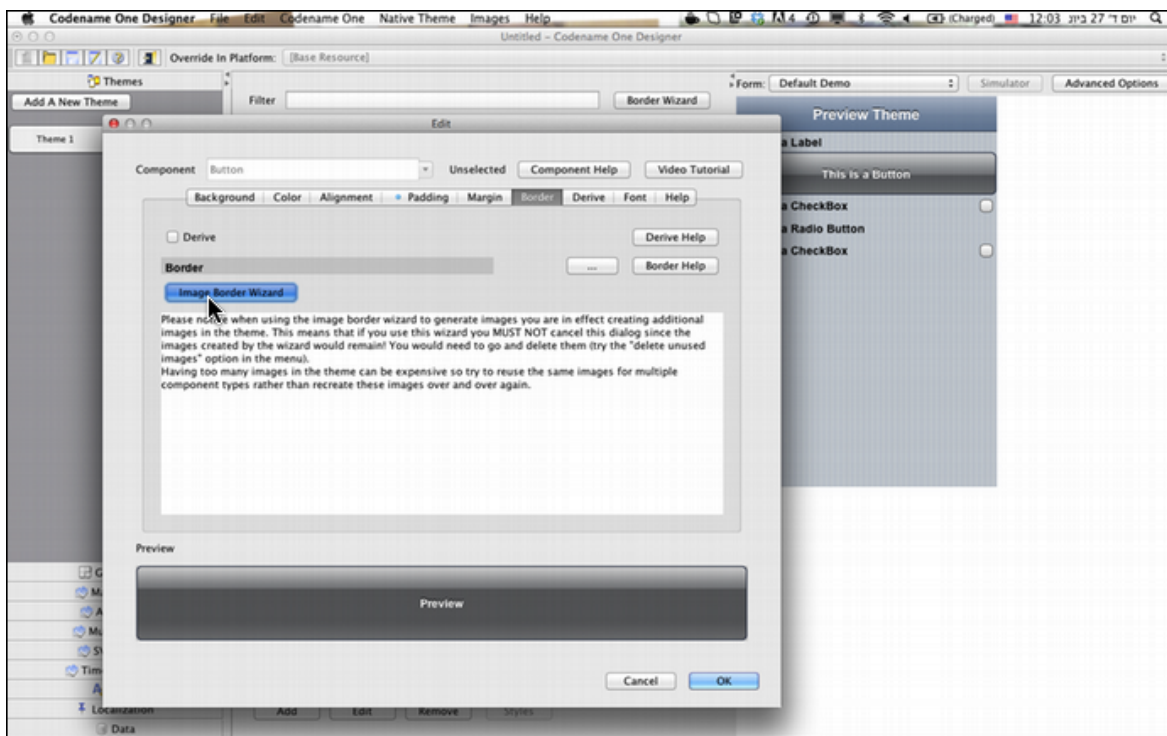
You can add a style to any one of the states in the Designer to make the component appear as expected in those cases.



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.

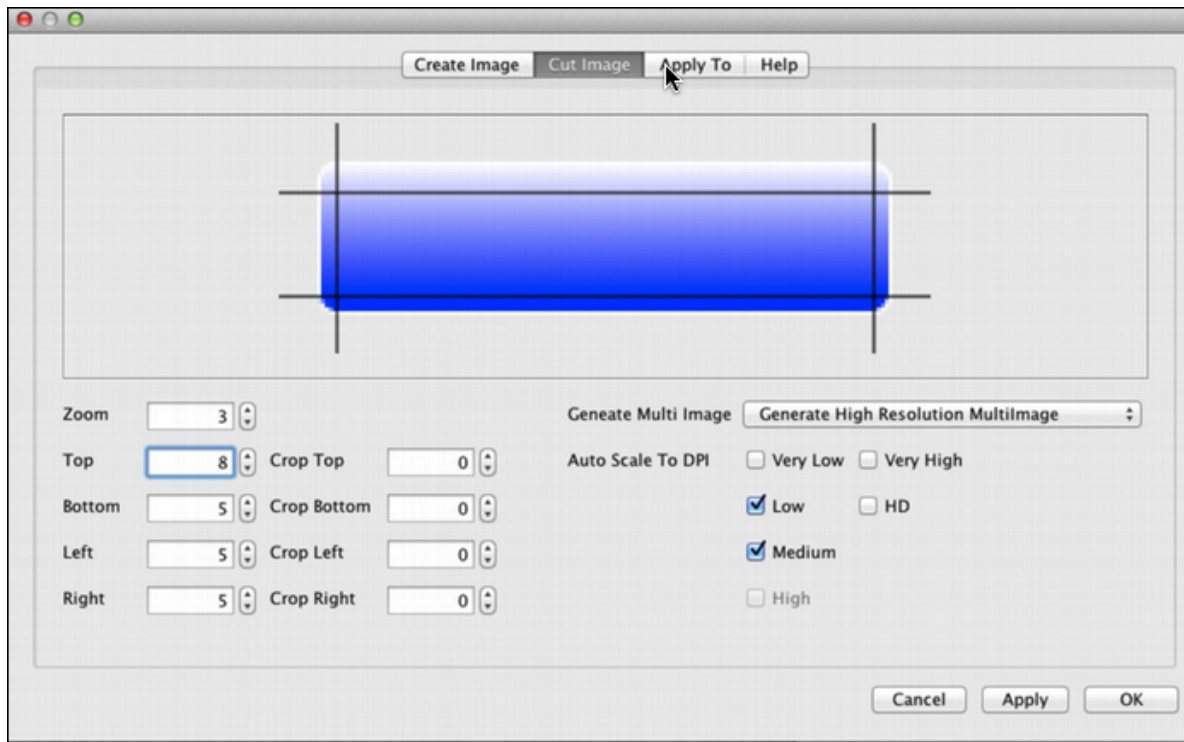


Border's are a remarkably powerful tool for customizing the appearance of a Component. The most powerful approach is the 9-piece image border, which is easiest to use when using the Image Border Wizard.

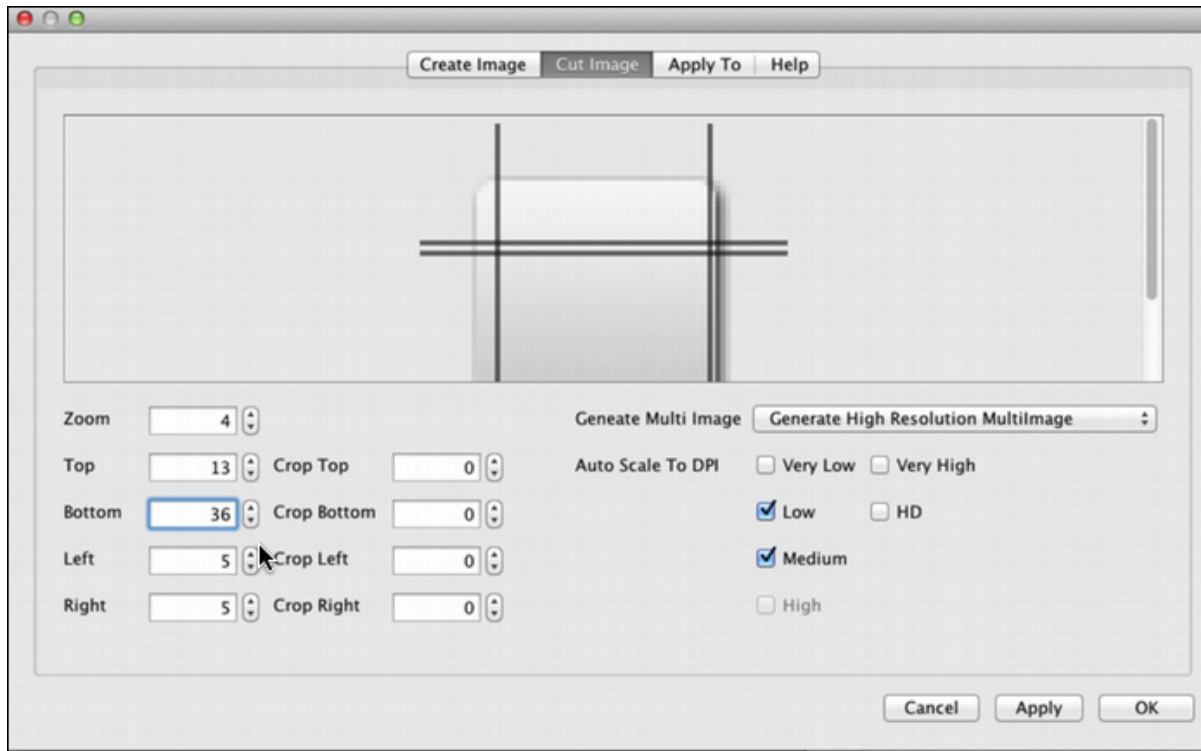


The image border wizard allows you to take an image and "cut it" into 9 distinct pieces: 4 corners, top, bottom, left, right & center.

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 section (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 LookAndFeel/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 UUID

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

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

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

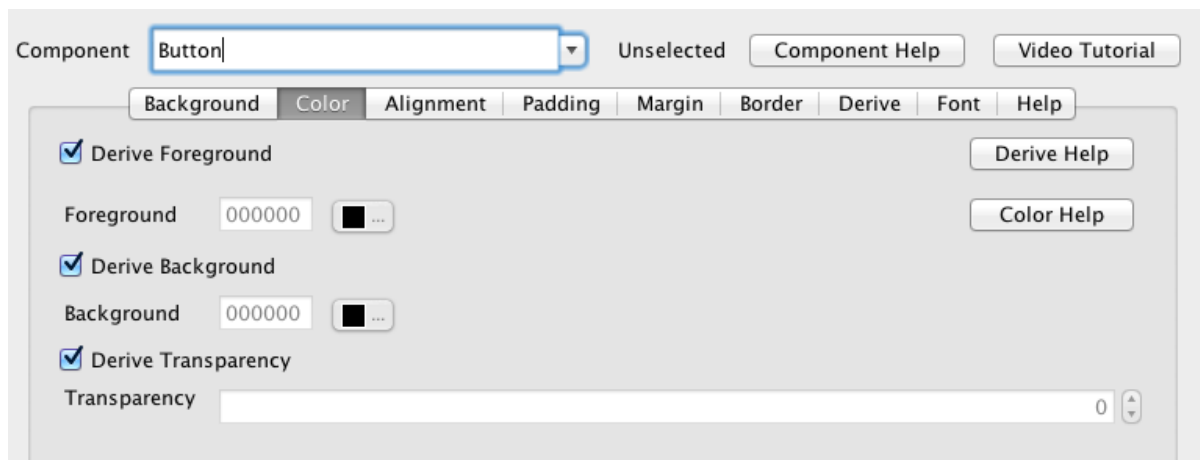
UUID can be customized via the GUI builder and allows for powerful selection of individual components.

Style Inheritance

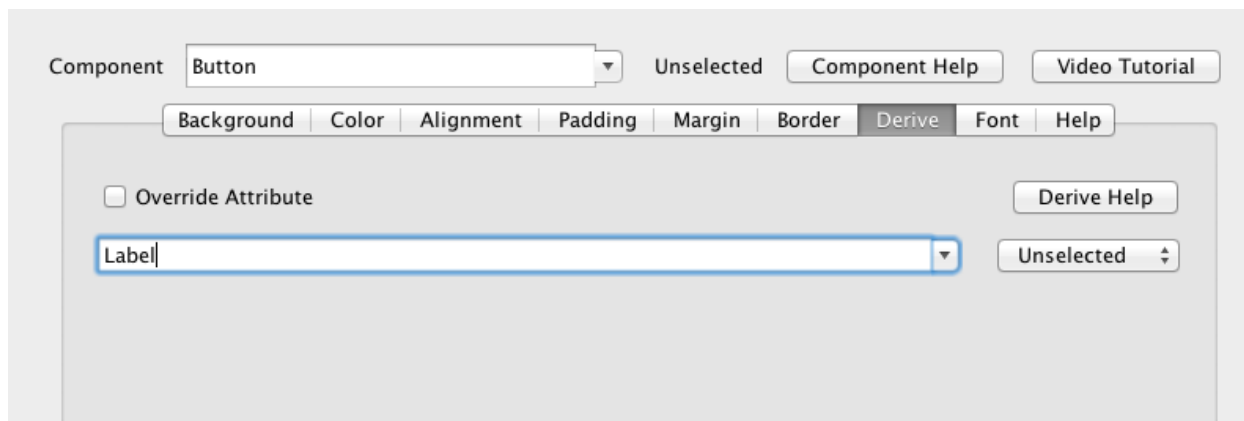
There are multiple defaults defined for elements within Codename One, e.g. a Container defaults to zero padding/margin and full transparency.

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.

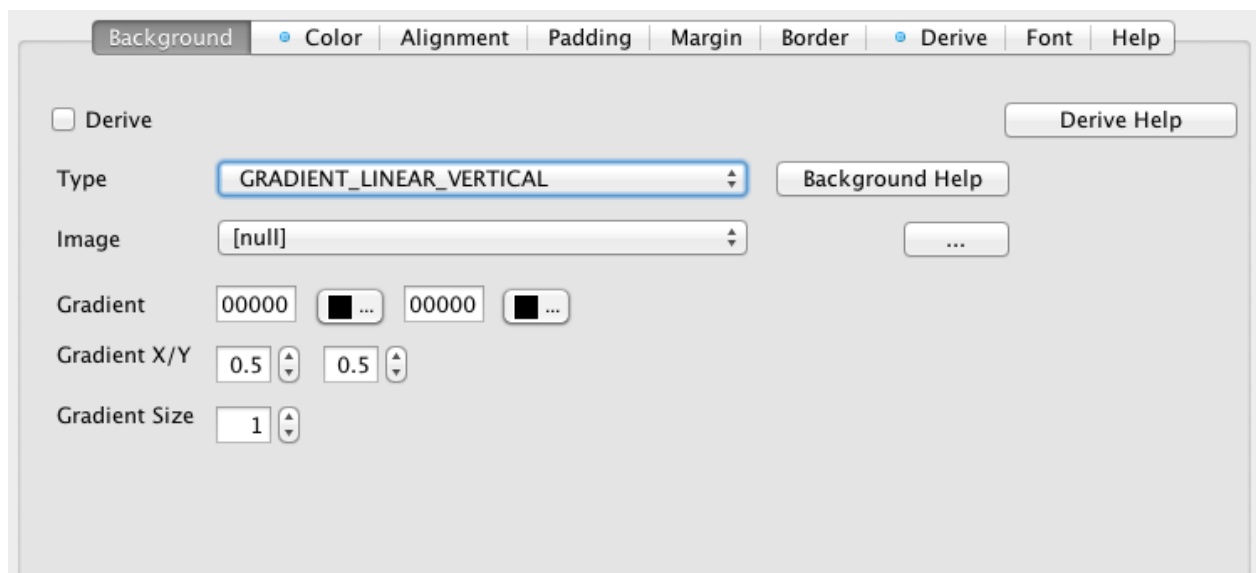
In the case of gradients you need to define all the gradient variables mentioned within the screen to define the source/destination colors and in the case of a radial the properties relevant to that.

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

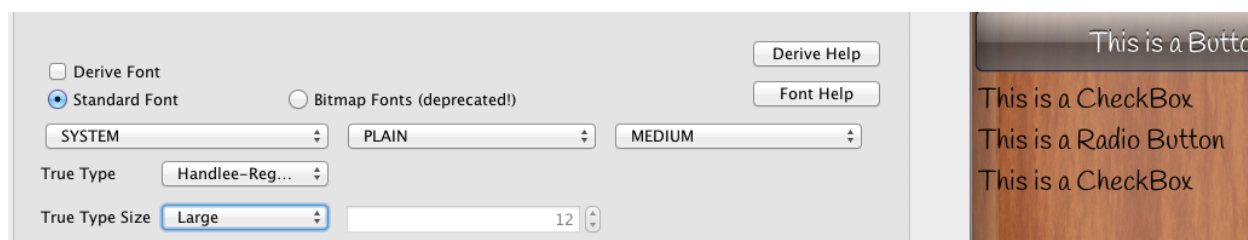
An image can be scaled across the background of the component; this can create a decent effect in some cases but often causes some distortion. You can use scale to fit/fill both of which create a more compelling scale effect that preserves aspect ratio.

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 types² system fonts and truetype fonts (TTF). 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

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.

developer has a selection from one of 3 generic sizes small, medium or large and basic style choices: bold/plain/italic.

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.

Notice that the file must have the ".ttf" extension otherwise the build server won't be able to recognize the file as a font and set it up accordingly (devices need fonts to be defined in very specific ways). Once you do that you can use the font from code or from the theme.

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.

To use fonts from code just use:

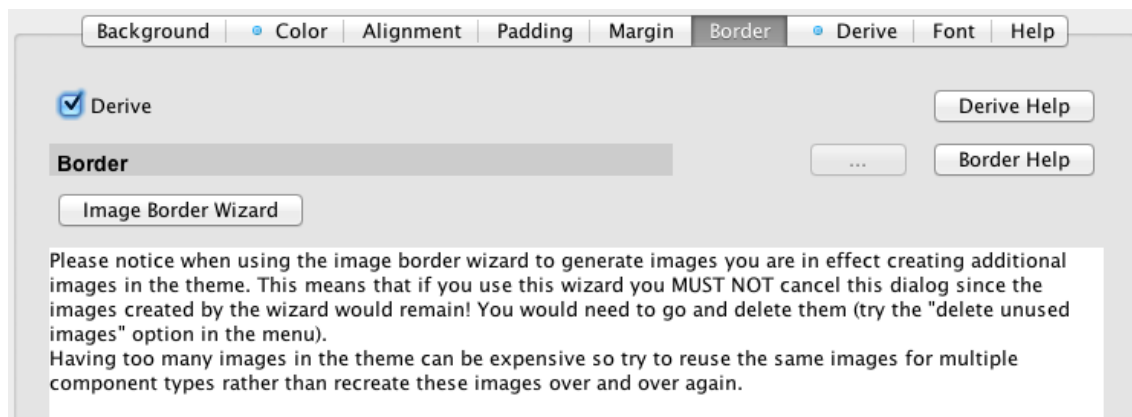
```
if(Font.isTrueTypeFileSupported()) {  
    Font myFont = Font.createTrueTypeFont(fontName, fontFileName);  
    myFont = myFont.derive(sizeInPixels, Font.STYLE_PLAIN);  
    // do something with the font  
}
```

Notice that in code only pixel sizes are supported so it's up to you to decide how to convert that. You also need to derive the font with the proper size unless you want a 0 sized font which probably isn't very useful.

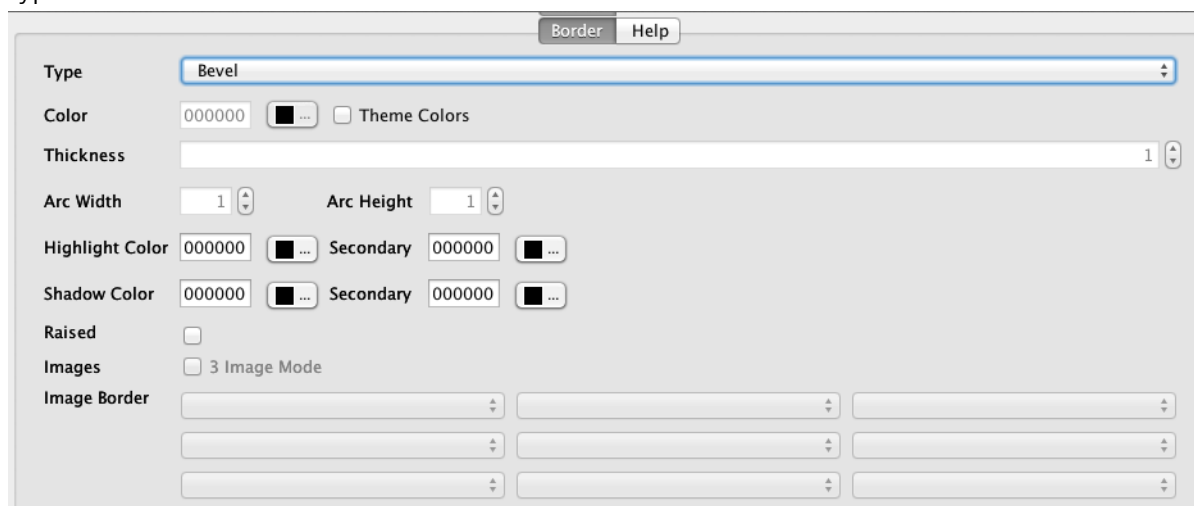
The font name is the difficult bit, iOS requires the name of the font which doesn't always correlate to the file name in order to load the font, its sometimes viewable within a font viewer but isn't always intuitive so be sure to test that on the device to make sure you got it right.

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:



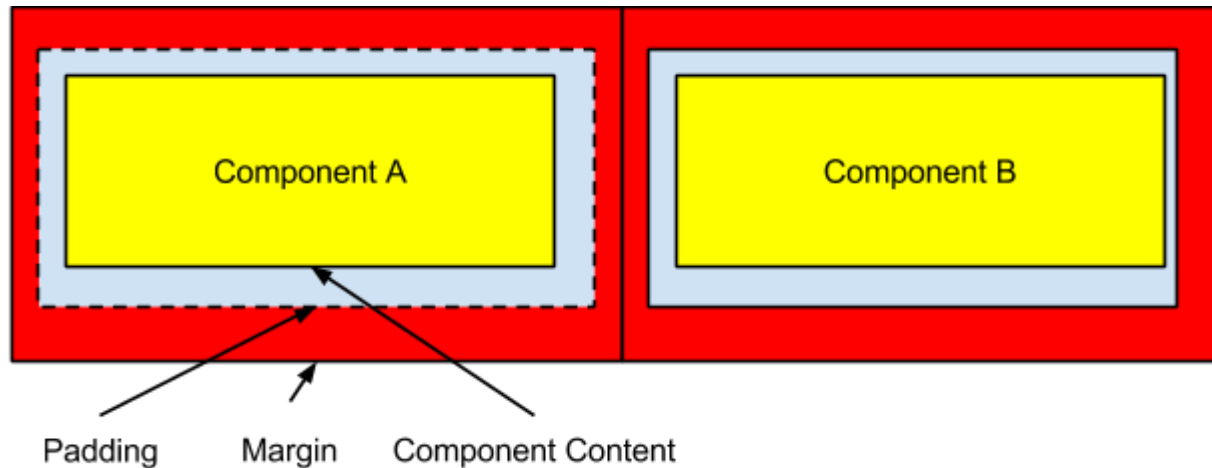
When pressing the ‘...’ button you can customize the border using multiple configuration types:



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:



The screenshot shows the 'Padding' tab in the Codename One Designer. The 'Derive' checkbox is unchecked. The settings are as follows:

Side	Value	Unit
Left	2	Pixels
Right	2	Millimeters (approximate)
Top	0	Pixels
Bottom	0	Pixels

Buttons for 'Derive Help' and 'Padding Help' are also visible.

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 components.

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

combolImage	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
includeNativeBool	True to derive from the platform native theme, false to create a blank theme that only uses the basic defaults.
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
minimizeOnBackBool	Indicates whether the form should minimize the entire application when the physical back button is pressed (if available) and no command is defined as the back command. Defaults to true.
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
popupCancelButtonBool	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 aarrgbbb 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 combolmage constant to a specific value in theme A and then switch to theme B that doesn't define the combolmage, the original theme A combolmage 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:



The dialog box titled "Select Resolutions" contains a table for selecting image sizes for different DPIs. The table has three columns: "Size", "Width", and "Height". The rows are "Very Low", "Low", "Medium", "High", "Very High", "HD", and "% (will affect all entries)". The "Very Low" row has a blue border around its Width input field, which contains the value "0". The "Medium" row has Width "64" and Height "4". The "High" row has Width "102" and Height "0". The "Very High" row has Width "128" and Height "4". The "HD" row has Width "0" and Height "0". The "% (will affect all entries)" row has Width "16" and Height "15". Below the table is a checkbox labeled "Square Images" which is checked. At the bottom right are "Cancel" and "OK" buttons.

Size	Width	Height
Very Low	0	0
Low	0	0
Medium	64	4
High	102	0
Very High	128	4
HD	0	0
% (will affect all entries)	16	15

☒ Square Images

Cancel OK

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 if 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 that 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.

Composite Components

Codename One components share a very generic hierarchy of inheritance e.g. Button derives from Label and thus receives all its abilities.

However, some components are composites and derive from the Container class. E.g. the Multi-Button is a composite button that derives from Container but acts as a button externally. Normally this is pretty seamless for the developer with a few things to keep in mind.

You should not use the Container derived methods on such a composite component. You can't cast it to the type that it relates to e.g. you can't cast MultiButton to Button.

Form

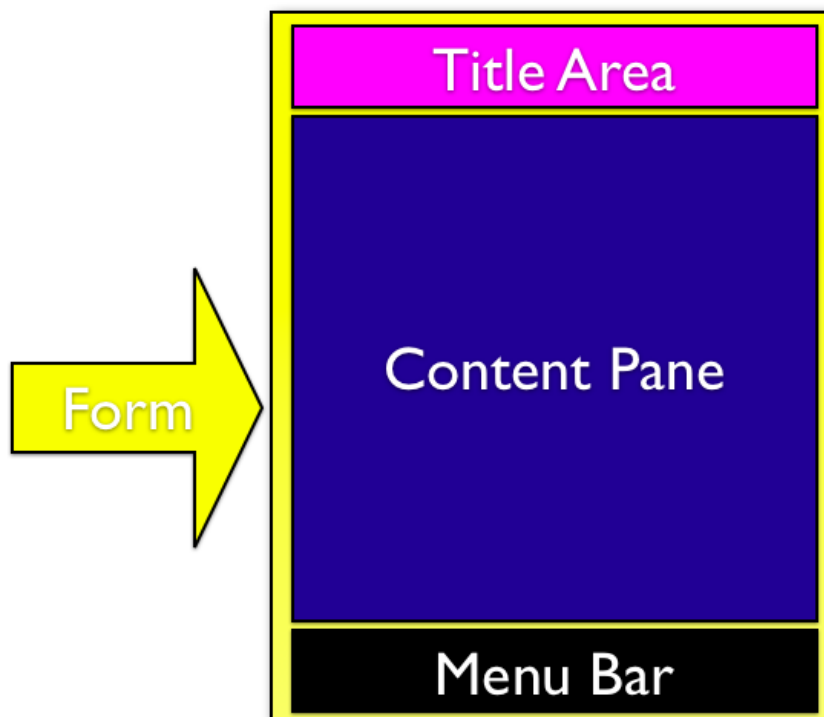
The top-level container of Codename One, Form derives container and it is effectively the element we "show". Only one form can be visible at any given time and we can get the currently visible form using the code:

```
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 it 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.

In addition to this the form has a layer on top of the content pane that is added implicitly when you invoke `getLayeredPane()`. This is a container that is placed on top of the content pane and allows you to place a component over another. Notice that you still need to place components using layout managers in order to get them to appear in the right place when using the layered pane.

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 `invokeAndWait()`).

Modal dialogs are an extremely useful way to prompt the user since the code has the user's 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 wrap and might end with "..." in some cases where appropriate. Developers can determine the placement of the label relative 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 (it's a [composite component](#)) 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.

Span Button

SpanButton is a [composite component](#) that looks/acts like a button but can break lines rather than crop them when the text is very long.

Span Label

SpanLabel is a [composite component](#) that looks/acts like a Label but can break lines rather than crop them when the text is very long.

On Off Switch

The OnOffSwitch allows you to write an application where the user can swipe a switch between two states on/off. This is a common UI paradigm in Android and iOS although its implemented in a radically different way in both platforms.

This is a rather elaborate component because of its very unique design on iOS but we were able to accommodate most of the small behaviors of the component into our version and it seamlessly adapts between the Android style and the iOS style.

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). By default `ComponentGroup` 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). The `ComponentGroupBool` flag is true by default in the iOS themes.

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.:

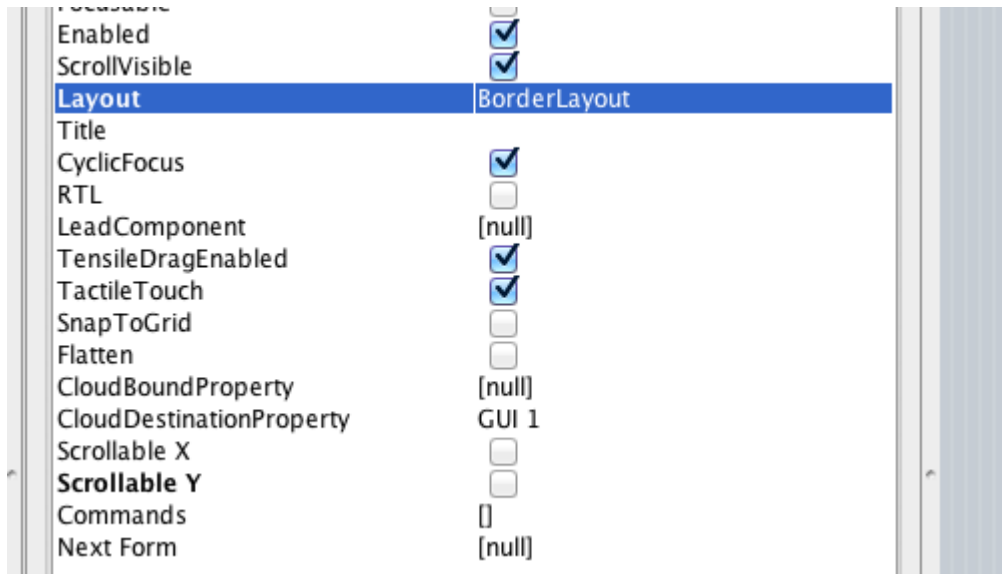
```
form.setScrollable(false);  
form.setLayout(new BorderLayout());  
form.addComponent(BorderLayout.CENTER, myList);
```

Using Lists In The GUI Builder

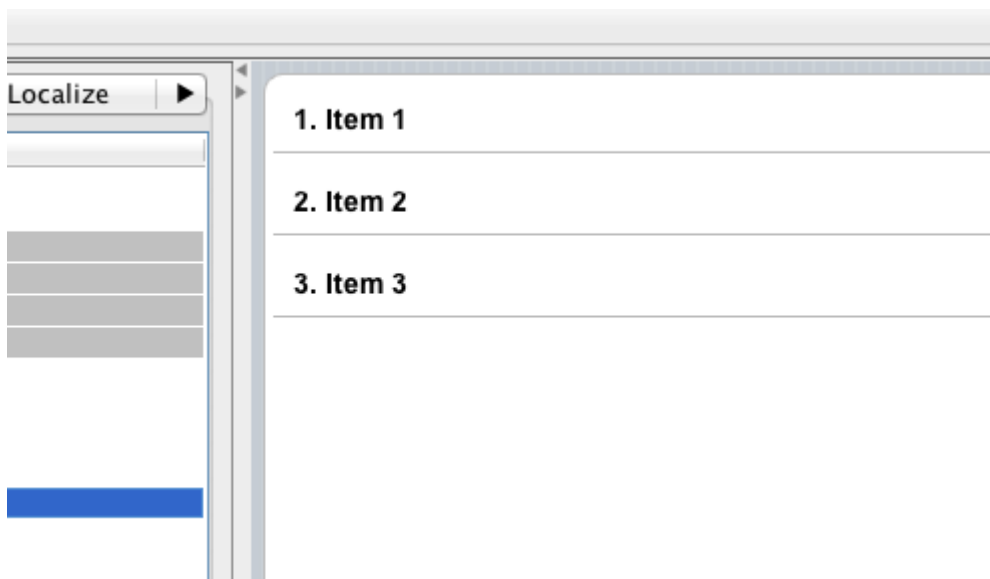
The Codename One GUI builder provides several simplifications to the concepts outlined below, you can build all portions of the list through the GUI builder or build portions of them using code if you so desire.

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.

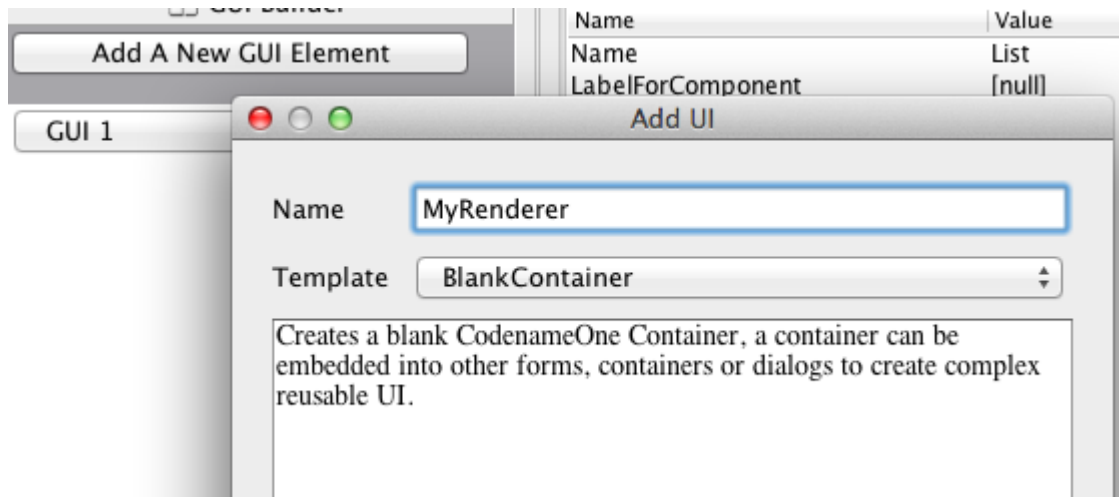
Start by creating a new GUI form, set its scrollable Y to false and set its layout to border layout



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



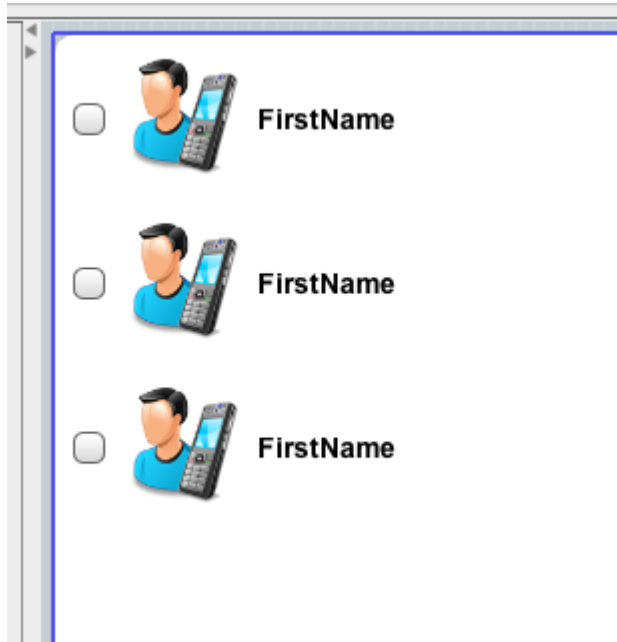
Next we will create the Renderer, which indicates how an individual item in the list appears, start by pressing the Add New GUI Element button and **select the “Blank Container”** option! Fill the name as “MyRenderer” or anything like that.



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.

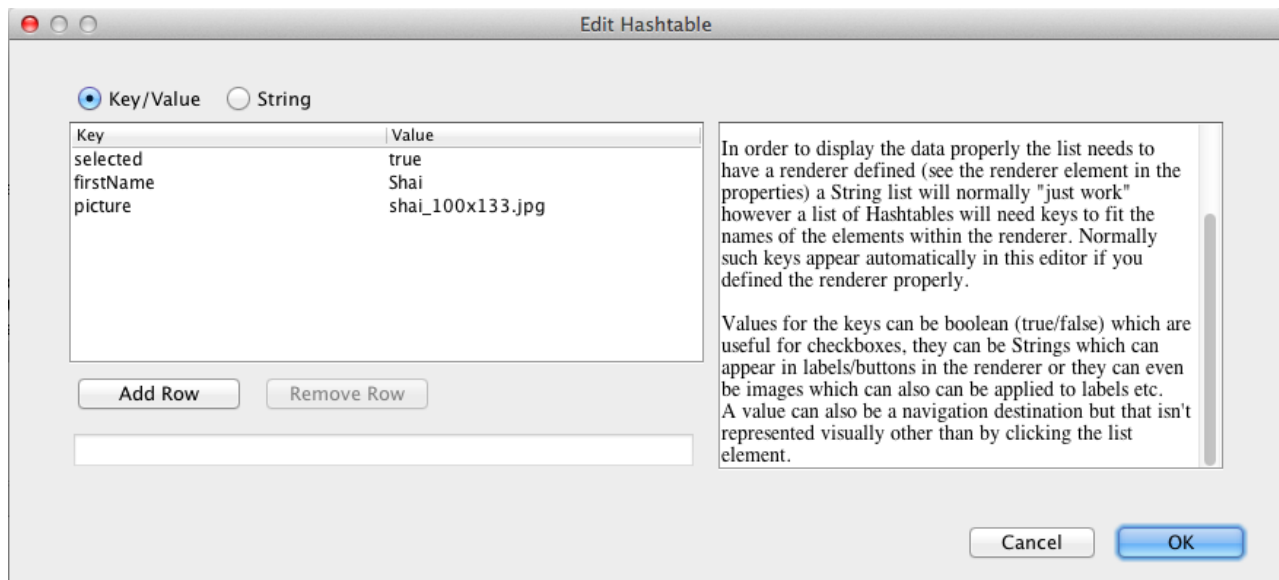


You'll notice that the data doesn't match the original content of the list, that is because the list contains strings instead of Hashtables. To fix this we must edit the list data.

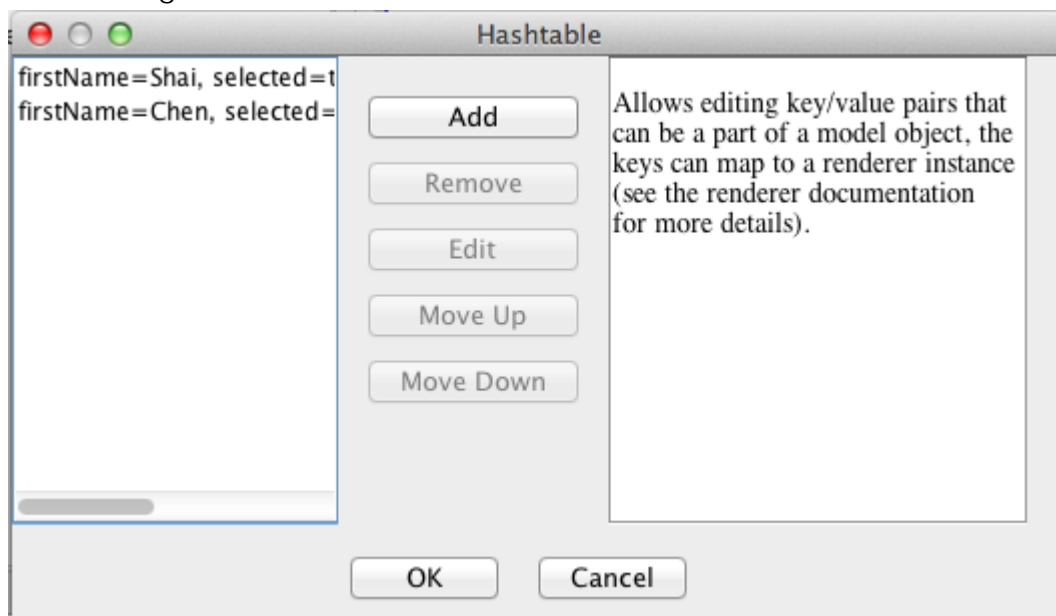
You can place your own data in the list using the GUI builder which is generally desired regardless of what you end up doing since this allows you to preview your designer in the GUI builder.

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).

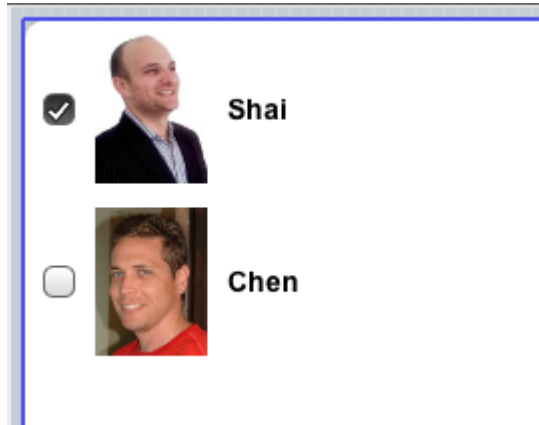
To populate the list via the GUI builder click the properties of the list and within them click the ListItems entry. The entries within can be strings or Hashtables, however in order to be customizable in the rendering stage we will need them all to be Hashtables. Remove all the current entries and add 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 render 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 we return a focus Label with semi transparency, as mentioned before the focus component can be modified within 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 UUID of the container passed to the generic list cell renderer and the background focus uses that same UUID 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 addDataChangeListener(DataChangeListener l) {
    }

    public void removeDataChangeListener(DataChangeListener 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 MultiList 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) {
```

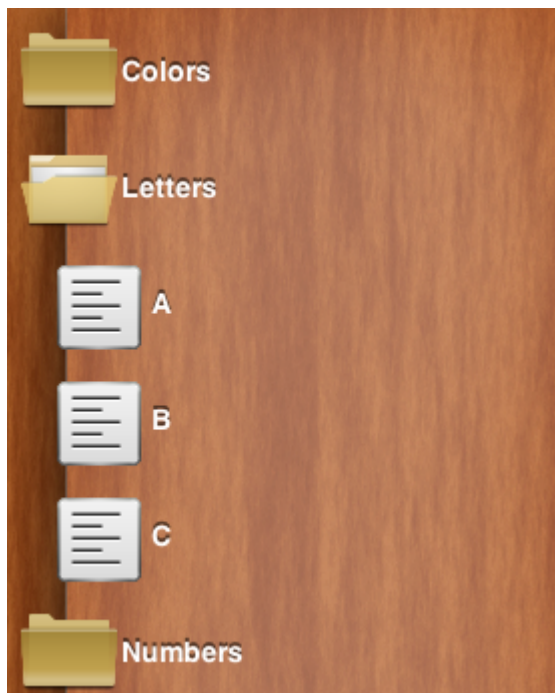
```
    if(parent == null) {
        Vector v = new Vector();
        for(int iter = 0 ; iter < arr[0].length ; iter++) {
            v.addElement(arr[0][iter]);
        }
        return v;
    }
```

```
    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

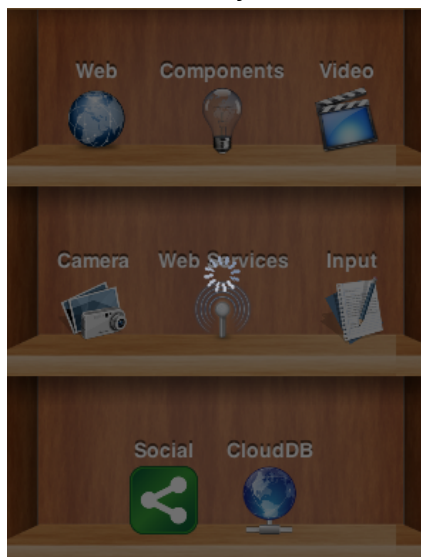
The share button allows you to help your users market your application by sharing it with their friends easily. When they press the button they will be faced with either the native sharing capability of the platform (as is available on Android & iOS) or a builtin set of sharing features such as facebook,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



Infinite Progress

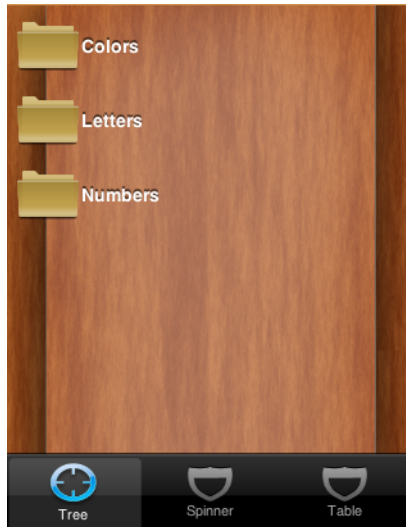
The infinite progress indicator spins an image infinitely; it is automatically associated with a default image for infinite progress from the native theme. This can be used in several ways e.g. you can place this component into a layout to indicate work underway.

A useful function in the infinite progress component is the `showInfiniteBlocking()` 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.

ImageViewer

The image viewer allows you to inspect, zoom and pan into an image. It also allows swiping between images if you have a set of images (using an image list model).

Here is a simple example of using the ImageViewer:

```
ImageViewer imv = new ImageViewer();
DefaultListModel<Image> images = new DefaultListModel<Image>(new Image[] {
    EncodedImage.create("/a.jpg"), EncodedImage.create("/b.jpg"),
    EncodedImage.create("/c.jpg")
});
imv.setImage(images.getItemAt(0));
imv.setImageList(images);
imv.setSwipePlaceholder(Image.createImage(5, 5));
```

Notice that we use a list to allow swiping between images (unnecessary if you have only one image), we also create a placeholder image to show while the image is still loading. Notice that encoded images aren't always fully loaded and so when you swipe if the images are really large you might see delays!

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");

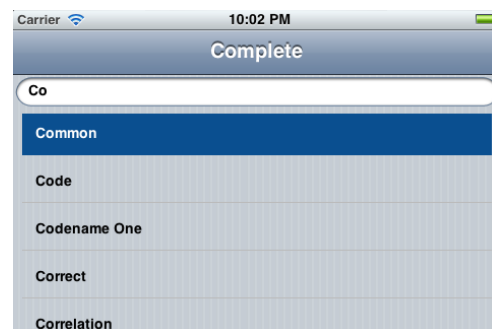
```

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

Auto Complete

The `AutoCompleteTextField` allows us to write text into a text field and select a completion entry from the list in a similar way to a search engine.

This is really easy to incorporate into your code, just replace your usage of `TextField` with `AutoCompleteTextField` and define the data that the autocomplete should work from. There is a default



implementation that accepts a String array or a ListModel for completion strings, this can work well for a "small" set of thousands (or tens of thousands) of entries.

However, if you wish to query a database or a web service you will need to derive the class and perform more advanced filtering by overriding the filter method and the getSuggestionModel method. You might also need to invoke updateFilterList() if your filter algorithm is asynchronous.

Here is a sample of a simple auto-complete that doesn't use the advanced features:

```
Form test = new Form("Complete");
test.setLayout(new BorderLayout(BorderLayout.Y_AXIS));
AutoCompleteTextField at = new AutoCompleteTextField(new String[] { "Common",
"Code", "Codename One", "Correct", "Correlation", "Co-location", "Corporate" } );
test.addComponent(at);
test.show();
```

Spinner & Picker

Spinner is a form of list that allows picking specific platform values from a spinning wheel in a similar way to the iOS date control. We are in the process of de-emphasizing Spinner, which varies too much between platforms in favor of the new Picker component that uses a native interface to pick an entry.

A Picker acts very much like a text field which will popup a Spinner dialog when tapped, this should work nicely on all platforms since the Picker will popup the native dialog/spinner when available.



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 hierarchy 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".

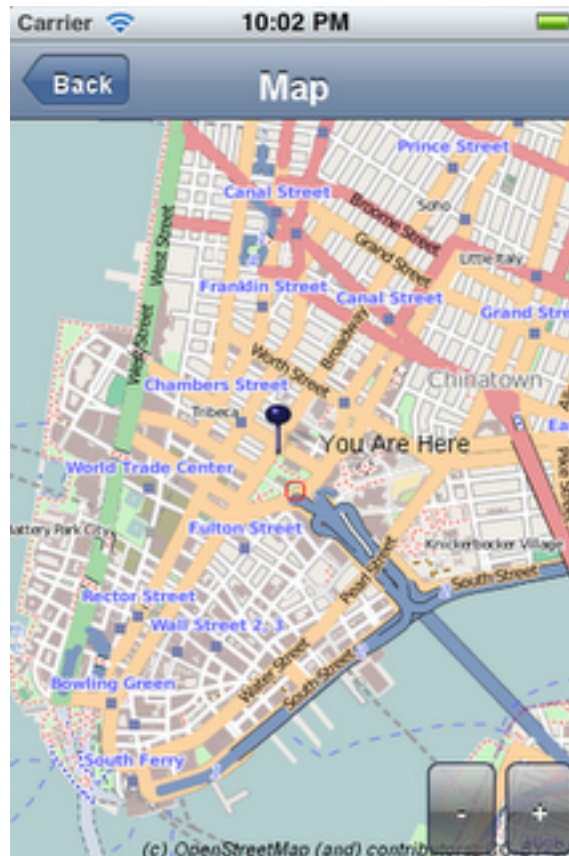
Navigation replaces the content of the embedded container; it finds the embedded container based on the component that broadcast the event. If you want to navigate manually just use the `showContainer()` method which accepts a component, you can give any component that is under the EmbeddedContainer 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.getLongitude());
    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.getLongitude());
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.

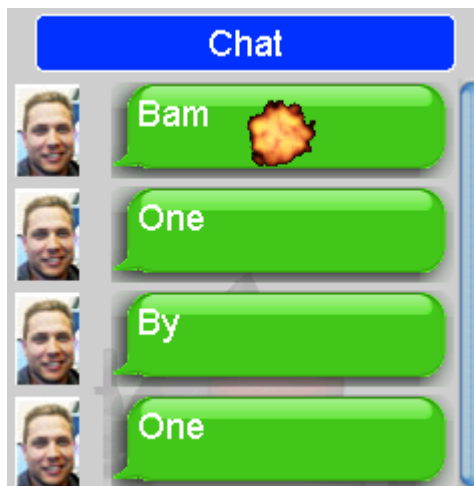
Developers can implement their own custom transition and install it to components by deriving the transition class, although most commonly the built in `CommonTransition` class is used for almost everything.

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

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)`.

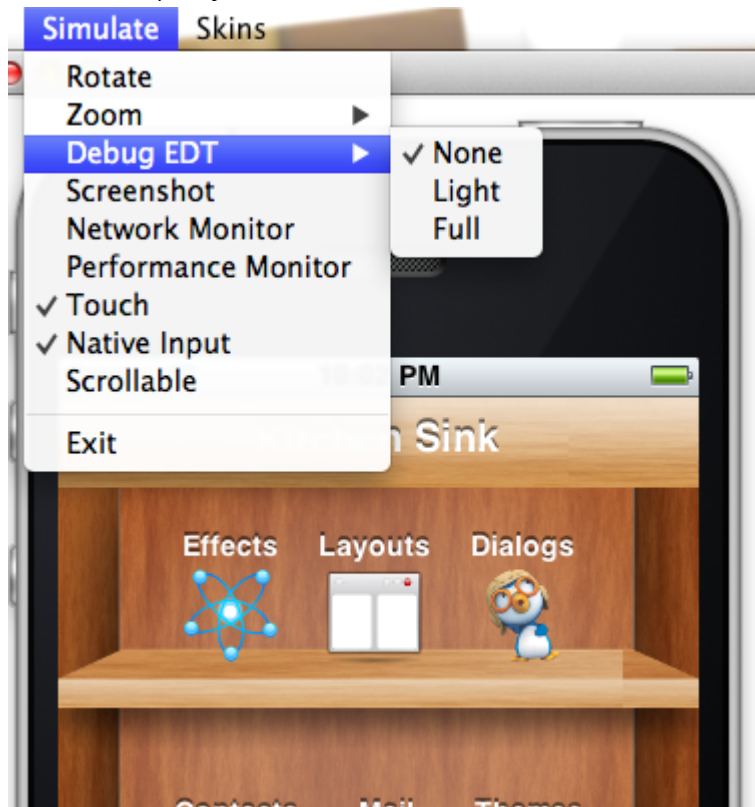
`isEDT()` is useful for generic code that needs to test whether the current code is executing on the EDT.

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 some caveats my apply though... It's 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 that 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 programmer's 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 `invokeAndWait`, 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: `invokeAndWait`.

The best way to explain this is by example:

```
public void actionPerformed(ActionEvent ev) {  
    label.setText("Initiating IO, please wait...");  
    Display.getInstance().invokeAndWait(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` "blocks" 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.

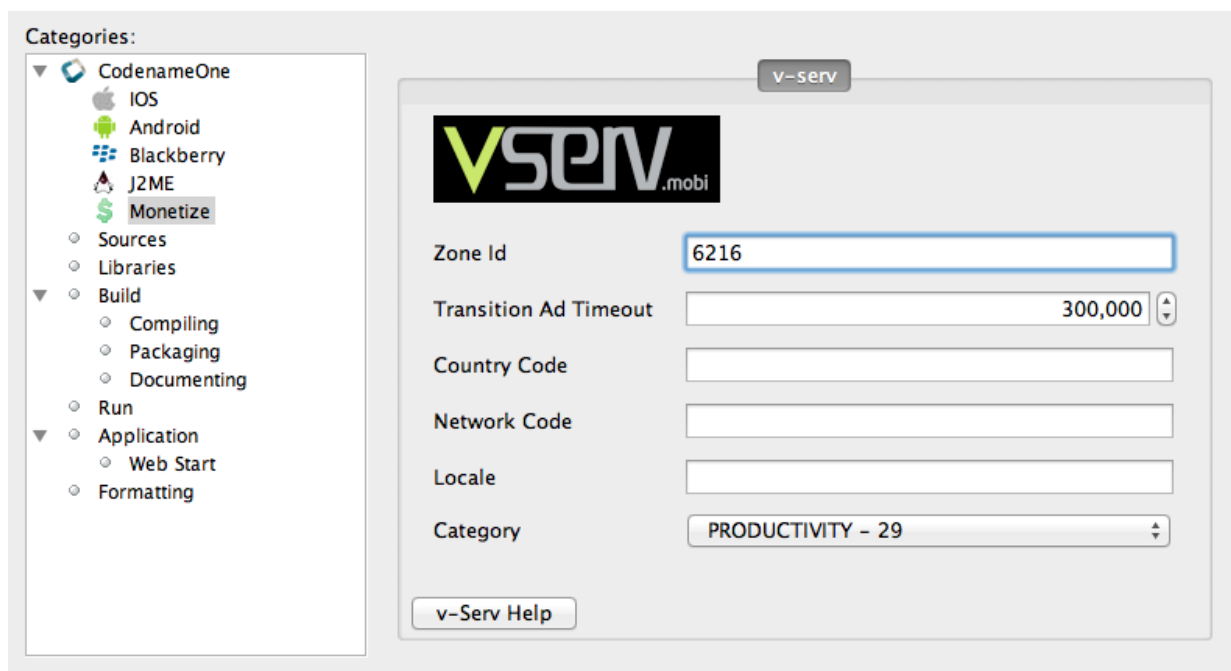
Monetization

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 provider³.

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.



The screenshot shows the 'Monetize' configuration window in the Codename One IDE. On the left is a 'Categories' tree with 'Monetize' selected. The main area is titled 'v-serv' and contains the vserv logo. Below the logo are several configuration fields: 'Zone Id' (text input with value 6216), 'Transition Ad Timeout' (spin box with value 300,000), 'Country Code' (text input), 'Network Code' (text input), 'Locale' (text input), and 'Category' (dropdown menu showing 'PRODUCTIVITY - 29'). A 'v-Serv Help' button is at the bottom left.

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.

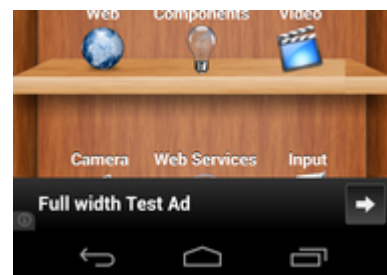
To initiate the inneractive ads you will need to enter the following line in your application `init(Object)` or `start()` method:

```
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 (appld in the GUI builder) with which you can determine the application that will receive the payouts for the ads.

Google Play Ads

The most effective network is the simplest banner ad support. To enable mobile ads just [create an ad unit](#) in Admob's website, you should end up with the key similar to this: ca-app-pub-8610616152754010/3413603324



To enable this for Android just define the `android.googleAdUnitId=ca-app-pub-8610616152754010/3413603324` in the build arguments and for iOS use the same as in `ios.googleAdUnitId`. The rest is seamless, the right ad will be created for you at the bottom of the screen and the form should automatically shrink to fit the ad. This shrinking is implemented differently between iOS and Android due to some constraints but the result should be similar and this should work reasonably well with device rotation as well.

In App Purchase

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

1. Google's in app purchase
2. Apple's 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();

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);
    }
    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"));
}
```

The item names in the demo code above should be hard coded and added to the appropriate stores inventory. Which is a very platform specific process for iTunes and Google play. Once this is done you should be able to issue a purchase request either in the real or the sandbox store.

Graphics, Drawing, Images & Fonts

This chapter covers the basics of drawing manually using the Codename One API, notice that drawing is considered a low level API that 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(0xffffffff);
        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 that will be covered in the IO section.

Here are the pros/cons and logic behind every image type and how it's 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 (Codename 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.

Understanding Encoded Images & Image Locking

To understand locking we first need to understand EncodedImage. EncodedImage stores the data of the image in RAM (png or JPEG data), which is normally pretty small, unlike the full-decoded image, which can take up to width X height X 4. When a user tries to draw an encoded image we check a WeakReference cache and if the image is cached then we show it otherwise we load the image, cache it then draw.

Naturally loading the image is more expensive so we want the images that are on the current form to remain in cache (otherwise GC will thrash a lot). That's where lock() kicks in, when lock() is active we keep a hard reference to the actual native image so it won't get GC'd. This REALLY improves performance!

Internally we invoke this automatically for bg images, icons etc. which results in a huge performance boost. This makes sense since these images are currently showing so they will be in RAM anyway. However, if you use a complex renderer or custom drawing UI you should lock() your images where possible!

To verify that locking might be a problem you can launch the performance monitor tool, if you get log messages that indicate that an unlocked image was drawn you might have a problem.

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.

That's 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).

Developers need to write the data of the object in the externalize method using the methods in the data output stream and read the data of the object in the internalize method e.g.:

```
public void externalize(DataOutputStream out) throws IOException {
    out.writeUTF(name);
    if(value != null) {
        out.writeBoolean(true);
        out.writeUTF(value);
    } else {
        out.writeBoolean(false);
    }
    if(domain != null) {
        out.writeBoolean(true);
        out.writeUTF(domain);
    } else {
        out.writeBoolean(false);
    }
    out.writeLong(expires);
}

public void internalize(int version, DataInputStream in) throws IOException {
    name = in.readUTF();
    if(in.readBoolean()) {
        value = in.readUTF();
    }
    if(in.readBoolean()) {
        domain = in.readUTF();
    }
    expires = in.readLong();
}
```

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](#).

Storage also offers a very simple API in the form of the Preferences class. The Preferences class allows developers to store simple variables, strings, numbers, booleans etc. in storage without wringing any storage code. This is a common usage within applications e.g. you have a server token that you need to store:

```
Preferences.set("token", myToken);
```

```
// token will be null if it was never set
```

```
String token = Preferences.get("token", null);
```

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 Codename One specific type; you cannot store arbitrary objects in the cloud.
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 below.

```
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 new 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.setRequestHeader(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, e.g. for downloading, caching images locally, RSS etc. You can find out more about these services in the [services package](#).

Of note is the `MultiPartRequest`, which allows submitting large blocks of data to a server without the limitations of typical requests. It includes special API's to add files thus allows

upload of images, video etc. Notice that the server to which the upload request submits data needs to be able to process a multipart request, which is a special mime request.

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.showInfiniteBlocking();  
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 methods are invoked by the parser to indicate internal parser states, this is similar to the way traditional XML SAX event parsers work.

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.

On the opposite side of the `XMLParser` we also have the `XMLWriter` class which can generate XML from the Element hierarchy thus allowing a developer to mutate (modify) the elements and save them to a writer stream.

Cached Data Service

The `CachedDataService` pretty useful, say you have an image stored locally as image X. Normally the `ImageDownloadService` will never check for update if it has a local cache of the image. This isn't a bad thing, its pretty efficient. However, it might be important to update the image if it changed but you don't want to fetch the whole thing...

The cached data service will fetch data if it isn't cached locally and cache it. When you "refresh" it will send a special HTTP request that will only send back the data if it has been updated since the last refresh:

```
CachedDataService.register();
CachedData d = (CachedData)Storage.getInstance().readObject("LocallyCachedData");

if(d == null) {
    d = new CachedData();
    d.setUrl("http://....");
}
// check if there is a new version of this on the server
CachedDataService.updateData(d, new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        // invoked when/if the data arrives, we now have a fresh cache
        Storage.getInstance().writeObject("LocallyCachedData", d);
    }
});
```

GZIP

Gzip is a very common compression format based on the lz algorithm, it's used by web servers around the world to compress data.

Codename One supports GZipInputStream and GZipOutputStream, which allow you to compress data seamlessly into a stream and extract compressed data from a stream. This is very useful and can be applied to every arbitrary stream.

Codename One also features a GZConnectionRequest, which will automatically unzip an HTTP response if it is indeed gzipped. Notice that some devices (iOS) always request gzip'ed data and always decompress it for us, however in the case of iOS it doesn't remove the gzipped header. The GZConnectionRequest is aware of such behaviors so its better to use that when connecting to the network (if applicable).

By default GZConnectionRequest doesn't request gzipped data (only unzips it when its received) but its pretty easy to do so just add the HTTP header Accept-Encoding: gzip e.g.:

```
GZConnectionRequest con = new GZConnectionRequest();
con.addRequestHeader("Accept-Encoding", "gzip");
```

Do the rest as usual and you should have smaller responses by potential.

This capability isn't in the global `ConnectionRequest` since it will increase the size of the distribution to everyone. If you do not need the gzip functionality the obfuscator will just strip it out during the compile process.

Miscellaneous Features

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

SMS, Dial (Phone) & E-Mail

SMS calling and emailing seem unrelated yet they are all available in Display as a one line command specifically Display's `sendSMS`, `sendMessage` & `dial` all allow you to perform these common tasks usually by launching the native application that performs the task.

The email messaging API has an additional ability within the `Message` class in `sendMessageViaCloud`. This method allows you to use the Codename One cloud to send an email without end user interaction. This feature is available to pro users only since it makes use of the Codename One cloud:

```
Message m = new Message("<html><body>Check out <a  
href=\"http://www.codenameone.com/\">Codename One</a></body></html>");  
m.setMimeType(Message.MIME_HTML);
```

```
// notice that we provide a plain text alternative as well in the send method  
boolean success = m.sendMessageViaCloudSync("Codename One",  
"destination@domain.com", "Name Of User", "Message Subject",  
"Check out Codename One at http://www.codenameone.com/");
```

Contacts API

The contacts API provides us with the means to query the phone's addressbook, delete elements from it and create new entries into it. To get the platform specific list of contacts you can use `String[] contacts = ContactsManager.getAllContacts();`

Notice that on some platforms this will prompt the user for permissions (specifically iOS) and the user might choose not to grant that permission. To detect whether this is the case you can invoke `isContactsPermissionGranted()` **after** invoking `getAllContacts()`. This can help you adapt your error message to the user.

Once you have a `Contact` you can use the `getContactByld` method, however the default method is a bit slow if you want to pull a large batch of contacts. The solution for this is to

only extract the data that you need via `getContactById(String id, boolean includesFullName,`

`boolean includesPicture, boolean includesNumbers, boolean includesEmail, boolean includeAddress)`

Here you can specify true only for the attributes that actually matter to you.

You can use `createContact(String firstName, String familyName, String officePhone, String homePhone, String cellPhone, String email)` to add a new contact and `deleteContact(String id)` to delete a contact.

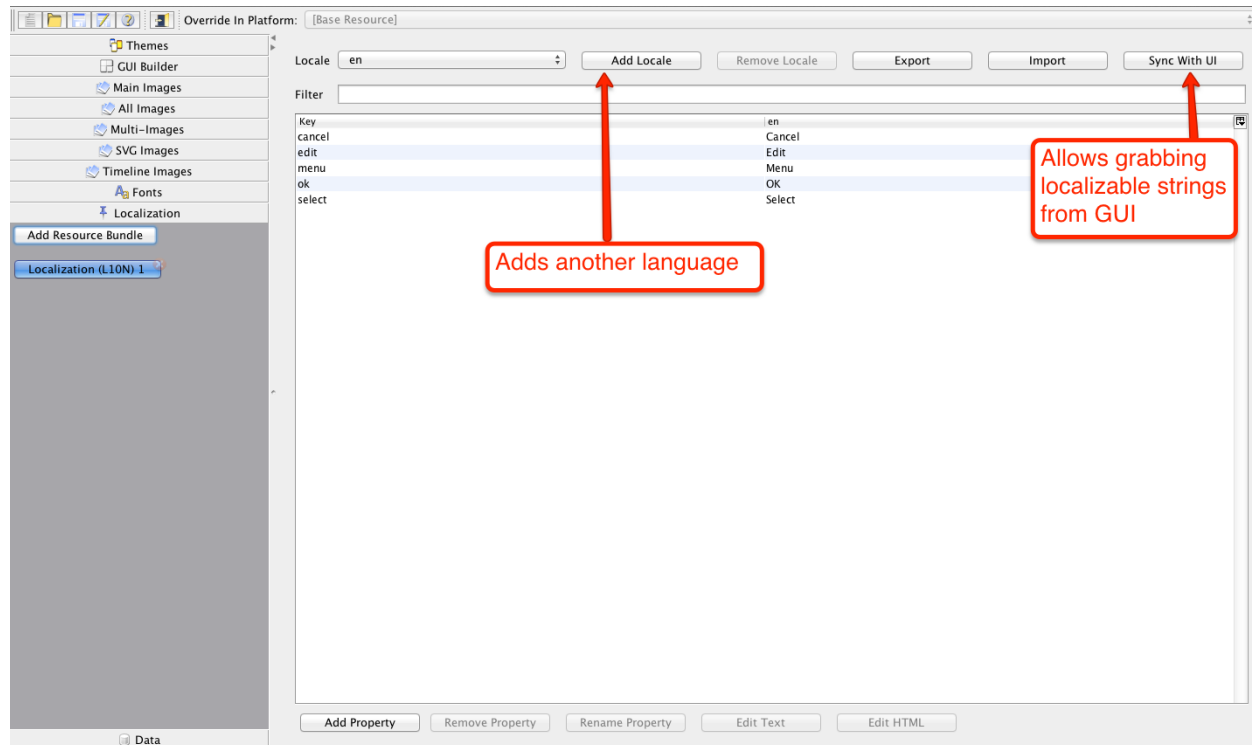
If you just want to display all contacts in a List to allow the user to pick a contact you can use the `ContactsModel` with a set of ID's.

Localization & Internationalization (L10N & I18N)

Localization (l10n) means adapting to a locale which is more than just translating to a specific language but also to a specific language within environment e.g. `en_US` != `en_UK`.

Internationalization (i18n) is the process of creating one application that adapts to all locales and regional requirements.

Codename One supports automatic localization and seamless internationalization of an application using the Codename One design tool. Notice that although localization is performed in the design tool most features apply to hand coded applications as well. The only exception is the tool that automatically extracts localizable strings from the GUI.



To translate an application you need to use the localization section of the Codename One Designer. This section features a handy tool to extract localization called Sync With UI, its a great tool to get you started assuming you used the GUI builder. You can add additional languages by pressing the Add Locale button.

This generates “bundles” in the resource file which are really just key/value pairs mapping a string in one language to another language.

You can install the bundle using code like this:

```
UIManager.getInstance().setBundle(res.getL10N("l10n", local));
```

Once installed a resource bundle takes over the UI and every string set to a label (and label like components) will be automatically localized based on the bundle. You can also use the localize method of UIManager to perform localization on your own.

An exception for localization is the TextField/TextArea components both of which contain user data, in those cases the text will not be localized to avoid accidental localization of user input.

You can preview localization in the theme mode within the Codename One designer by selecting advanced and picking your locale then clicking the theme again.

You can export and import resource bundles as standard Java properties files, CSV and XML. The formats are pretty standard for most localization shops, the XML format

Codename One supports is the one used by Android's string bundles which means most shops should easily localize it.

Localization Manager

The `LocalizationManager` class includes a multitude of features useful for common localization tasks. It allows formatting numbers/dates & time based on platform locale. It also provides a great deal of the information you need such as the language/locale information you need to pick the proper resource bundle.

RTL/Bidi

RTL stands for right to left, in the world of internationalization it refers to languages that are written from right to left (Arabic, Hebrew, Syriac, Thaana).

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.

The main issue in the Codename One world is in the layouts, which need to reverse on the fly. Codename One supports this via an RTL flag on all components that is derived from the global RTL flag in `UIManager`.

Resource bundles can also include special case constant `@rtl`, which indicates if a language is written from right to left. This allows everything to automatically reverse.

When in RTL mode the UI will be the exact mirror so WEST will become EAST, RIGHT will become LEFT and this would be true for paddings/margins as well.

If you have a special case where you don't want this behavior you will need to wrap it with an `isRTL` check.

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

- Bidi algorithm - allows converting between logical to visual representation for rendering
- Global 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.

Location - GPS

The location API allows us to track changes in device location or the current user position. The most basic usage for the API allows us to just fetch a device Location, notice that this API is blocking and can take a while to return:

```
Location position = LocationManager.getLocationManager().getCurrentLocationSync();
```

Notice that there is a method called `getCurrentLocation()` which will return the current state immediately and might not be accurate for some cases.

The `getCurrentLocationSync()` method is very good for cases where you only need to fetch a current location once and not repeatedly query location. It activates the GPS then turns it

off to avoid excessive battery usage. However, if an application needs to track motion or position over time it should use the location listener API to track location as such:

```
public MyListener implements LocationListener {  
    public void locationUpdated(Location location) {  
        // update UI etc.  
    }  
}
```

```
    public void providerStateChanged(int newState) {  
        // handle status changes/errors appropriately  
    }  
}
```

```
LocationManager locationManager = LocationManager.getInstance();  
locationManager.setLocationListener(new MyListener());
```

Capture - Photos, Video, Audio

The capture API allows us to use the camera to capture photographs or the microphone to capture audio. It even includes an API for video capture.

The API itself couldn't be simpler:

```
String filePath = Capture.capturePhoto();
```

Just captures and returns a path to a photo (temporary file which you should copy locally), you can either open it using the Image class or copy it using the FileSystemStorage class. Video and audio include similar API's.

Codescan - Barcode & QR code scanner

The codescan package allows us to scan barcodes and qr codes using the device camera. Notice that on weaker devices (feature phones and RIM devices) this functionality is very limited and as of this writing this feature isn't available on Windows Phone.

Using the API is quite simple, just invoke the call to scan and implement the proper callback to get the results:

```
if(Codescan.getInstance() != null) {  
    final Button qrCode = new Button("Scan QR");  
    cnt.addComponent(qrCode);  
    qrCode.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent evt) {  
            Codescan.getInstance().scanQRCode(new ScanResult() {
```

```
        public void scanCompleted(String contents, String formatName, byte[]
rawBytes) {
            qrCode.setText("QR: " + contents);
        }

        public void scanCanceled() {
        }

        public void scanError(int errorCode, String message) {
        }
    };
}
});
final Button barCode = new Button("Scan Barcode");
cnt.addComponent(barCode);
barCode.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        CodeScanner.getInstance().scanBarCode(new ScanResult() {
            public void scanCompleted(String contents, String formatName, byte[]
rawBytes) {
                barCode.setText("Bar: " + contents);
            }

            public void scanCanceled() {
            }

            public void scanError(int errorCode, String message) {
            }
        });
    }
});
}
```

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.

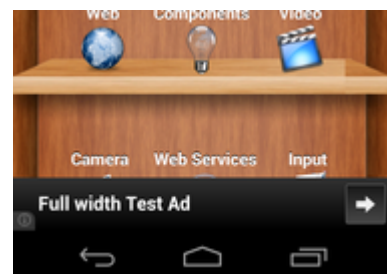
If your application is not a GUI builder application or you would like to send more detailed data you can use the `Analytics.visit()` method to indicate that you are entering a specific page.

In 2013 Google introduced an improved application level analytics API that is specifically built for mobile apps. However, it requires a slightly different API from the server. You can activate this specific mode by invoking `setAppsMode(true)`.

When using this mode you can also report errors and crashes to the Google analytics server using the `sendCrashReport(Throwable, String message, boolean fatal)` method.

Facebook Support (legacy)

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



⁴ See <http://developers.facebook.com/docs/reference/api/>

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 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("received 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);
```

Native Facebook Support

The previous section covered support for Facebook that at the time of this writing works on all platforms, however this support is being deprecated partially by Facebook. Currently the old method will still work but developers are expected to migrate to the native login over time.

The after deprecation the FaceBookAccess class should still work for the most part, the only major change should be in the login process.

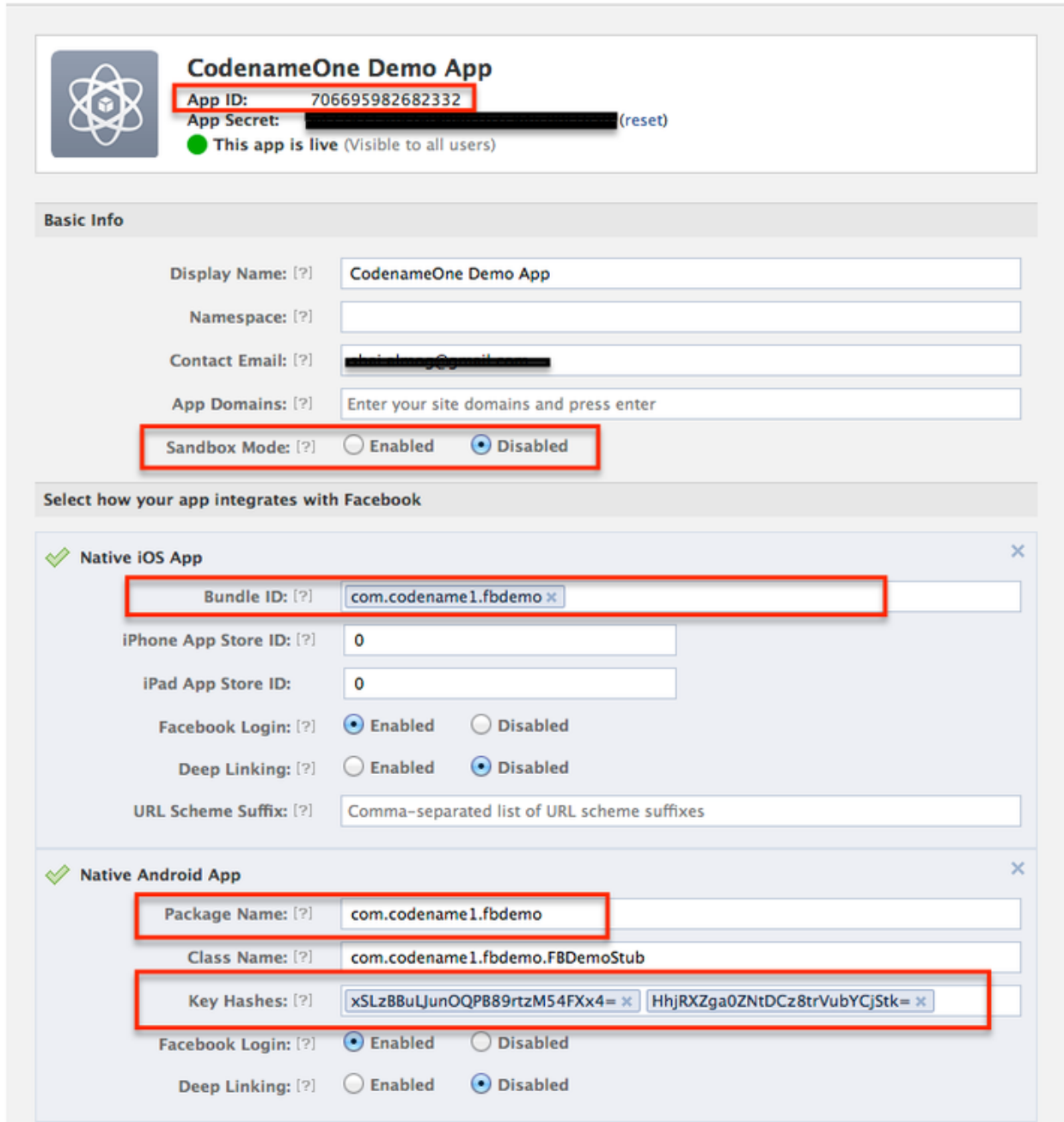
If you just want something in the form of a "share button" we suggest you refer to the builtin share button, which uses native sharing on both iOS and Android.

The new Facebook API is very simple; in fact as of this writing it includes only 5 methods. There are really 3 significant methods in the API, login/logout and isLoggedIn(). You can also bind a listener to login event callbacks, which is really pretty simple. The difficulty isn't here though.

Before you get started you need to go to the page on Facebook for app creation:<https://developers.facebook.com/apps>

Here you should create your app and make sure to enter the package name of the Codename One application both for the section marked as Bundle Id and Package Name (see the red highlighting in the figure below).

Apps ▶ CodenameOne Demo App ▶ Basic



CodenameOne Demo App

App ID: 706695982682332
App Secret: [REDACTED] (reset)
This app is live (Visible to all users)

Basic Info

Display Name: [?] CodenameOne Demo App
Namespace: [?]
Contact Email: [?] [REDACTED]
App Domains: [?] Enter your site domains and press enter
Sandbox Mode: [?] ☐ Enabled ☒ Disabled

Select how your app integrates with Facebook

✓ Native iOS App

Bundle ID: [?] com.codename1.fbdemo
iPhone App Store ID: [?] 0
iPad App Store ID: [?] 0
Facebook Login: [?] ☒ Enabled ☐ Disabled
Deep Linking: [?] ☐ Enabled ☒ Disabled
URL Scheme Suffix: [?] Comma-separated list of URL scheme suffixes

✓ Native Android App

Package Name: [?] com.codename1.fbdemo
Class Name: [?] com.codename1.fbdemo.FBDemoStub
Key Hashes: [?] xSLzBBuLJunOQP889rtzM54FXx4= HhjRXZga0ZNtDCz8trVubYCjStk=
Facebook Login: [?] ☒ Enabled ☐ Disabled
Deep Linking: [?] ☐ Enabled ☒ Disabled

Once you do that you need to define the build argument facebook.appld to the app ID in the Facebook application (see the red marking at the top of the image).

Now when you send a build and invoke FacebookConnect.login() this should work as expected on iOS but it will fail on Android. The reason is that Facebook requires a hash

from Android developers to identify your app. However, their instructions to generate said hash don't work... The only way we could find for generating the hash properly is on an Android device.

If you have DDMS you can connect the device to your machine and see the printouts including the hashcode (notice the hashcode will change whenever you send a debug build so make sure to only use Android release builds). You can also get the value of the hashcode from `Display.getInstance().getProperty("facebook_hash", null);` This will return the hash only on Android of course.

You can take this hash and paste it into the section marked Key Hashes in the native android app section. Notice you can have multiple hashes if you have more than one certificates or applications.

Once login is successful the existing facebook API's from the Facebook package should work pretty much as you would expect.

SideMenuBar - Hamburger Sidemenu

The Hamburger sidemenu is the menu style popularized by the Facebook app, its called a Hamburger because of the 3-line icon on the top left resembling a hamburger patty between two buns (get it: its a side menu...)!

To enable the side menu set the command behavior to side menu and it just works. You can do this by setting the `commandBehavior` theme constant in the Codename One designer to "Side" or via the `setCommandBehavior` method in `Display`. You will also need to invoke:

Then just add commands and watch them make their way into the side menu allowing you to build any sort of navigation you desire.

The side menu goes much deeper than that, e.g. the ability to place a side menu on the right, top or on both sides of the title (as in the facebook app). You can accomplish this by using code such as `cmd.putClientProperty(SideMenuBar.COMMAND_PLACEMENT_KEY, SideMenuBar.COMMAND_PLACEMENT_VALUE_RIGHT);`

Or as you might see in this more detailed example where you can just swap menu placements on the fly:

```
public class MyApplication {
```

```
private Form current;
private enum SideMenuMode {
    SIDE, RIGHT_SIDE {
        public String getCommandHint() {
            return SideMenuBar.COMMAND_PLACEMENT_VALUE_RIGHT;
        }
    }, BOTH_SIDES {
        boolean b;
        public String getCommandHint() {
            b = !b;
            if(b) {
                return null;
            }
            return SideMenuBar.COMMAND_PLACEMENT_VALUE_RIGHT;
        }
    }, TOP {
        public String getCommandHint() {
            return SideMenuBar.COMMAND_PLACEMENT_VALUE_TOP;
        }
    };

    public String getCommandHint() {
        return null;
    }

    public void updateCommand(Command c) {
        String h = getCommandHint();
        if(h == null) {
            return;
        }
        c.putClientProperty(SideMenuBar.COMMAND_PLACEMENT_KEY, h);
    }
};
```

```
SideMenuMode mode = SideMenuMode.SIDE;
```

```
public void init(Object context) {
    try{
        Resources theme = Resources.openLayered("/theme");
```

```
UIManager.getInstance().setThemeProps(theme.getTheme(theme.getThemeResourceNames()[0]));
```

```
UIManager.getInstance().getLookAndFeel().setMenuBarClass(SideMenuBar.class);
```

```
Display.getInstance().setCommandBehavior(Display.COMMAND_BEHAVIOR_SIDE_NAVIGATION);
```

```
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

```
public void start() {  
    if (current != null) {  
        current.show();  
        return;  
    }  
    newHiForm("Clean");  
}
```

```
void newHiForm(String title) {  
    Form hi = new Form(title);  
    hi.setName(title);  
    buildSideMenu(hi);  
    hi.show();  
}
```

```
void buildSideMenu(Form hi) {  
    Command changeToSideMenuLeft = new Command("Left Menu") {  
        public void actionPerformed(ActionEvent ev) {  
            mode = SideMenuMode.SIDE;  
            newHiForm("Left");  
        }  
    };  
    Command changeToSideMenuRight = new Command("Right Menu") {  
        public void actionPerformed(ActionEvent ev) {  
            mode = SideMenuMode.RIGHT_SIDE;  
        }  
    };  
}
```

```
        newHiForm("Right");
    }
};
Command changeToSideMenuBoth = new Command("Both Menu") {
    public void actionPerformed(ActionEvent ev) {
        mode = SideMenuMode.BOTH_SIDES;
        newHiForm("Both");
    }
};
Command changeToSideMenuTop = new Command("Top Menu") {
    public void actionPerformed(ActionEvent ev) {
        mode = SideMenuMode.TOP;
        newHiForm("Top");
    }
};

Command dummy = new Command("Dummy 1");
Command dummy2 = new Command("Dummy 2");

mode.updateCommand(dummy);
hi.addCommand(dummy);
mode.updateCommand(dummy2);
hi.addCommand(dummy2);
mode.updateCommand(changeToSideMenuLeft);
hi.addCommand(changeToSideMenuLeft);
mode.updateCommand(changeToSideMenuRight);
hi.addCommand(changeToSideMenuRight);
mode.updateCommand(changeToSideMenuBoth);
hi.addCommand(changeToSideMenuBoth);
mode.updateCommand(changeToSideMenuTop);
hi.addCommand(changeToSideMenuTop);
}

public void stop() {
    current = Display.getInstance().getCurrent();
}

public void destroy() {
```

```
}  
}
```

One of the nice things about the side menu bar is that you can add just about anything into the side menu bar by using the `SideComponent` property e.g.:

```
Component customCmp = ...;  
Command cmd = ...;  
cmd.putClientProperty("SideComponent", customCmp);
```

This is remarkably useful but its also somewhat problematic for some developers, the `SideMenuBar` is pretty complex so if we just set a button to the custom component and invoke `showForm()` we will not have any transition out of the side menu bar. Thankfully we added several options to solve these issues.

The first is `actionable` which you enable by just turning it on as such:

```
cmd.putClientProperty("Actionable", Boolean.TRUE);
```

This effectively means that the custom component will look exactly the same, but when it's touched/clicked it will act like any other command on the list. This uses a lead component trick to make the hierarchy (or component) in `customCmp` act as a single action.

There are several additional options that allow you to just bind action events and then "manage" the `SideMenuBar` e.g.:

- `SideMenuBar.isShowing()` - useful for writing generic code that might occur when the `SideMenuBar` is on the form.
- `SideMenuBar.closeCurrentMenu()` - allows you to close the menu, this is useful if you are not navigating to another form.
- `SideMenuBar.closeCurrentMenu(Runnable)` - just like `closeCurrentMenu()` however it will invoke the `run()` method when complete. This allows you to navigate to another form after the menu close animation completed.

The `TitleCommand` property allows you to flag a command as something you would want to see in the right hand title area and not within the `SideMenu` area. Just place it into a component using `cmd.putClientProperty("TitleCommand", Boolean.TRUE);`

Last but not least we also have some helpful theme constants within the side menu bar that you might not be familiar with:

- `sideMenuImage` - pretty obvious, this is the hamburger image we use to open the menu.

- `sideMenuPressImage` - this is the pressed version of the image above. Its optional and the `sideMenuImage` will be used by default.
- `rightSideMenuImage/rightSideMenuPressImage` - identical to the `sideMenuImage/sideMenuPressImage` only specific to the right side navigation.
- `sideMenuFoldedSwipeBool` - by default a swipe will open the side menu. You can disable that functionality by setting this theme constant to false.
- `hideBackCommandBool` - often comes up in discussion, allows hiding the back command from the side menu so it only appears in the hardware button/iOS navigation.
- `hideLeftSideMenuBool` - allows hiding the left hand menu which is useful for a case of top or right based side menu.
- `sideMenuShadowImage` - image that represents the drop shadow drawn on the side of the menu.
- `sideMenuTensileDragBool` - allows disabling the tensile draw within the side menu command area

Pull To Refresh

Pull to refresh is the common UI paradigm that Twitter popularized where the user can pull down the form/container to receive an update. Adding this to Codename One couldn't be simpler! Just invoke `addPullToRefresh(Runnable)` on a scrollable container (or form) and the runnable method will be invoked when the refresh operation occurs.

Infinite Scroll Adapter

Pull to refresh is only half the story although it is a really nice feature useful for pulling new updates. Codename One has the ability to have infinite (or really large lists) but making a container with arbitrary components grow infinitely is normally a bit of a hassle.

In some of the newer web UI's such as Tumblr and Twitter the data is fetched dynamically when you reach a fixed location in the form, this is a simpler approach than the one demonstrated by the list model but in some regards its more practical. A user can't just start jumping around and fetching the entire list, this works better with most REST API's and is pretty powerful on its own.



For this purpose we created the `InfiniteScrollAdapter`, which is a really simple class that binds to a container and gives you the ability to add components to it. The API is remarkably simple you just invoke the static method `InfiniteScrollAdapter.createInfiniteScroll()` with an empty container then wait for it to invoke the runnable you submit to it.

The runnable will be invoked on the EDT so be sure not to block it (unless you use an `AndWait` or `invokeAndBlock` method), in it you can fetch data and once you are done add any set of components you like using the `addMoreComponents()` method. Notice that you shouldn't just add/remove components on your own since this will mess up the container.

Here is a simple example that adds buttons and sleeps to simulated slow network activity:

```
final Form test = new Form("Infinite");

test.setLayout(new BoxLayout(BoxLayout.Y_AXIS));
InfiniteScrollAdapter.createInfiniteScroll(test.getContentPane(), new Runnable() {
    private int counter = 1;
    public void run() {
        // simulate network latency
        Display.getInstance().invokeAndBlock(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {}
            }
        });

        Component[] buttons = new Component[20];
        for(int iter = 0 ; iter < buttons.length ; iter++) {
            buttons[iter] = new Button("Button: " + counter);
            counter++;
        }
        InfiniteScrollAdapter.addMoreComponents(test.getContentPane(), buttons, true);
    }
});
test.show();
```

Performance, Size & Debugging

Reducing Resource File Size

It's 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 pales 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. Notice that JPEG's work on all supported devices and are typically 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 element justifies 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.

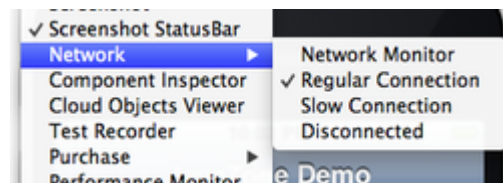
Performance Monitor

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.

Network Speed

This feature is actually more useful for general debugging however it is sometimes useful to simulate a slow/disconnected network to see how this affects performance. For this purpose the Codename One simulator allows you to slow



down networking or even fake a disconnected network to see how your application handles such cases.

Debugging Codename One Sources

One of the biggest advantages in Codename One over pretty much any other mobile solution is that its realistically open source. Realistically means that even an average developer can dig into 90% of the Codename One source code change it and contribute to it!

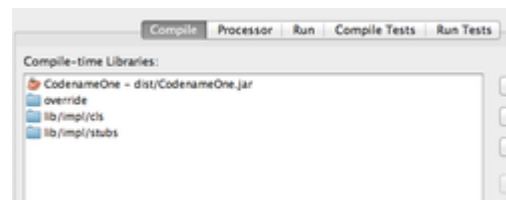
However, sadly most developers don't even try and most of those who do focus only on the aspect of building for devices rather than the advantage of much easier debugging. By incorporating the Codename One sources you can instantly see the effect of changes we made in SVN without waiting for a plugin update. You can, debug into Codename One code which can help you pinpoint issues in your own code and also in resolving issues in Codename One!

Start by [checking out the Codename One sources from SVN](http://codenameone.googlecode.com/svn/trunk/), use the following URL <http://codenameone.googlecode.com/svn/trunk/> which should allow for anonymous readonly checkout of the latest sources!

Now that you have the sources open the CodenameOne project that is in the root and the JavaSEPort that is in the Ports directory using NetBeans. Notice that these projects might be marked in red and you will probably need to right click on them and select Resolve Reference Problems. You will probably need to fix the JDK settings, and the libraries to point at the correct local paths.

Once you do that you can build both projects without a problem. Notice that you will probably get a minor compilation error due to a build.xml line in the Codename One project, don't fret. Just edit that line and comment it out.

Select any Codename One project in NetBeans, right click and click properties.

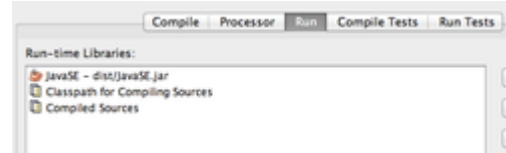


Now select "Libraries" from the tree to your right select all the jars within the compile tab. Click remove.

Click the Add Project button and select the project for Codename One in the SVN.

Now select the Run tab and remove the JavaSE.jar file from there by selecting it and pressing remove.

Add the JavaSEPort project using the Add Project button and then use the Move Up button to make sure it is at the top most position since it needs to override everything else at runtime.



You are now good to go, now you can just place breakpoints within Codename One source code, edit it and test it. You can step into it with the debugger which can save you a lot of time when tracking a problem.

Device Testing Framework/Unit Testing

Codename One includes a built in testing framework and test recorder tool as part of the simulator. This allows developers to build both functional and unit test execution on top of Codename One. It even enables sending tests for execution on the device (pro-only feature).

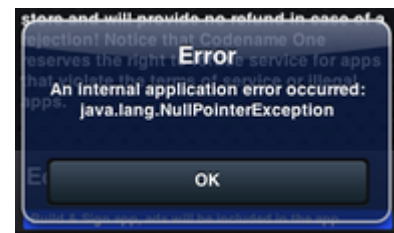
To get started with the testing framework, launch the application and open the test recorder in the simulator menu. Once you press record a test will be generate for you as you use the application.

You can build tests using the Codename One testing package to manipulate the Codename One UI programmatically and perform various assertions.

EDT Error Handler and sendLog

Handling errors or exceptions in a deployed product is pretty difficult, most users would just throw away your app and some would give it a negative rating without providing you with the opportunity to actually fix the bug that might have happened.

Google improved on this a bit by allowing users to submit stack traces for failures on Android devices but this requires the users approval for sending personal data which you might not need if you only want to receive the stack trace and maybe some basic application state (without violating user privacy).



For quite some time Codename One had a very powerful feature that allows you to both catch and report such errors, the error reporting feature uses the Codename One cloud which is exclusive for pro/enterprise users. Normally in Codename One we catch all exceptions on the EDT (which is where most exceptions occur) and just display an error to the user as you can see in the picture. Unfortunately this isn't very helpful to us as developers who really want to see the stack; furthermore we might prefer the user doesn't see an error message at all!

Codename One allows us to grab all exceptions that occur on the EDT and handle them using the method `addEdtErrorHandler` in the `Display` class. Adding this to the `Log`'s ability to report errors directly to us and we can get a very powerful tool that will send us an email with information when a crash occurs!

```
Display.getInstance().addEdtErrorHandler(new ActionListener() {  
    public void actionPerformed(ActionEvent evt) {  
        evt.consume();  
        Log.p("Exception in AppName version " +  
Display.getInstance().getProperty("AppVersion", "Unknown"));  
        Log.p("OS " + Display.getInstance().getPlatformName());  
        Log.p("Error " + evt.getSource());  
        Log.p("Current Form " + Display.getInstance().getCurrent().getName());  
        Log.e((Throwable)evt.getSource());  
        Log.sendLog();  
    }  
});
```

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.debug	true/false defaults to true - indicates whether to include the debug version in the build
android.release	true/false defaults to true - indicates whether to include the release version in the build
android.installLocation	Maps to android:installLocation manifest entry defaults to auto. Can also be set to internalOnly or preferExternal.
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, services etc.
android.xpermissions	additional permissions for the Android manifest
android.xintent_filter	Allows adding an intent filter to the main android activity
android.licenseKey	The license key for the Android app, this is useful when using features such as purchase to verify that the in-app-purchase wasn't injected by an attacker
android.stack_size	Size in bytes for the Android stack thread
android.statusbar_hidden	true/false defaults to false. When set to true hides the status

n	bar on Android devices.
android.sharedUserId	Allows adding a manifest attribute for the sharedUserId option
android.sharedUserLabel	Allows adding a manifest attribute for the sharedUserLabel option
android.web_loading_hidden	true/false defaults to false - set to true to hide the progress indicator that appears when loading a web page on Android.
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.
android.theme	Light or Dark defaults to Light. On Android 4+ the default Holo theme is used to render the native widgets in some cases and this indicates whether holo light or holo dark is used. Currently this doesn't affect the Codename One theme but that might change 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 false). 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.themeMode	default/legacy/modern/auto (defaults to default). Default means you don't define a theme mode. Currently this is equivalent to legacy. In the future we will switch this to be equivalent to auto. legacy - this will behave like iOS 6 regardless of the device you are running on. modern - this will behave like iOS 7 regardless of the device you are running on. auto - this will behave like iOS 6 on older devices and iOS 7

	on newer devices.
ios.interface_orientation	UIInterfaceOrientationPortrait by default. Indicates the orientation, one or more of (separated by colon :): UIInterfaceOrientationPortrait, UIInterfaceOrientationPortraitUpsideDown, UIInterfaceOrientationLandscapeLeft, UIInterfaceOrientationLandscapeRight
ios.no_strip	true/false (defaults to false) a pro only feature that keeps debug information within the built binary thus allowing crash reports from iOS to contain symbol information (only on debug builds not on itunes store builds).
ios.xcode_version	The version of xcode used on the server. Defaults to 4.5; currently accepts 5.0 as an option and nothing else.
ios.unsafe	true/false (defaults to false). If you define ios.unsafe=true You will get an application that won't throw ArrayIndexOutOfBoundsException exceptions or NullPointerExceptions, however it might crash for such cases! This is one of those flags that you will need to test REALLY well before using in production, however once enabled it should noticeably improve the performance of Codename One.
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.
rim.nativeBrowser	true/false defaults to false. Enables the native blackberry browser on OS 5 or higher. It is disabled by default since it might casue crashes on some cases.
rim.obfuscation	true/false defaults to false. Obfuscate the JAR before invoking the rimc compiler.

ios.plistInject	entries to inject into the iOS plist file during build.
ios.includePush	true/false (defaults to false). Whether to include the push capabilities in the iOS build.
noExtraResources	true/false (defaults to false). Blocks codename one from injecting its own resources when set to true, the only effect this has is in slightly reducing archive size.
j2me.iconSize	Defaults to 48x48. The size of the icon in the format of width x height (without the spacing).

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!](http://www.codenameone.com)" 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. It's 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). Finders are shortcut methods that allow us to find a component instance within the container hierarchy. 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 NativeInterface {  
    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.

Notice that if you want to use a native library (jar, .a file etc.) just place it within the appropriate native directory and it will be packaged into the final executable. You would only be able to reference it from the native code and not from the Codename One code, which means you will need to build native interfaces to access it.

Native Permissions

Normally permissions in Codename One are pretty seamless, we traverse the bytecode and automatically assign permissions to Android applications based on the API's used by the developer.

However, when accessing native functionality this just won't work since native code might require specialized permissions and we don't/can't run any serious analysis on it (it can be just about anything).

So if you require additional permissions in your Android native code you need to define them in the build arguments using the `android.xpermissions` build argument and setting it to your additional permissions e.g.: `<uses-permission android:name="android.permission.READ_CALENDAR" />`

Libraries - cn1lib

Support for JAR files in Codename One has been a source of confusion so its probably a good idea to revisit this subject again and clarify all the details.

The first source of confusion is changing the classpath. You should NEVER change the classpath or add an external JAR via the IDE classpath UI. The reasoning here is very simple, these IDE's don't package the JAR's into the final executable and even if they did these JAR's would probably use features unavailable or inappropriate for the device (e.g. `java.io.File` etc.).

There are two use cases for wanting JAR's and they both have very different solutions:

1. Modularity - you want to divide your work to an external group. For this purpose use the `cn1lib` approach.
2. Work with an existing JAR. For this you will need native interfaces mentioned in the section above. Notice that native interfaces can be used within a `cn1lib`!

`Cn1lib`'s address the modularity aspect (for existing jars just refer to the native interfaces section), you can wrap them with a `cn1lib` but you will need a native interface anyway. You can create a `cn1lib` in NetBeans and IDEA, it's really just a simple ant project with some special targets and a simple ant task for stubbing. In it you can write all your source code (including native code and libs as described below), when you build the file you will get a `cn1lib` file that you can place in your project's lib directory.

After a right click and refresh project libs completion will be available for you and you will be able to work as if the code was a part of your project.

You can automate this process by editing the build.xml and copying/refreshing projects, all operations with these libs are just simple ant tasks.

Drag & Drop

Unlike other platforms that 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 dragging 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 building classes to provide a slightly different feel to some of the transition effects.

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.

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.

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.

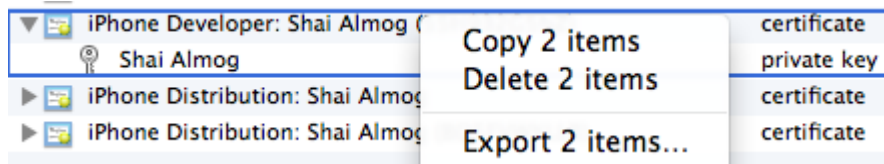
You then create a provisioning profile which comes in two flavors: distribution (for building the release version of your application) and development. The development provisioning profile needs to contain the devices on which you want to test.

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

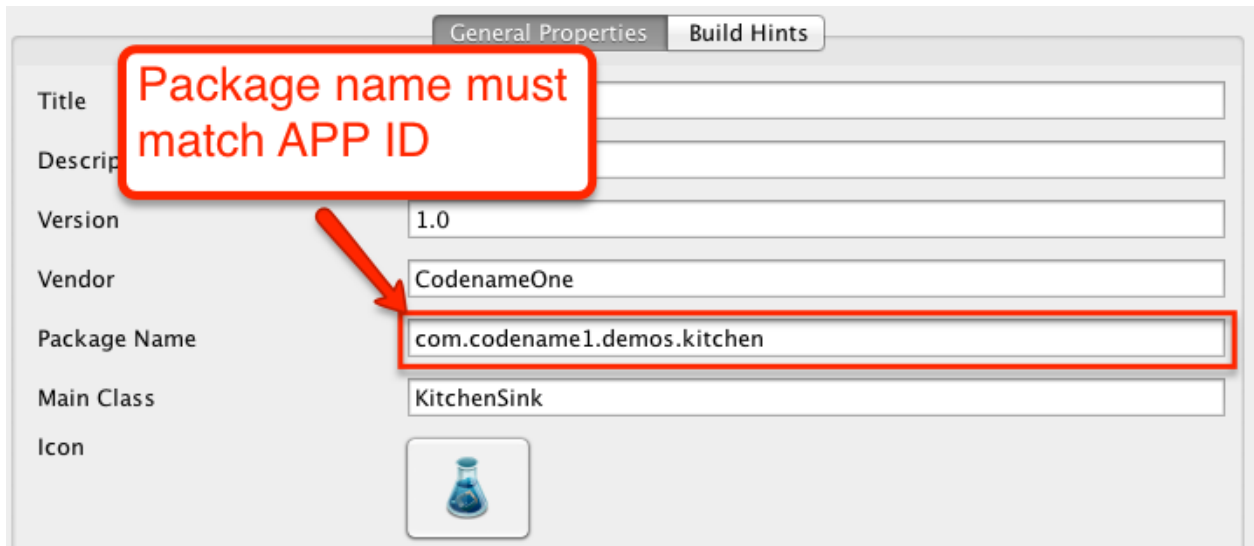
iOS Code Signing Fail Checklist

Below is a list of common things people get wrong when signing and a set of suggestions for things to check. Notice that some of these signing failures will sometimes manifest themselves during build and sometimes will manifest during the install of the application.

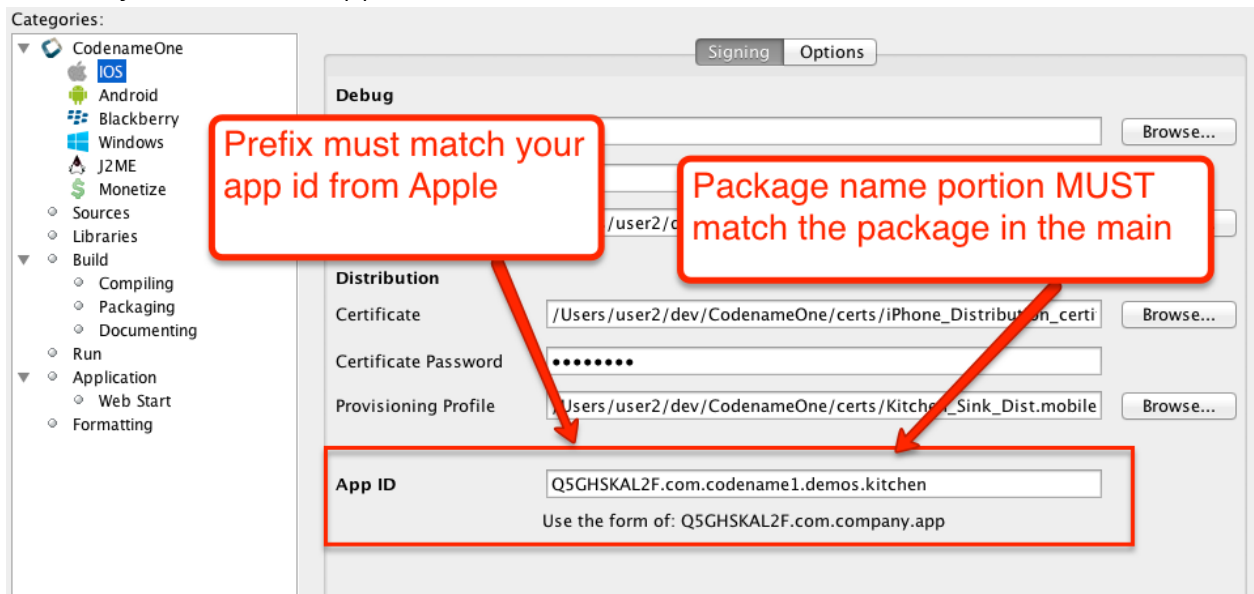
1. You must use a Mac to generate the P12 certificates. There is no way around it! Tutorials that show otherwise will not work!
We would like to automate it in the future (in a similar way to our Android signing tool), but for now you can use [MacInCloud](#), which has a free version.
Notice that this is something you need to do once a year (generate P12), you will also need a Mac to upload your final app to the store though.
2. When exporting the P12 certificate make sure that you selected BOTH the public and the private keys as illustrated here. If you only see one entry (no private key) then you created the CSR (signing request) on a different machine than the one where you imported the resulting CER file.



3. Make sure the package matches between the main preferences screen in the IDE and the iOS settings screen.




4. Make sure the prefix for the app id in the iOS section of the preferences matches the one you have from Apple



5. Make sure your provisioning profile's app id matches your package name or is a * provisioning profile. Both are sampled in the pictures below, notice that you would need an actual package name for push/in-app-purchase support as well as for app store distribution.

AllApps *

 **ID**

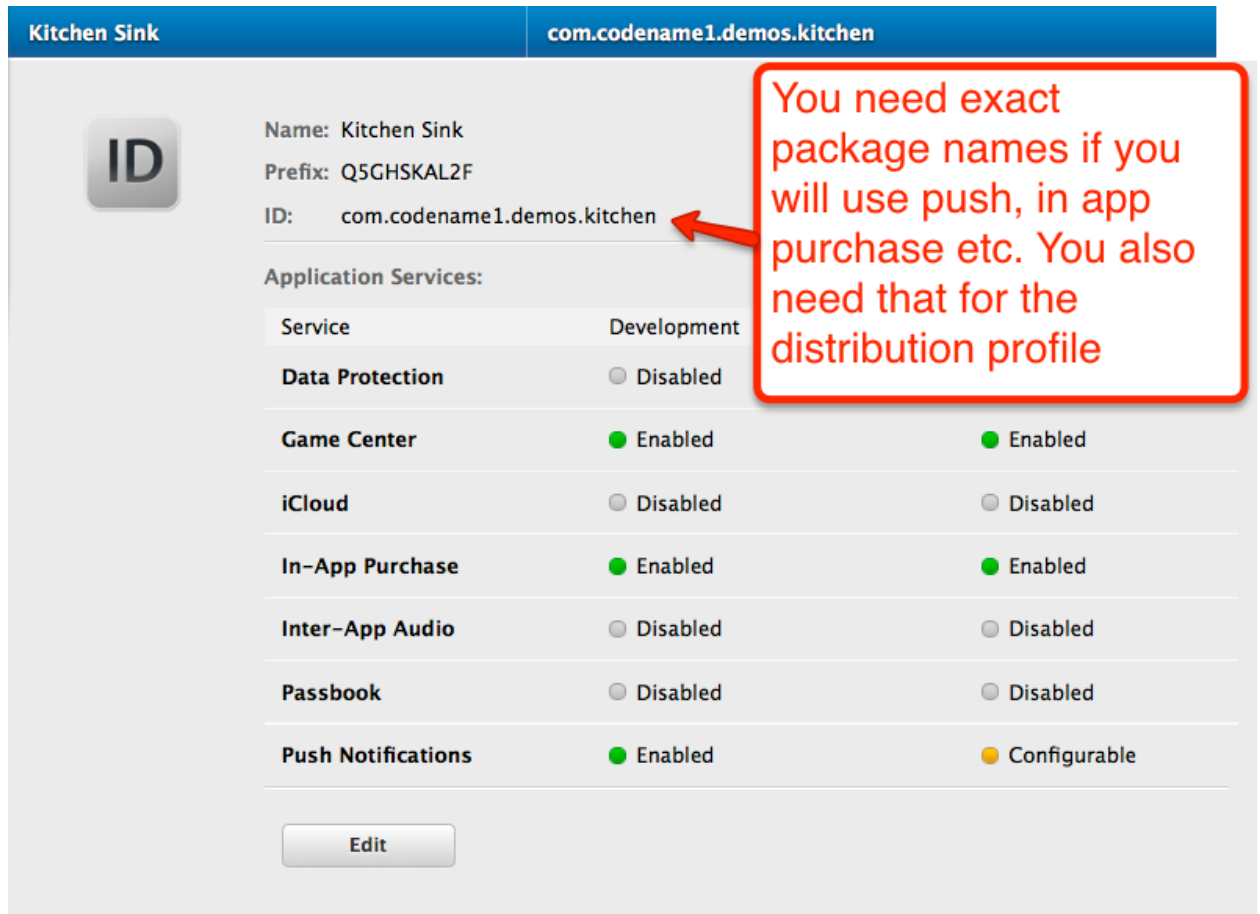
Name: AllApps
Prefix: Q5GHSKAL2F
ID: *

Generic provisioning works for apps that don't have push

Application Services:

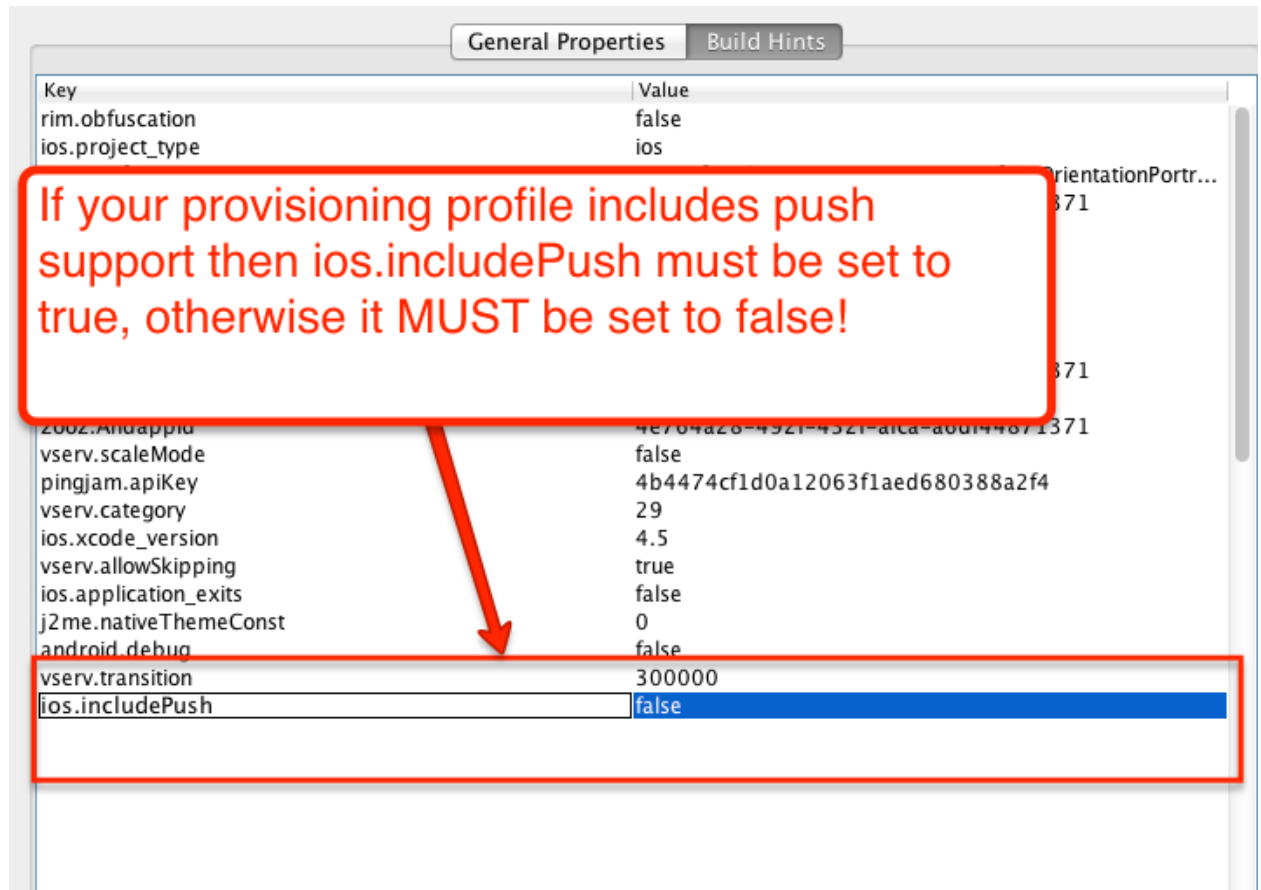
Service	Development	Distribution
Data Protection	<input type="radio"/> Disabled	<input type="radio"/> Disabled
Game Center	<input type="radio"/> Disabled	<input type="radio"/> Disabled
iCloud	<input type="radio"/> Disabled	<input type="radio"/> Disabled
In-App Purchase	<input type="radio"/> Disabled	<input type="radio"/> Disabled
Inter-App Audio	<input type="radio"/> Disabled	<input type="radio"/> Disabled
Passbook	<input type="radio"/> Disabled	<input type="radio"/> Disabled
Push Notifications	<input type="radio"/> Disabled	<input type="radio"/> Disabled

Edit



6. Make sure the certificate and provisioning profile are from the same source (if you work with multiple accounts), notice that provisioning profiles and certificates expire so you will need to regenerate provisioning when your certificate expires or is revoked.
7. If you declare push in the provisioning profile then `ios.includePush` (in the build arguments) **MUST** be set to true, otherwise it **MUST** be set to false (see pictures

below).



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](#). Once you obtain the certificates you need to install 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

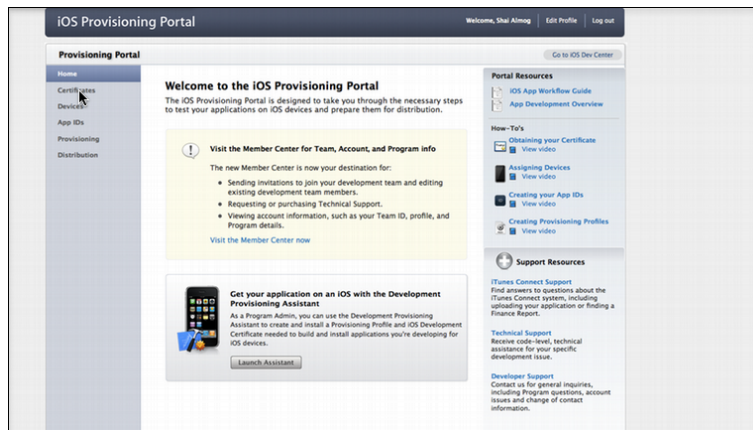
Currently signing J2ME applications isn't supported. You can use tools such as the Sprint WTK to sign the resulting jad/jar produced by Codename One.

Appendix: Working With iOS

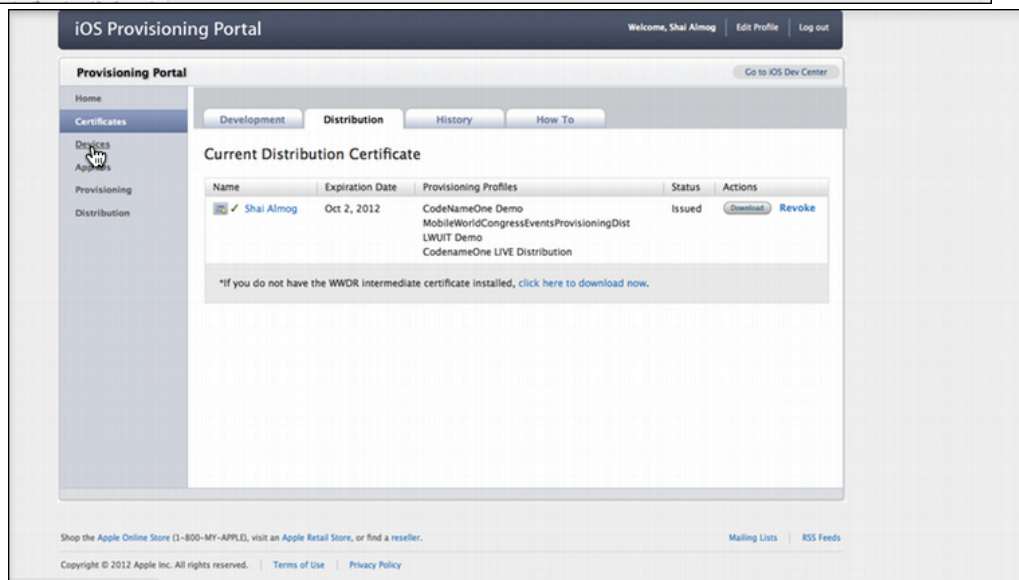
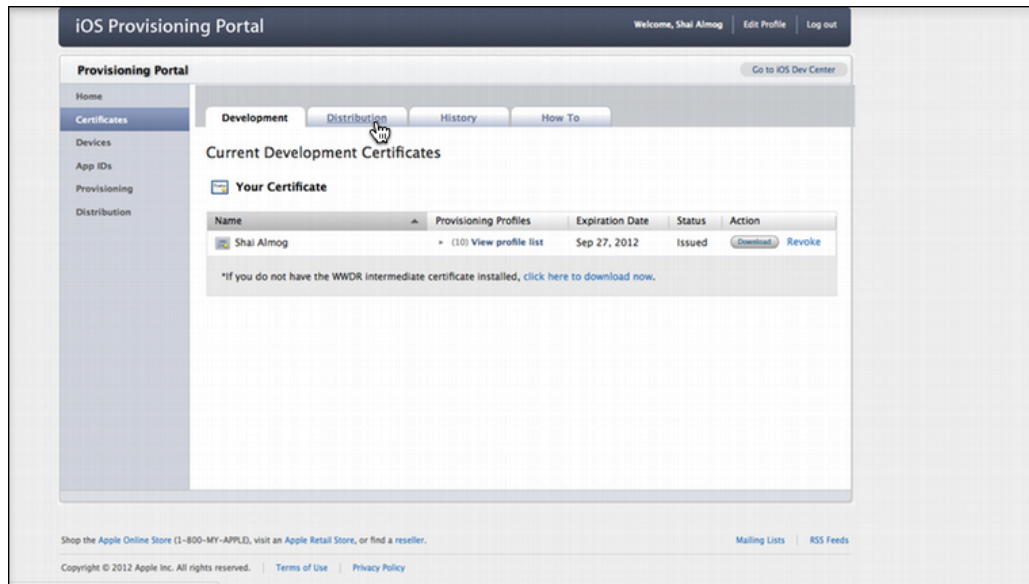
Provisioning Profile & Certificates

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.

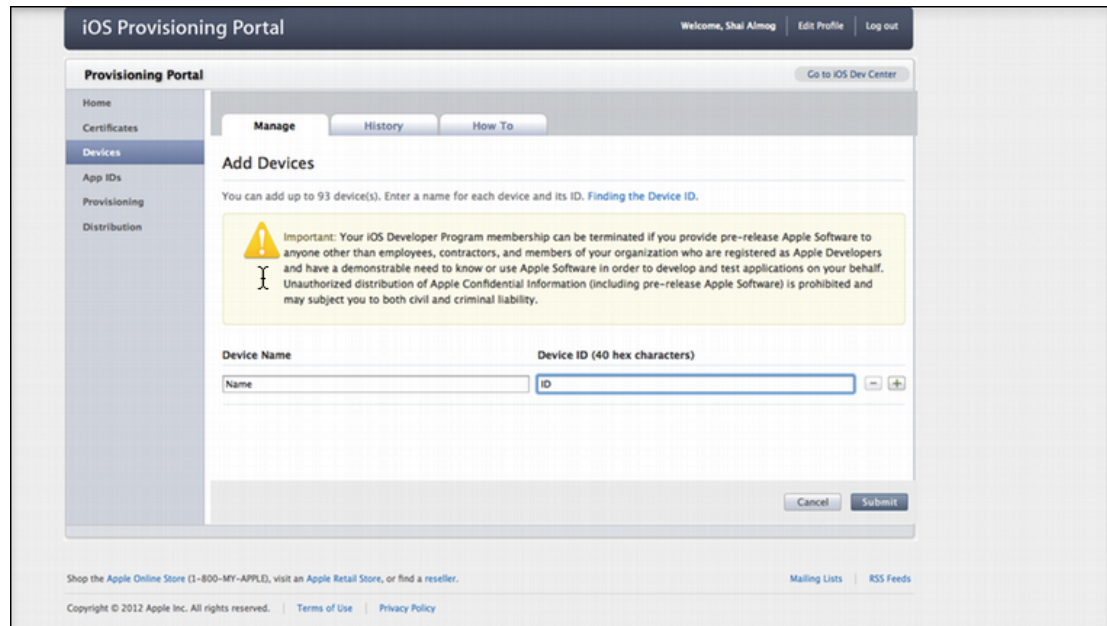
Start by logging in to the iOS-provisioning portal



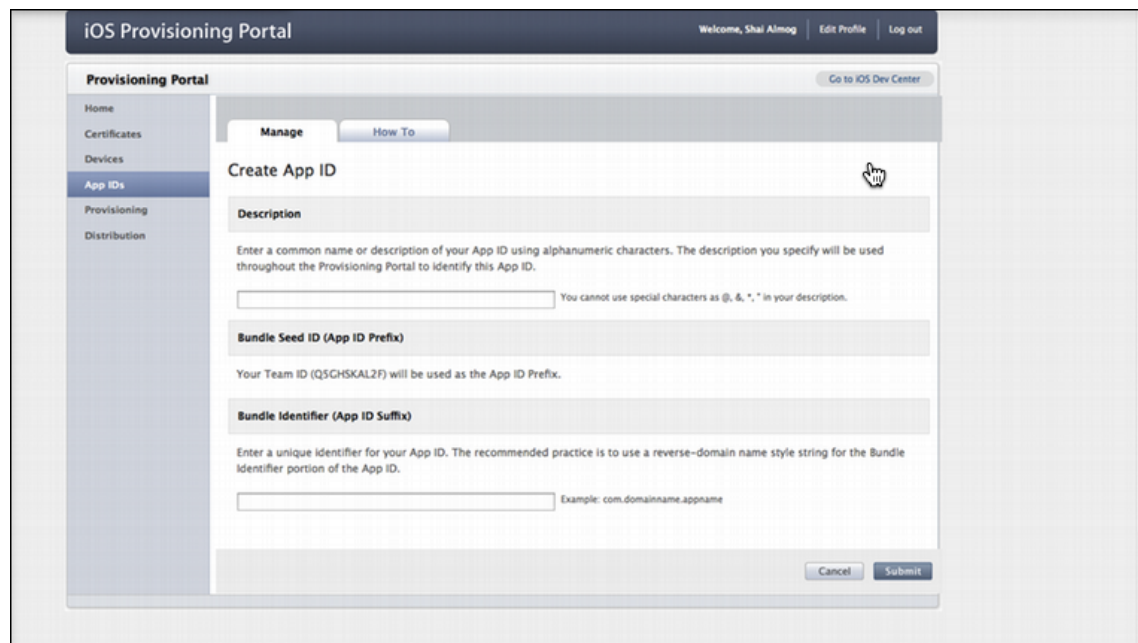
In the certificates section you can download your development and distribution certificates.



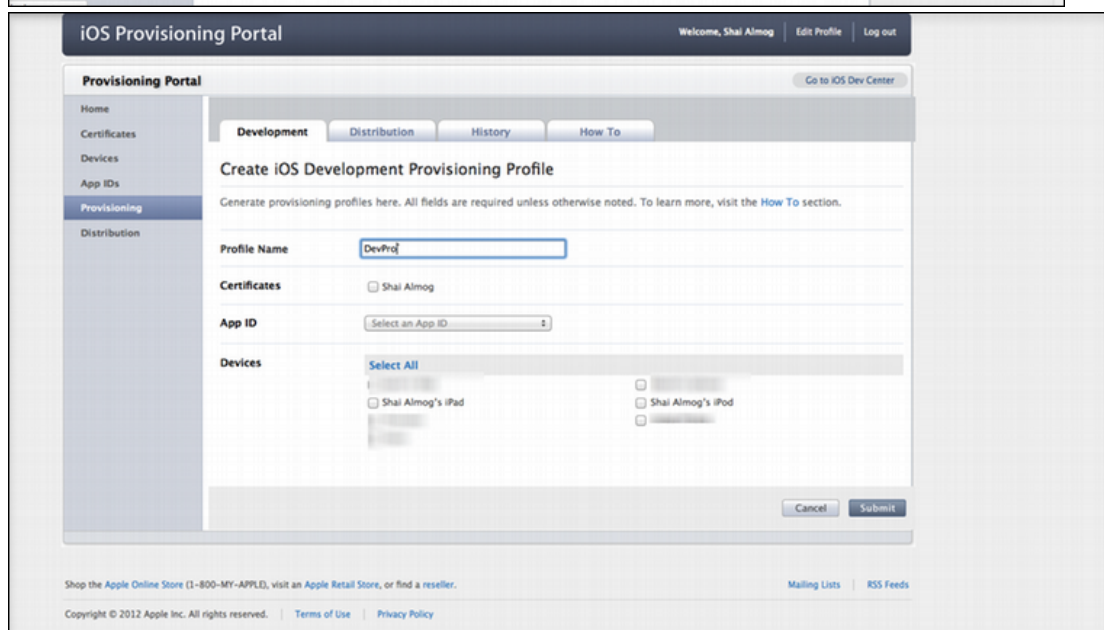
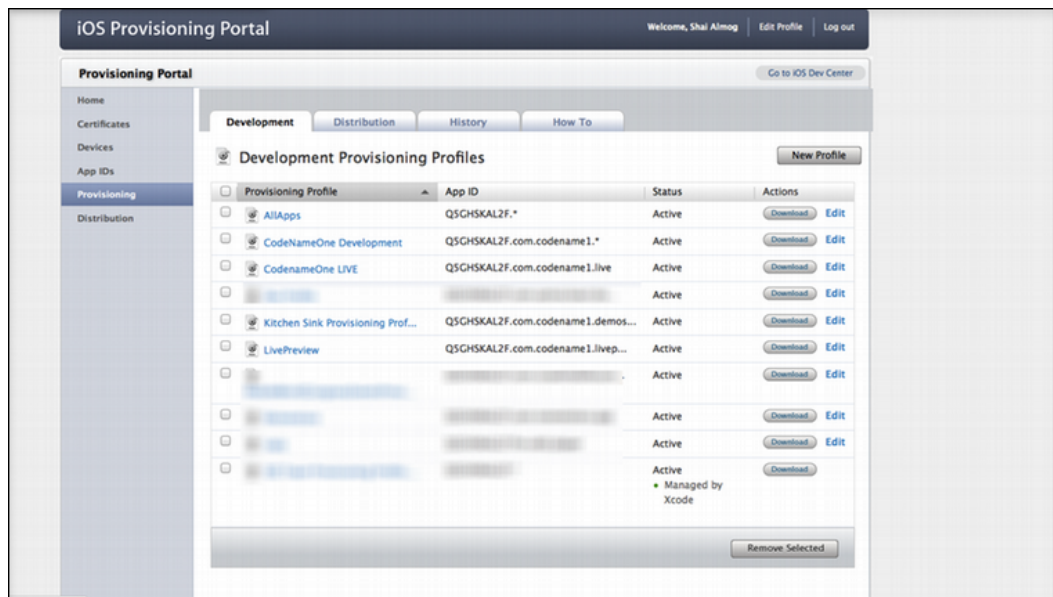
In the devices section add device ids for the development devices you want to support. Notice no more than 100 devices are supported!



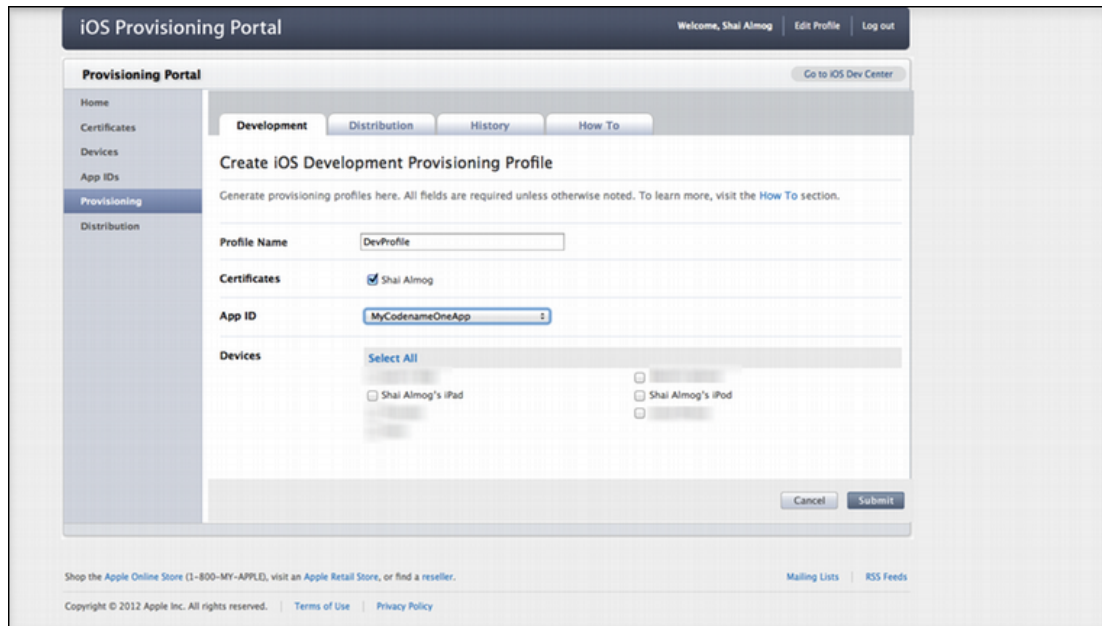
Create an application id; it should match the package identifier of your application perfectly!



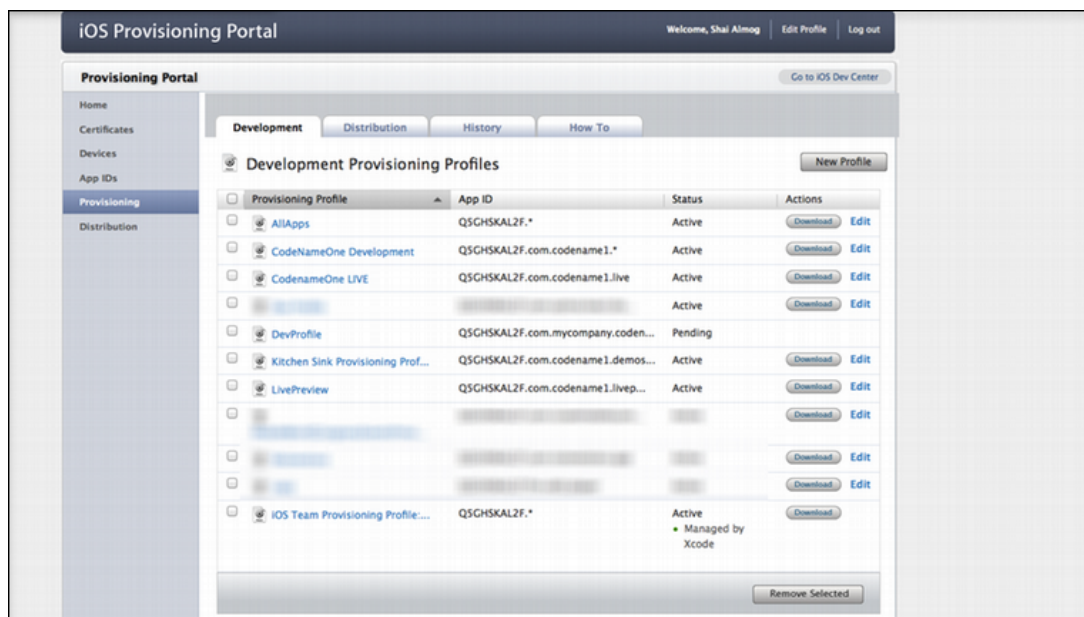
Create a provisioning profile for development, make sure to select the right app and make sure to add the devices you want to use during debug.



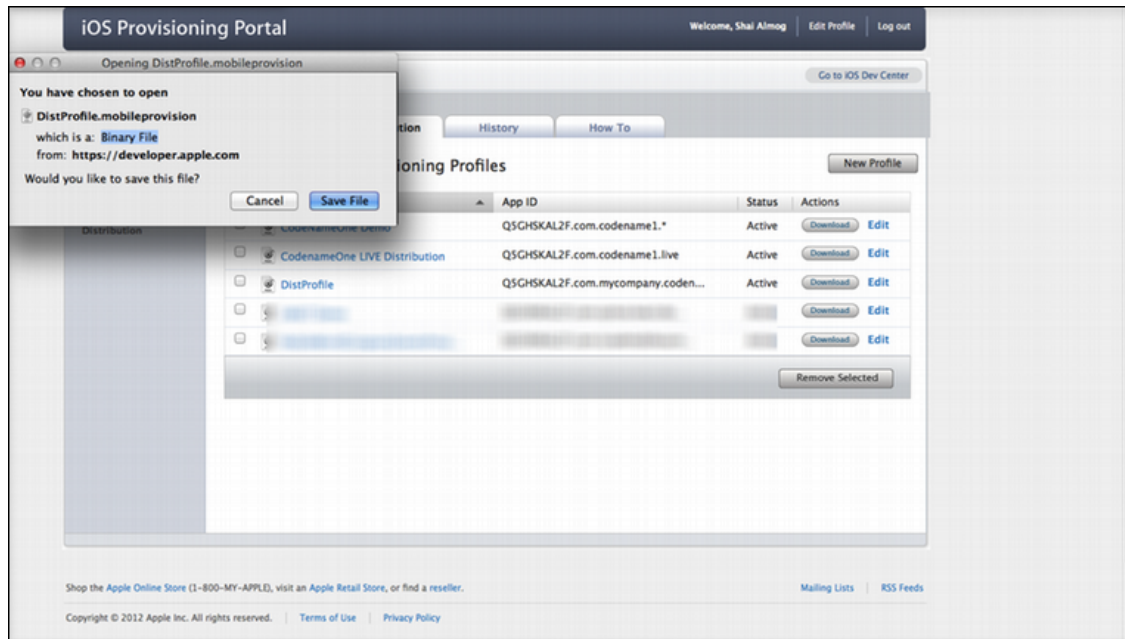
Refresh the screen to see the profile you just created and press the download button to download your development provisioning profile.



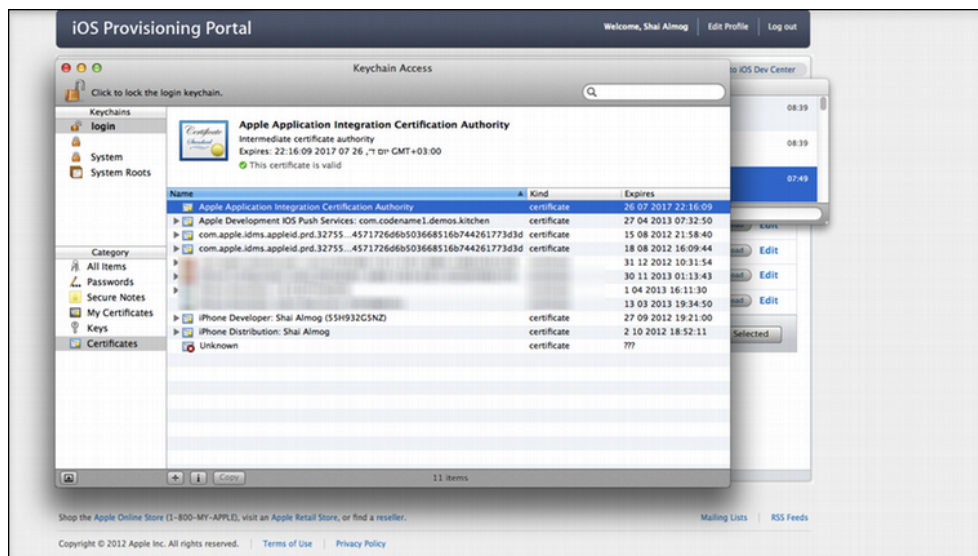
Create a distribution provisioning profile; it will be used when uploading to the app store. There is no need to specify devices here.



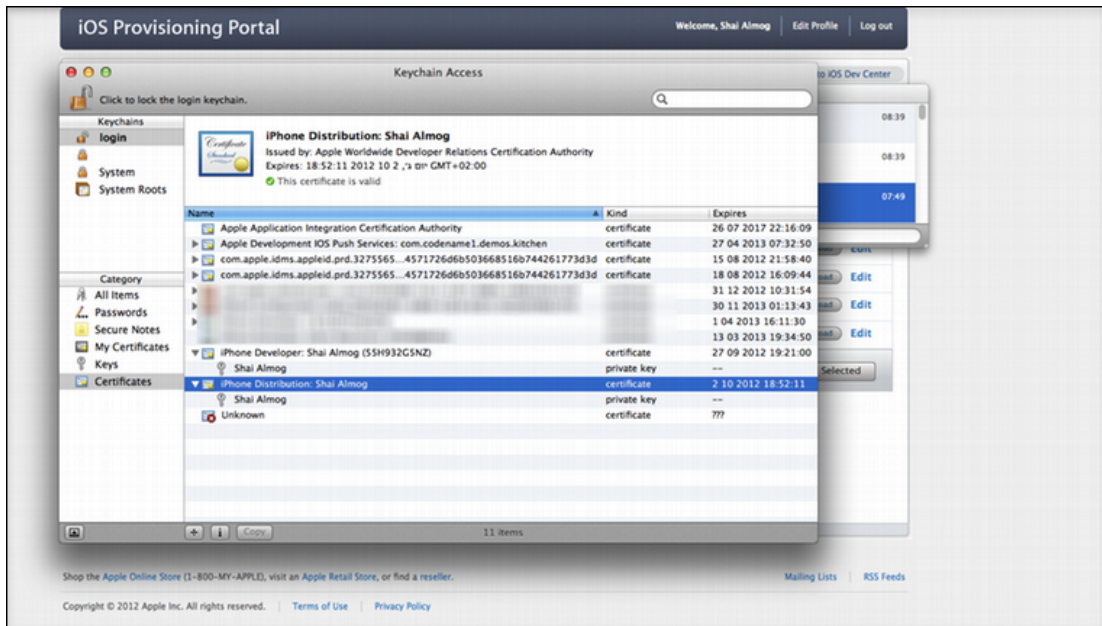
Download the distribution provisioning profile.



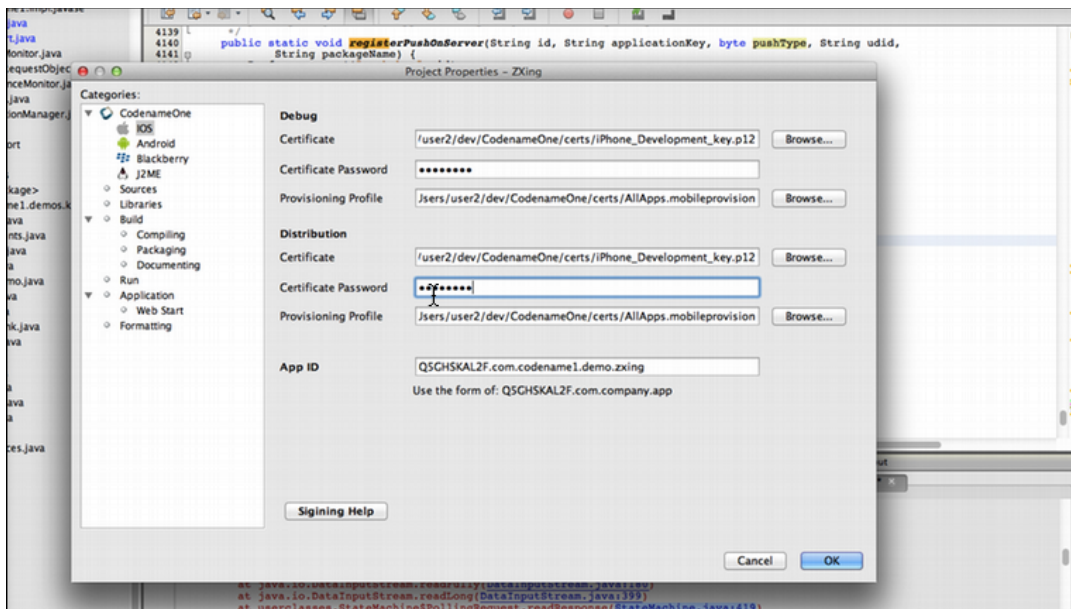
We can now import the cer files into the key chain tool on a Mac by double clicking the file, on Windows the process is slightly more elaborate



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

Push notification is only enabled for pro accounts in Codename One due to some of the server side infrastructure we need to maintain to support this feature.

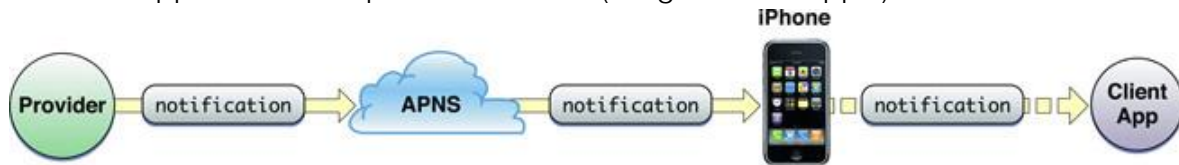
Push notification allows you to send a message to a device, usually a simple text message which is sent to the application or prompted to the user appropriately. When supported by the device it can be received even when the application isn't running and doesn't require polling which can drain the devices battery.

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 which isn't ideal.

Currently Codename One supports pushing to Google authorized Android devices: GCM (Google Cloud Messaging), to iOS devices: Push Notification & to blackberry devices.

For other devices we will fallback to polling the server in a given frequency, not an ideal solution by any means so Codename One provides the option to not fallback.

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

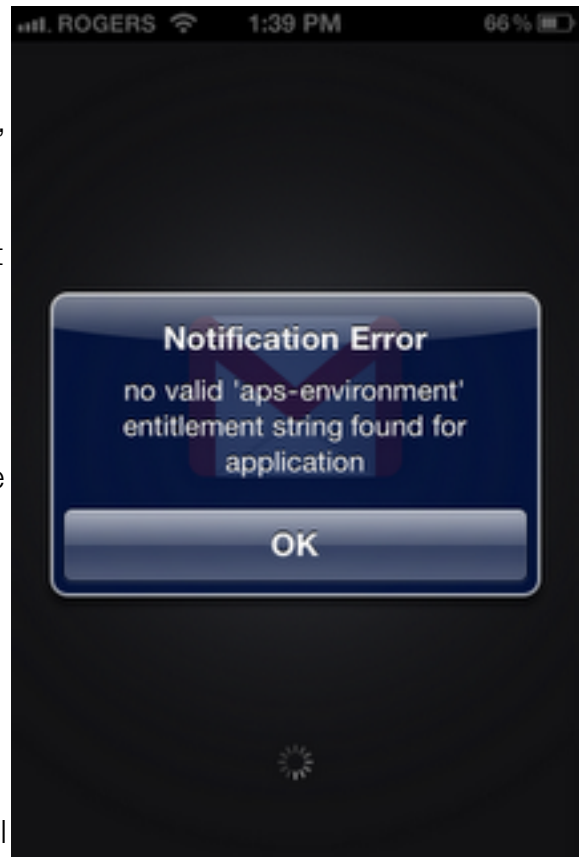


The "provider" is the server code that wishes to notify the device. It needs to ask Apple to push to a specific device and a specific client application. There are many complexities not mentioned here such as the need to have a push certificate or how the notification to APNS actually happens but the 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(metaData, 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 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 invoked with the device native push ID (not the id you should use. Once this method is invoked the device is ready to receive push messages.
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.



First there are several prerequisites you will need in order to get started with push:

- Android - you can find the full instructions from Google at <http://developer.android.com/google/gcm/gs.html>. You will need a project id that looks something like this: 4815162342.
You will also need the server key, which looks something like this:

AlzaSyATSw_rGeKnzKWULMGek7MDfEjRxJ1ybqo.

- iOS - You will need to create a provisioning profile that doesn't have the * element within it.

For that provisioning profile you will need to enable push and download a push certificate. Notice that this push certificate should be converted to a P12 file in the same manner we used in the [signing tutorials](#).

You will need the password for that P12 file as well.

You will need a distribution P12 and a testing P12.

Warning! The P12 for push is completely different from the one used to build your application, don't confuse them!

You will need to place the certificate on the web so our push server can access them, we often use dropbox to store our certificates for push.

- RIM - you need to register with RIM for credentials to use their push servers at https://developer.blackberry.com/devzone/develop/platform_services/push_overview.html.

Notice that initially you need to register for evaluation and later on move your app to production. This registration will trigger an email which you will receive that will contain all the information you will need later on. Such as your app ID, push URL (which during development is composed from your app ID), special password and client port number.

To start using push (on any platform) you will need to implement the PushCallback interface within your main class. The methods in that interface will be invoked when a push message arrives:

```
public class PushDemo implements PushCallback {
```

```
    private Form current;
```

```
    public void init(Object context) {  
    }
```

```
    public void start() {  
        if(current != null){  
            current.show();  
        }  
        return;  
    }
```

```
    }  
    new StateMachine("/theme");  
}  
  
public void stop() {  
    current = Display.getInstance().getCurrent();  
}  
  
public void destroy() {  
}  
  
public void push(String value) {  
    Dialog.show("Push Received", value, "OK", null);  
}  
  
public void registeredForPush(String deviceId) {  
    Dialog.show("Push Registered", "Device ID: " + deviceId + "\nDevice Key: " +  
Push.getDeviceKey() , "OK", null);  
}  
  
public void pushRegistrationError(String error, int errorCode) {  
    Dialog.show("Registration Error", "Error " + errorCode + "\n" + error, "OK", null);  
}  
}
```

You will then need to register to receive push notifications (its OK to call register every time the app loads) by invoking this code below (notice that the google project id needs to be passed to registration):

@Override

```
protected void onMain_RegisterForPushAction(Component c, ActionEvent event) {  
    Hashtable meta = new Hashtable();  
    meta.put(com.codename1.push.Push.GOOGLE_PUSH_KEY, findGoogleProjectId(c));  
    Display.getInstance().registerPush(meta, true);  
}
```

Sending the push is a more elaborate affair; we need to pass the elements to the push that is necessary for the various device types depending on the target device. If we send null as the destination device our message will be sent to all devices running our app.

However, if we use the device key which you can get via `Push.getDeviceKey()` you can target the device directly. **Notice that the device key is not the argument passed to the registration confirmation callback!**

Other than that we need to send various arguments whether this is a production push (valid for iOS where there is a strict separation between the debug and the production push builds) as well as the variables discussed above:

@Override

```
protected void onMain_SendPushAction(Component c, ActionEvent event) {
    String dest = findDestinationDevice(c).getText();
    if(dest.equals("")) {
        dest = null;
    }
    boolean prod = findProductionEnvironement(c).isSelected();
    String googleServerKey = findGoogleServerKey(c).getText();
    String iOSCURL = findIlosCert(c).getText();
    String iOSCPassword = findIlosPassword(c).getText();
    String bbPushURL = findBbPushURL(c).getText();
    String bbAppld = findBbAppld(c).getText();
    String bbPassword = findBbPassword(c).getText();
    String bbPort = findBbPort(c).getText();
    Push.sendPushMessage(findPushMessage(c).getText(), dest, prod, googleServerKey,
iOSCURL, iOSCPassword, bbPushURL, bbAppld, bbPassword, bbPort);
}
```

Unfortunately we aren't done yet!

We must define the following build arguments in the project properties:

ios.includePush=true

rim.includePush=true

rim.ignor_legacy=true

rim.pushPort=...

rim.pushAppld=...

rim.pushBpsURL=...

Once you define all of these push should work for all platforms.

You can perform the same task using our server API to send a push message directly from your server using the following web API. Notice that all arguments must be submitted as post!

URL = <https://codename-one.appspot.com/sendPushMessage>

Argument	Values	Description
device	numeric id or none	Optional, if omitted the message is sent to all devices
packageName	com.myapp...	The package name of your main class uniquely identifies your app. This is required even when submitting a device ID
email	x@y.com	The email address of the developer who built the app. This provides validation regarding the target of the push
type	numeric defaults to 1	The type of push, standard is 1 but there are additional types like 2 which is a silent push (nothing will be shown to the user) or 3 which combines a hidden payload with text visible to the user
auth	Google authorization key	Needed to perform the GCM push
certPassword	password	The password for the P12 push certificate for an iOS push
cert	URL	A url containing a downloadable P12 push certificate for iOS

body	Arbitrary message	The payload message sent to the device
production	true/false	Whether the push is sent to the iOS production or debug environment
burl	URL	The blackberry push URL
bbAppId		App ID sent by RIM
bbPass		Push password from RIM
bbPort		Push port from RIM

Appendix: Creating Codename One Maker Plugins

Codename One Maker (<http://www.codenameone.com/maker.html>) is an on device app maker tool that includes a drag and drop GUI builder and a host of other features.

Codename One developers can create plugins to extend the functionality of Maker and leverage the full scope of Codename One's capabilities within this easy to use tool.

This effectively turns Maker into a tool that is limited only by your imagination and not by some arbitrary constraints we put forth.

One thing to keep in mind before proceeding: A plugin can't be previewed within Maker. The user will be able to pass arguments (settings) to the plugin and see it within the app but he won't be able to launch it (we can't dynamically download code according to store EULA's and also some technical limitations).

So how does this work?

A Codename One Maker plugin is really just 2 files an XML file and a CN1Lib file. The XML file describes the requirements of the plugin from the user and gives us basic details about the plugin, only the XML file is ever downloaded to the device.

When the user sends the build, the details are sent to the server. The server downloads and incorporates the CN1Lib file, which can include native code or any other capabilities. Its compiled much in the same way as any Codename One library in that regard. When the built application on the device the user can see the plugin in action.

So how do we build a hello world plugin?

I created a simple Twitter feed plugin just to show the basic principals (see the full project at the bottom of this post). To start off we need to create a new Library Project in NetBeans (sorry currently Eclipse doesn't support Library projects although its theoretically possible to work with it to build these projects).

We can remove the hello world code and create a new package with our company and name of the plugin, and then we need to implement the plugin. The plugin is a class that derives from MakerPlugin here is the simple Twitter plugin from the code below:

```
package com.mycompany.plugin.mytweets;
```

```
import com.codename1.components.InfiniteProgress;
import com.codename1.components.MultiButton;
import com.codename1.io.ConnectionRequest;
import com.codename1.io.JSONParser;
import com.codename1.io.NetworkManager;
import com.codename1.maker.MakerPlugin;
import com.codename1.ui.Button;
import com.codename1.ui.Container;
import com.codename1.ui.Display;
import com.codename1.ui.Label;
import com.codename1.ui.TextArea;
import com.codename1.ui.events.ActionEvent;
import com.codename1.ui.events.ActionListener;
import com.codename1.ui.layouts.BorderLayout;
import com.codename1.ui.layouts.BoxLayout;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Hashtable;
import java.util.Vector;
```

```
/**
```

```
 * This is simple hello world maker plugin showing how we can get arguments and then
 * use Codename One
```

```
 * to provide all sorts of functionality
```

```
 *
```

```
 * @author Shai Almog
```

```
 */
```

```
public class Plugin extends MakerPlugin {
```

```
    @Override
```

```
    public String getPackageName() {
```

```
        return "com.mycompany.plugin.mytweets";
```

```
}
```

```
@Override
```

```
public String getDeveloperEmailId() {  
    return "youremailhere@x.com";  
}
```

```
@Override
```

```
public String getPluginName() {  
    return "My Tweets";  
}
```

```
@Override
```

```
public Container createEmbeddedUI() {  
    final Container tweets = new Container(new BorderLayout(BoxLayout.Y_AXIS));  
    tweets.setScrollableY(true);  
    ConnectionRequest t = new ConnectionRequest() {  
        private Hashtable h;  
        @Override  
        protected void readResponse(InputStream input) throws IOException {  
            JSONParser jp = new JSONParser();  
            h = jp.parse(new InputStreamReader(input));  
        }  
    }
```

```
@Override
```

```
protected void postResponse() {  
    Vector v = (Vector)h.get("results");  
    tweets.removeAll();  
    for(Object currentObj : v) {  
        Hashtable current = (Hashtable)currentObj;  
        String user = (String)current.get("from_user_name");  
        String date = (String)current.get("created_at");  
        String text = (String)current.get("text");  
        String url = "";  
        int indi = text.indexOf("http://t.co/");  
        if(indi > -1) {  
            int last = text.indexOf(' ', indi);  
            if(last == -1) {
```

```
        last = last = text.length();
    }
    url = text.substring(indi, last);
}
final String finalURL = url;
Container entry = new Container(new BorderLayout());
entry.setUIID("MultiButton");
TextArea t = new TextArea(text);
t.setEditable(false);
t.setUIID("MultiLine1");
Button lead = new Button();
lead.setUIID("Label");
entry.addComponent(BorderLayout.CENTER, t);
entry.addComponent(BorderLayout.EAST, lead);
Container south = new Container(new BorderLayout());
entry.addComponent(BorderLayout.SOUTH, south);
Label dateText = new Label(date);
dateText.setUIID("MultiLine2");
Label name = new Label(user);
name.setUIID("MultiLine3");
south.addComponent(BorderLayout.CENTER, dateText);
south.addComponent(BorderLayout.EAST, name);
entry.setLeadComponent(lead);
tweets.addComponent(entry);
lead.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        if(finalURL != null && finalURL.length() > 0) {
            Display.getInstance().execute(finalURL);
        }
    }
});
}
tweets.animateLayout(350);
}

};
t.setUrl("http://search.twitter.com/search.json");
```

```

        t.setPost(false);
        t.setContentType("application/json");
        t.addRequestHeader("Accept", "application/json");

        // we pass arguments to the plugin via the meta data, the argument types
        // are declared in the plugin XML
        String user = (String)getMetaData().get("user");
        if(user == null || user.length() == 0) {
            user = "@codename-one";
        } else {
            if(!user.startsWith("@")) {
                user = "@" + user;
            }
        }
        t.addArgument("q", user);
        tweets.addComponent(new InfiniteProgress());
        NetworkManager.getInstance().addToQueue(t);
        return tweets;
    }
}

```

Notice the overridden methods above are a part of the plugin interface, once we are in the plugin itself we can just write any Codename One code that we want although keep in mind that I try not to block the execution thread... Otherwise I might create an unpleasant experience when building a tabs based application.

Also notice that I enable scrollability since the parent form won't be accessible we disabled scrolling there (to avoid nested scrolling issues), if you need scrolling you need to explicitly declare it.

Its probably obvious but bares stating that the plugin class must be public, have a no argument constructor (or no constructor which is the same thing) and mustn't be abstract.

You will also need one more file which is the xml descriptor file, in my case its twitter.mplugin (in the root of the downloaded file) which you can see right here:

```

<?xml version="1.0" encoding="UTF-8"?>
<plugin makerVer="0.4" name="My Tweets"
package="com.mycompany.plugin.mytweets"
developer="emailUsedForRegistration@domain.com"

```

```
    developerName="My Company Name" version="1.0"
help="http://website.com/help.html"
cn1lib="https://dl.dropboxusercontent.com/u/57067724/cn1/HelloCodenameOnePlugin.cn1lib" class="Plugin">
    <arg name="user" display="Twitter User" type="string"
default="@codename-one" details="The twitter user handle with or without
the @ sign" />;
</plugin>
```

Notice several things:

- We declare the maker version, this is crucial since if someone hasn't updated Maker on the device and your plugin expects a specific version... It should fail to install.
- The package name and the class name must match exactly your class since that is how we generate the plugin calls on the server!
- The cn1lib entry MUST point at an absolute URL where the plugin can be downloaded from, the download only happens on the build server so the build will fail if the file isn't accessible.
- You can define as many arguments as you want but currently all of them must be strings, we are working on adding more options in the future.

Appendix: 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.

Appendix: Casual Game Programming

While game developers have traditionally used C/OpenGL to get every bit of performance out of a device, Java offers a unique opportunity for casual game developers. In this article we will build a simple card game in Java that can run unchanged on iOS, Android and RIM devices.

Casual games are often the most influential games of all, they cross demographics such as the ubiquitous solitaire or even the chart topping Angry birds. Putting them in the same game category as 3D FPS games doesn't always make sense.

Yes, framerates are important but ubiquity, social connectivity & gameplay are even more important for this sub genre of the game industry. The mobile aspect highlights this point further, the way app stores are built releasing often puts your game at an advantage over its competitor's. Yet releasing to all platforms and all screen sizes becomes an issue soon enough.

Java has been familiar for mobile game developers for quite some time for better or worse. Despite all its issues J2ME was a pretty amazing tool considering the fact that it was last updated in 2004 (iPhone was introduced in 2007), game developers were able to squeeze quite a lot of power from that very limited platform. Many of these early J2ME game developers have since moved to Android game development which is also growing rapidly. In this article I'm going to go over the process of writing a simple card game for iOS, Android and RIM devices using Codename One which is an open source platform for mobile application development in Java. The value of using Codename One here is mostly in its ability to target iOS which has a far better retention rate and revenue stream than Android, although the true value is in ubiquity and the ability of our users to share the application. Codename One itself is not a game development platform and is designed mostly as an application development platform. However, some developers used the tool to build casual games.

Typically a game is comprised of a game loop which updates UI status based on game time and renders the UI. However, with casual games constantly rendering is redundant and with mobile games it could put a major drain on the battery life. Instead we will use components to build the game elements and let Codename One do the rendering for us.

The Game

We will create a poker game for 1 player that doesn't include the betting process or any of the complexities such as AI (you can see the game running on the simulator here:

www.youtube.com/watch?v=4IQGBT3VsSQ), card evaluation or validation. This allows us to fit the whole source code in 270 lines of code (more due to comments). I also chose to simplify the UI for touch devices only, technically it would be pretty easy to add keypad support but it would complicate the code and require additional designs (for focus states). The game consists of two forms: Splash screen and the main game UI.

Codename One has a GUI builder that allows drag and drop development, however we won't be using it since its really difficult to convey GUI builder activity in an article.

Getting Started

Make sure to select the Hello World (Manual) option since we don't want to use the GUI builder.

Also make sure to enter a valid package name pointing to a domain you own in the common Java convention, this is important since it would be hard to change the name later on. The package name is used as a unique identifier in most app stores and once the app is published it can't be changed!

Once you clicked finish you should have a new project and you should be able to write the game interaction code. You can press run to see the hello world app and you will notice the project also has a theme.res file which includes potential project resources. But first lets go over the issue of dealing with screen resolutions.

Handling Multiple Device Resolutions

In mobile device programming every pixel is crucial because of the small size of the screen, however we can't shrink down our graphics too much because it needs to be "finger friendly" (big enough for a finger) and readable. There is great disparity in the device world, even within the iOS family the current iPad has more than twice the screen density of the iPad mini. This means that an image that looks good on the iPad mini will seem either small or very pixelated on an iPad, on the other hand an image that looks good on the iPad would look huge (and take up too much RAM) on the iPad mini. The situation is even worse when dealing with phones and Android devices.

Thankfully there are solutions, such as using multiple images for every density (DPI).

However, this is tedious for developers who need to scale the image and copy it every time for every resolution. Codename One has a feature called Multimage which implicitly

scales the images to all the resolutions on the desktop and places them within the res file, in runtime you will get the image that matches your devices density.

There is a catch though... Multimage is designed for applications where we want the density to determine the size. So an iPad will have the same density as an iPhone since both share the same amount of pixels per inch. This makes sense for an app since the images will be big enough to touch and clear. Furthermore, since the iPad screen is larger more data will fit on the screen!

However, game developers have a different constraint when it comes to game elements. In the case of a game we want the images to match the device resolution and take up as much screen real estate as possible, otherwise our game would be constrained to a small portion of the tablet and look small. There is a solution though, we can determine our own DPI level when loading resources and effectively force a DPI based on screen resolution only when working with game images!

To work with such varied resolutions/DPI's and potential screen orientation changes we need another tool in our arsenal: layout managers.

If you are familiar with AWT/Swing this should be pretty easy, Codename One allows you to codify the logic that flows Components within the UI. We will use the layout managers to facilitate that logic and preserve the UI flow when the device is rotated.

Resources

To save some time/effort I suggest using the ready made resource files linked in the On The Web section below. I suggest skipping this section and moving on to the code, however for completeness here is what I did to create these resources:

You will need a gamedata.res file that contains all the 52 cards as multi images using the naming convention of 'rank suite.png' example: 10c.png (10 of clubs) or ad.png (Ace of diamonds).

To accomplish this I created 52 images of roughly 153x217 pixels for all the cards then used the designer tool and selected "Quick Add Multimages" from the menu. When prompted I selected HD resolution. This effectively created 52 multi-images for all relevant resolutions.

I also modified the default theme that came in the application in small ways to create the white over green color scheme, I opened it in the designer tool by double clicking it and selected the theme.

I then pressed Add and added a Form entry with background NONE, background color 6600 and transparency 255.

I added a Label style with transparency 0 and foreground 255 and then copied the style to pressed/selected (since its applied to buttons too).

I did the same for the SplashTitle/SplashSubtitle but there I also set the alignment to center, the font to bold and in the case of SplashTitle to Large font as well.

The Splash Screen

The first step is creating the splash animation as you can see in the screenshots in figure 2.

Figure 2: Animation stages for the splash screen opening animation



The animation in the splash screen and most of the following animations are achieved using the simple tool of layout animations. In Codename One components are automatically arranged into position using layout managers, however this isn't implicit unless the device is rotated. A layout animation relies on this fact, it allows you to place components in a position (whether by using a layout manager or by using `setX/setY`) then invoke the layout animation code so they will slide into their "proper" position based on the layout manager rules.

You can see how I achieved the splash screen animation of the cards sliding into place in Listing 1 within the `showSplashScreen()` method. After we change the layout to a box X layout we just invoke `animateHierarchy` to animate the cards into place.

Notice that we use the `callSerially` method to start the actual animation. This call might not seem necessary at first until you try running the code on iOS. The first screen of the UI is very important for the iOS port which uses a screenshot to speed startup (this is a feature of the native iOS platform which as of this writing requires 7 screenshots in different resolutions/orientations for every application, Codename One generates those shots automatically and adds them to your app). If we won't have this `callSerially` invocation the screenshot rendering process will not succeed and the animation will stutter.

We also have a cover transition defined here; it's just a simple overlay when moving from one form to another.

The Game UI

Initially when entering the game form we have another animation where all the cards are laid out as you can see in Figure 3. We then have a long sequence of animation where the cards unify into place to form a pile (with a cover background falling on top) after which dealing begins and cards animate to the rival (with back only showing) or to you with the face showing. Then the instructions to swap cards fade into place.

Figure 3: Game form startup animation and deal animation



This animation is really easy to accomplish although it does have several stages. In the first stage we layout the cards within a grid layout (13x4), then when the animation starts (see the `UITimer` code within `showGameUI()`) we just change the layout to a layered layout, add the back card (so it will come out on top based on z-ordering) and invoke `animateLayout`. Notice that here we use `animateLayoutAndWait`, which effectively blocks the calling thread until the animation is completed. This is a VERY important and tricky subject!

Codename One is for the most part a single threaded API, it supports working on other threads but it is your responsibility to invoke everything on the EDT (Event Dispatch Thread). Since the EDT does the entire rendering, events etc. if you block it you will effectively stop Codename One in its place! However, there is a trick: `invokeAndBlock` is a feature that allows you to stop the EDT and do stuff then restore the EDT without “really” stopping it. Its tricky, I won’t get into this in this article (this subject deserves an article of its own) but the gist of it is that you can’t just invoke `Thread.sleep()` in a Codename One application (at least not on the EDT) but you can use clever methods such as `Dialog.show()`, `animateLayoutAndWait` etc. and they will block the EDT for you. This is really convenient since you can just write code serially without requiring event handling for every single feature.

Now that we got that out of the way, the rest of the code is clearer. Now we understand that `animateLayoutAndWait` will literally wait for the animation to complete and the next lines can do the next animation. Indeed after that we invoke the `dealCard` method that hands the cards to the players. This method is also blocking (using `and wait` methods internally) it also marks the cards as draggable and adds that drag and drop logic which we will later use to swap cards.

Last but not least in the animation department, we use a method called `replace` to fade in a component using a transition.

To handle the dealing we added an action listener to the deck button, this action listener is invoked when the cards are dealt and that completes the game.

Summary

It is really easy to create a functional and attractive mobile game in Java and have it work on all devices. Adding networking and social interaction would be relatively easy and the place where using Java really shines. While normally game developers deal with graphics systems and game loops, you can still create a very attractive casual game while staying within the comfort zone of components. The value here is in easy support for orientation changes and various resolutions.

Developing this game demo took me one afternoon with most of the time being spent at cutting the card images to the right size, I hope you will find this useful and join us on the Codename One discussion forum with questions/comments.

```
package com.codename1.demo.poker;

import com.codename1.ui.Button;
import com.codename1.ui.Component;
import com.codename1.ui.Container;
import com.codename1.ui.Dialog;
import com.codename1.ui.Display;
import com.codename1.ui.Form;
import com.codename1.ui.Image;
import com.codename1.ui.Label;
import com.codename1.ui.TextArea;
import com.codename1.ui.animations.CommonTransitions;
import com.codename1.ui.events.ActionEvent;
import com.codename1.ui.events.ActionListener;
import com.codename1.ui.geom.Dimension;
import com.codename1.ui.layouts.BorderLayout;
import com.codename1.ui.layouts.BoxLayout;
import com.codename1.ui.layouts.FlowLayout;
import com.codename1.ui.layouts.GridLayout;
import com.codename1.ui.layouts.LayeredLayout;
import com.codename1.ui.plaf.UIManager;
```

```
import com.codename1.ui.util.Resources;
import com.codename1.ui.util.UTimer;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
```

```
/**
```

```
 * Demo app showing how a simple poker card game can be written using Codename One,
this
```

```
 * demo was developed for an <a href="http://www.sdjournal.org">SD Journal</a> article.
```

```
 * @author Shai Almog
```

```
*/
```

```
public class Poker {
```

```
    private static final char SUITE_SPADE = 's';
```

```
    private static final char SUITE_HEART = 'h';
```

```
    private static final char SUITE_DIAMOND = 'd';
```

```
    private static final char SUITE_CLUB = 'c';
```

```
    private Resources cards;
```

```
    private Form current;
```

```
    private final static Card[] deck;
```

```
    static {
```

```
        // we initialize constant card values that will be useful later on in the game
```

```
        deck = new Card[52];
```

```
        for(int iter = 0 ; iter < 13 ; iter++) {
```

```
            deck[iter] = new Card(SUITE_SPADE, iter + 2);
```

```
            deck[iter + 13] = new Card(SUITE_HEART, iter + 2);
```

```
            deck[iter + 26] = new Card(SUITE_DIAMOND, iter + 2);
```

```
            deck[iter + 39] = new Card(SUITE_CLUB, iter + 2);
```

```
        }
```

```
    }
```

```
/**
```

```
 * We use this method to calculate a "fake" DPI based on screen resolution rather than
its actual DPI
```

** this is useful so we can have large images on a tablet*
*/

```
private int calculateDPI() {
    int pixels = Display.getInstance().getDisplayHeight() *
Display.getInstance().getDisplayWidth();
    if(pixels > 1000000) {
        return Display.DENSITY_HD;
    }
    if(pixels > 340000) {
        return Display.DENSITY_VERY_HIGH;
    }
    if(pixels > 150000) {
        return Display.DENSITY_HIGH;
    }
    return Display.DENSITY_MEDIUM;
}
```

/**

** This method is invoked by Codename One once when the application loads*
*/

```
public void init(Object context) {
    try{
        // after loading the default theme we load the card images as a resource with
        // a fake DPI so they will be large enough. We store them in a resource rather
        // than as files so we can use the Multimage functionality
        Resources theme = Resources.openLayered("/theme");

        UIManager.getInstance().setThemeProps(theme.getTheme(theme.getThemeResourceNames()[0]));
        cards = Resources.open("/gamedata.res", calculateDPI());
    } catch(IOException e) {
        e.printStackTrace();
    }
}
```

/**

** This method is invoked by Codename One once when the application loads and when it is restarted*

```
*/
public void start() {
    if(current != null){
        current.show();
        return;
    }
    showSplashScreen();
}

/**
 * The splash screen is relatively bare bones. Its important to have a splash screen for
 * iOS
 * since the build process generates a screenshot of this screen to speed up perceived
 * performance
 */
public void showSplashScreen() {
    final Form splash = new Form();

    // a border layout places components in the center and the 4 sides.
    // by default it scales the center component so here we configure
    // it to place the component in the actual center
    BorderLayout border = new BorderLayout();

    border.setCenterBehavior(BorderLayout.CENTER_BEHAVIOR_CENTER_ABSOLUTE);
    splash.setLayout(border);

    // by default the form's content pane is scrollable on the Y axis
    // we need to disable it here
    splash.setScrollable(false);
    Label title = new Label("Poker Ace");

    // The UIID is used to determine the appearance of the component in the theme
    title.setUIID("SplashTitle");
    Label subtitle = new Label("By Codename One");
    subtitle.setUIID("SplashSubTitle");

    splash.addComponent(BorderLayout.NORTH, title);
    splash.addComponent(BorderLayout.SOUTH, subtitle);
}
```



```
Label as = new Label(cards.getImage("as.png"));
Label ah = new Label(cards.getImage("ah.png"));
Label ac = new Label(cards.getImage("ac.png"));
Label ad = new Label(cards.getImage("ad.png"));
```

// a layered layout places components one on top of the other in the same dimension,
it is

// useful for transparency but in this case we are using it for an animation

```
final Container center = new Container(new LayeredLayout());
center.addComponent(as);
center.addComponent(ah);
center.addComponent(ac);
center.addComponent(ad);
```

```
splash.addComponent(BorderLayout.CENTER, center);
```

```
splash.show();
```

```
splash.setTransitionOutAnimator(CommonTransitions.createCover(CommonTransitions.SLIDE_VERTICAL, true, 800));
```

// postpone the animation to the next cycle of the EDT to allow the UI to render fully
once

```
Display.getInstance().callSerially(new Runnable() {
    public void run() {
```

// We replace the layout so the cards will be laid out in a line and animate the
hierarchy

// over 2 seconds, this effectively creates the effect of cards spreading out

```
center.setLayout(new BoxLayout(BoxLayout.X_AXIS));
center.setShouldCalcPreferredSize(true);
splash.getContentPane().animateHierarchy(2000);
```

// after showing the animation we wait for 2.5 seconds and then show the game
with a nice

// transition, notice that we use UI timer which is invoked on the Codename One
EDT thread!

```
new UITimer(new Runnable() {
    public void run() {
```

```

        showGameUI();
    }
    }).schedule(2500, false, splash);
}
});
}

/**
 * This is the method that shows the game running, it is invoked to start or restart the
game
 */
private void showGameUI() {
    // we use the java.util classes to shuffle a new instance of the deck
    final List<Card> shuffledDeck = new ArrayList<Card>(Arrays.asList(deck));
    Collections.shuffle(shuffledDeck);

    final Form gameForm = new Form();

    gameForm.setTransitionOutAnimator(CommonTransitions.createCover(CommonTransition
s.SLIDE_VERTICAL, true, 800));
    Container gameFormBorderLayout = new Container(new BorderLayout());

    // while flow layout is the default in this case we want it to center into the middle of
the screen
    FlowLayout fl = new FlowLayout(Component.CENTER);
    fl.setValign(Component.CENTER);
    final Container gameUpperLayer = new Container(fl);
    gameForm.setScrollable(false);

    // we place two layers in the game form, one contains the contents of the game and
another one on top contains instructions
    // and overlays. In this case we only use it to write a hint to the user when he needs
to swap his cards
    gameForm.setLayout(new LayeredLayout());
    gameForm.addComponent(gameFormBorderLayout);
    gameForm.addComponent(gameUpperLayer);

    // The game itself is comprised of 3 containers, one for each player containing a grid

```

of 5 cards (grid layout

// divides space evenly) and the deck of cards/dealer. Initially we show an animation where all the cards

// gather into the deck, that is why we set the initial deck layout to show the whole deck 4x13

```
final Container deckContainer = new Container(new GridLayout(4, 13));
```

```
final Container playerContainer = new Container(new GridLayout(1, 5));
```

```
final Container rivalContainer = new Container(new GridLayout(1, 5));
```

// we place all the card images within the deck container for the initial animation

```
for(int iter = 0 ; iter < deck.length ; iter++) {
```

```
    Label face = new Label(cards.getImage(deck[iter].getFileName()));
```

// containers have no padding or margin this effectively removes redundant spacing

```
    face.setUIID("Container");
```

```
    deckContainer.addComponent(face);
```

```
}
```

// we place our cards at the bottom, the deck at the center and our rival on the north

```
gameFormBorderLayout.addComponent(BorderLayout.CENTER, deckContainer);
```

```
gameFormBorderLayout.addComponent(BorderLayout.NORTH, rivalContainer);
```

```
gameFormBorderLayout.addComponent(BorderLayout.SOUTH, playerContainer);
```

```
gameForm.show();
```

// we wait 1.8 seconds to start the opening animation, otherwise it might start while the transition is still running

```
new UITimer(new Runnable() {
```

```
    public void run() {
```

```
        // we add a card back component and make it a drop target so later players
```

```
        // can drag their cards here
```

```
        final Button cardBack = new Button(cards.getImage("card_back.png"));
```

```
        cardBack.setDropTarget(true);
```

```
        // we remove the button styling so it doesn't look like a button by using setUIID.
```

```
        cardBack.setUIID("Label");
```

```
        deckContainer.addComponent(cardBack);
```

```
// we set the layout to layered layout which places all components one on top of  
the other then animate
```

```
// the layout into place, this will cause the spread out deck to "flow" into place  
// Notice we are using the AndWait variant which will block the event dispatch  
thread (legally) while
```

```
// performing the animation, normally you can't block the dispatch thread (EDT)  
deckContainer.setLayout(new LayeredLayout());  
deckContainer.animateLayoutAndWait(3000);
```

```
// we don't need all the card images/labels in the deck, so we place the card  
back
```

```
// on top then remove all the other components  
deckContainer.removeAll();  
deckContainer.addComponent(cardBack);
```

```
// Now we iterate over the cards and deal the top card from the deck to each  
player
```

```
for(int iter = 0 ; iter < 5 ; iter++) {  
    Card currentCard = shuffledDeck.get(0);  
    shuffledDeck.remove(0);  
    dealCard(cardBack, playerContainer,  
cards.getImage(currentCard.getFileName()), currentCard);  
    currentCard = shuffledDeck.get(0);  
    shuffledDeck.remove(0);  
    dealCard(cardBack, rivalContainer, cards.getImage("card_back.png"),  
currentCard);  
}
```

```
// After dealing we place a notice in the upper layer by fade in. The trick is in  
adding a blank component
```

```
// and replacing it with a fade transition  
TextArea notice = new TextArea("Drag cards to the deck to swap\ntap the  
deck to finish");  
notice.setEditable(false);  
notice.setFocusable(false);  
notice.setUIID("Label");  
notice.getUnselectedStyle().setAlignment(Component.CENTER);
```

```

gameUpperLayer.addComponent(notice);
gameUpperLayer.layoutContainer();

// we place the notice then remove it without the transition, we need to do this
since a text area
// might resize itself so we need to know its size in advance to fade it in.
Label temp = new Label(" ");
temp.setPreferredSize(new Dimension(notice.getWidth(), notice.getHeight()));
gameUpperLayer.replace(notice, temp, null);

gameUpperLayer.layoutContainer();
gameUpperLayer.replace(temp, notice, CommonTransitions.createFade(1500));

// when the user taps the card back (the deck) we finish the game
cardBack.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        // we clear the notice text
        gameUpperLayer.removeAll();

        // we deal the new cards to the player (the rival never takes new cards)
        while(playerContainer.getComponentCount() < 5) {
            Card currentCard = shuffledDeck.get(0);
            shuffledDeck.remove(0);
            dealCard(cardBack, playerContainer,
cards.getImage(currentCard.getFileName()), currentCard);
        }

        // expose the rivals deck then offer the chance to play again...
        for(int iter = 0 ; iter < 5 ; iter++) {
            Button cardButton = (Button)rivalContainer.getComponentAt(iter);

            // when creating a card we save the state into the component itself
            which is very convenient
            Card currnetCard = (Card)cardButton.getClientProperty("card");
            Label l = new Label(cards.getImage(currnetCard.getFileName()));
            rivalContainer.replaceAndWait(cardButton, l,
CommonTransitions.createCover(CommonTransitions.SLIDE_VERTICAL, true, 300));
        }
    }
});

```

```
// notice dialogs are blocking by default so its pretty easy to write this logic
if(!Dialog.show("Again?", "Ready to play Again", "Yes", "Exit")) {
    Display.getInstance().exitApplication();
}

// play again
showGameUI();
}
});
}
}).schedule(1800, false, gameForm);
}

/**
 * A blocking method that creates the card deal animation and binds the drop logic
 * when cards are dropped on the deck
 */
private void dealCard(Component deck, final Container destination, Image
cardImage, Card currentCard) {
    final Button card = new Button();
    card.setUIID("Label");
    card.setIcon(cardImage);

    // Components are normally placed by layout managers so setX/Y/Width/Height
    // shouldn't be invoked. However,
    // in this case we want the layout animation to deal from a specific location. Notice
    // that we use absoluteX/Y
    // since the default X/Y are relative to their parent container.
    card.setX(deck.getAbsoluteX());
    int deckAbsY = deck.getAbsoluteY();
    if(destination.getY() > deckAbsY) {
        card.setY(deckAbsY - destination.getAbsoluteY());
    } else {
        card.setY(deckAbsY);
    }
    card.setWidth(deck.getWidth());
    card.setHeight(deck.getHeight());
}
```

```
destination.addComponent(card);
```

// we save the model data directly into the component so we don't need to keep track of it. Later when we

```
// need to check the card type a user touched we can just use getClientProperty  
card.putClientProperty("card", currentCard);  
destination.getParent().animateHierarchyAndWait(400);  
card.setDraggable(true);
```

// when the user drops a card on a drop target (currently only the deck) we remove it and animate it out

```
card.addDropListener(new ActionListener() {  
    public void actionPerformed(ActionEvent evt) {  
        evt.consume();  
        card.getParent().removeComponent(card);  
        destination.animateLayout(300);  
    }  
});  
}
```

```
public void stop() {  
    current = Display.getInstance().getCurrent();  
}
```

```
public void destroy() {  
}
```

```
static class Card {  
    private char suite;  
    private int rank;  
  
    public Card(char suite, int rank) {  
        this.suite = suite;  
        this.rank = rank;  
    }  
  
    private String rankToString() {
```

```
    }
```

```
        if(rank > 10) {
            switch(rank) {
                case 11:
                    return "j";
                case 12:
                    return "q";
                case 13:
                    return "k";
                case 14:
                    return "a";
            }
        }
        return "" + rank;
    }

    public String getFileName() {
        return rankToString() + suite + ".png";
    }
}
```