

1. Introduction

While writing this article I came to realize that there was a lot more information that I want to impart to you than can be covered in just 3 articles (unless I made them all 40 pages long, and you don't have time to read something like that, do you?). So, I will append this series of articles with a fourth article next month at no extra charge to you ☺.

In the previous article we covered:

- BlackBerry architecture
- Web service basics with Domino
- Basic BlackBerry MDS applications
- Advanced BlackBerry MDS Applications
- Deploying BlackBerry Applications to handsets

In this third article (now out of four), we shall show you how to use the BlackBerry Java Development Environment (JDE).

This environment gives you complete control over the application and gives you access to all of the device capability - but at a cost. You are now responsible for getting data to and from the device.

A reasonable level of Java knowledge is assumed.

The final article next month will deal with making this application communicate with Domino and refresh data over the air.

Get some coffee, and some free time - this article will take some time to digest.

2. JDE Applications

In the previous article, we discussed using some simple and some complex Web services. Using the BlackBerry MDS toolkit, we could then easily build applications that consumed this data, and display it to our user.

In a perfect world, this might be sufficient. However, users are often off-line and therefore cannot use interactive web services. Or perhaps you wish to use some of the BlackBerry built-in peripherals, such as the camera or GPS. Maybe you wish to sell the application or perhaps distribute it to users outside of your immediate control.

For these reasons, and for many more, you might want to use the JDE to build java applications.

Let's now walk through building a simple BlackBerry application using JDE from the ground up.

1. *Obtain and become familiar with the JDE*

Like the MDS toolkit, the BlackBerry JDE is free and can be downloaded from <http://www.blackberry.com/developers>. Go do that now. Since it's not some catwalk model, it may take some time to squeeze down those Internet tubes.

The JDE comes with lots of sample applications, illustrating different parts of the BlackBerry API. Make some time and build, deploy and understand the ones that you may be interested in.

The JDE also comes with a full BlackBerry simulator. Though I say simulator, it's running almost exactly the same code as the handsets, so it's more of an "emulator".

One other thing to consider downloading and installing at this point is the BlackBerry MDS simulator (again, this is available for free from <http://blackberry.com/developers>). This is a standalone web application that allows your BlackBerry simulator to make web calls out into your corporate intranet.

With this installed, you can quickly and easily check out your intranet web applications on the simulator, to see exactly what the user would see.

Go download and install all those goodies and come back to the article.

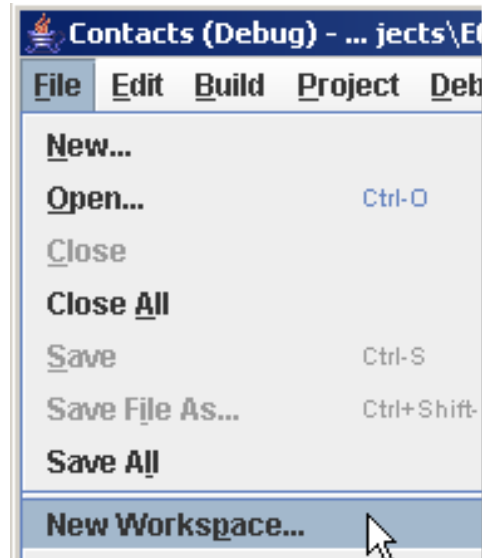
Oh, one last thing. Just like the BlackBerry Enterprise server, all of this stuff is Windows only - so if you have a Mac, or Linux, then run all this on a VMWare or Parallels XP image. This is what I do, and it works well.

Of course, even if you have an XP machine, or god forbid, Vista, sometimes it's still best to run all this stuff in VMWare or Parallels - then you don't end up with a massively complex machine that takes ages to rebuild when it inevitably crashes.

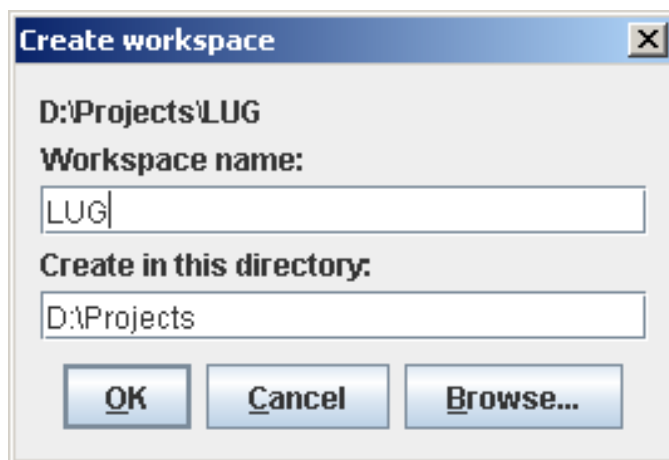
2. **Creating a new Workspace, Project and Java file**

The first thing we shall do is create a new workspace in the JDE to hold our new projects. This keeps all of these projects insulated from each other, and makes them far easier to move around later.

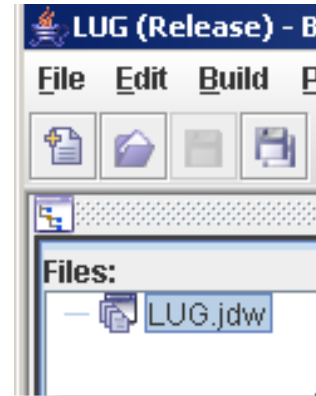
1. Open the Java Development Environment, and click on File, New Workspace:



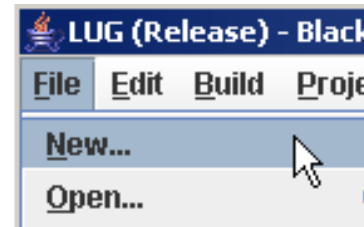
2. Let's enter LUG as the name of the workspace, and give it a new directory.



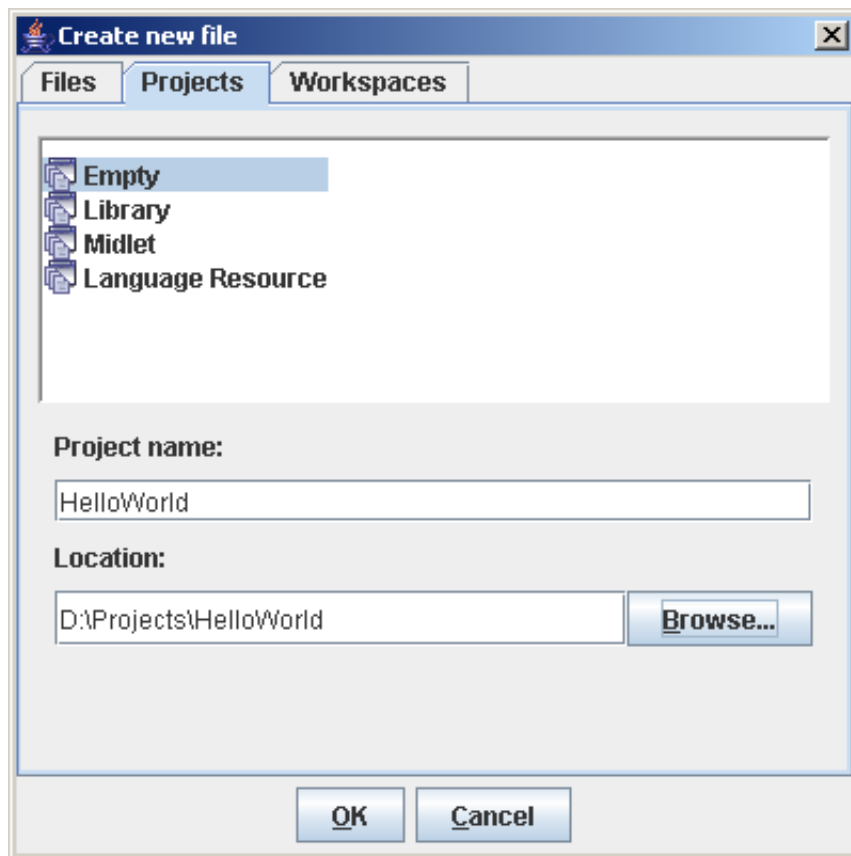
3. We should now see a nice clear workspace.



4. Let's go and create a new project. Let's call it something cool and froody like "Hello World". So start off with "File, New"....



5. And in the dialog box, select the Projects tab.



Enter the name of the project, and enter a new blank directory for it to live in.

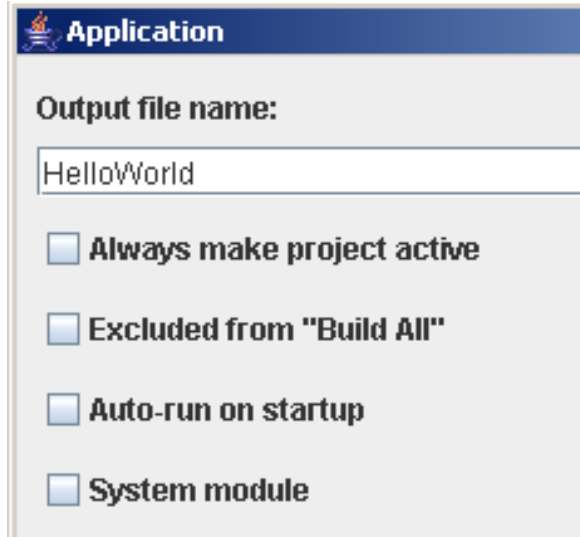
The JDE will try and put it beside the JDE itself underneath "Program Files" but I usually don't let it...

6. Just in case you mistyped the directory, you're asked to confirm it.



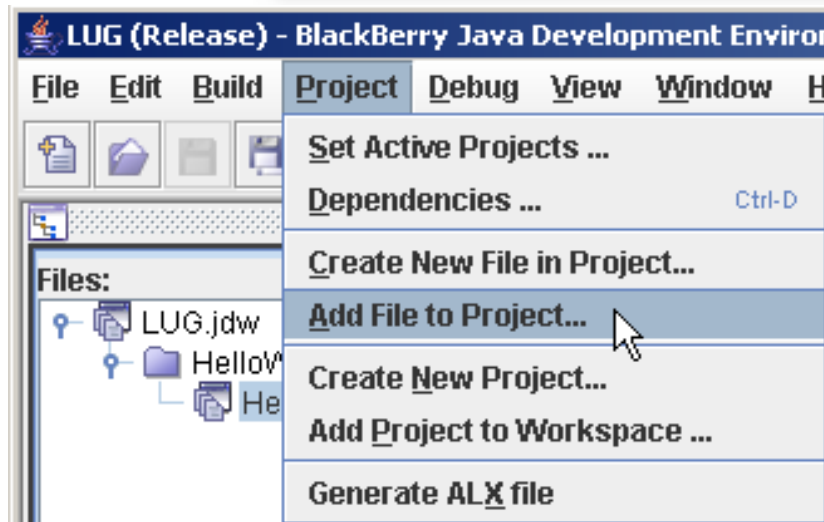
7. And finally, we're asked to confirm some project defaults. Now, defaults are a little like sleeping children - you never touch them unless you have a very good reason. And, for this application, we don't have a very good reason.

So let's leave them asleep, and click "Finish."



8. Now we can start adding files to our project.

Click on Project, Add File to Project.





9. What's in a name? Well, this is where naming conventions start getting really important. No, really. Why?

1. Your BlackBerry will run more than one application at a time. So clearly, we can't really have more than one application called HelloWorld, can we. So the name of the application needs to be unique.
2. Java itself enforces naming conventions in its libraries. So for instance, most of the core Java networking starts with "java.net."
3. As a Java convention, commercial programs start with "com". Typically, you then put the name of your firm, and then a unique project name. So in this case, I'm putting the Java files in a subdirectory called "com\hads\helloWorld".

This also means that all files placed in that subdirectory will be in the same Java "package", and therefore all Java classes can access each other - quite important and very typical.

4. Whilst this seems overkill at this point, unique application numbers are required later in the application. The JDE can take a name such as "com.hads.helloWorld" and produce a unique hash code from it.

This means that we can then use that unique hash code within our application and have a fairly good guarantee that anything created using it will be only used by our application.

10. The JDE will then confirm the file you wish to create.



11. The JDE will then create a fairly blank Java file, which we can extend.

```
/*
 * helloWorld.java
 *
 * © <your company here>, 2003-2005
 * Confidential and proprietary.
 */

package com.hads1.helloWorld.helloWorld.java;

/**
 *
 */
class helloWorld {
    helloWorld() { }
}
```

Congratulations. You've created your first Java project. Granted, functionally it does absolutely nothing. Let's add some functionality...

3. *Creating our Hello World Application.*

Everything in the JDE (and let's face it, almost everything in Java) is based around a framework. You're given various classes which contain most of the functionality you require, and you extend them, adding your functionality.

JDE development is no different.

Of course, the hard thing to figure out when you're staring at a blank screen is where to start. And this is where this article should help - this should give you enough to get an application up and running. Later, you should revisit this when you develop your own code, and find out more about each framework class as you require.

The thing to remember is that:

- BlackBerry UI based applications are extended from a class called UiApplication, and usually implements two interfaces - KeyListener and TrackWheel Listener.
- You have to implement something for these interfaces to work, even if it's a blank function to work.
- The BlackBerry application framework does NOT instantiate your class. It relies on a static "main" function for you to do it. Just like a stand-alone Java package, in fact.

Let's now look at our starting code, and implement this class. Let's start by defining a single import library:

```
import net.rim.device.api.ui.*;
```

And changing our class definition to:

```
class helloWorld extends UiApplication  
    implements KeyListener, TrackwheelListener
```

We also have to add a "main" function that will allow this class to construct itself.

```
// Our Entry Point  
public static void main(String[] args)  
{  
    try  
    {  
        System.out.println("Starting helloWorld");  
        helloWorld H = new helloWorld();  
        H.enterEventDispatcher();  
    }  
    catch (Exception e)  
    {  
        System.err.println("An error has occurred " +  
            e.toString());  
        System.exit(0);  
    }  
}
```

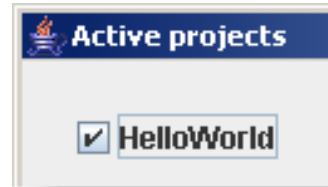
But when we click on the Build Menu or press F7, nothing happens! Why?

This is because in the JDE the designers recognize that you might have a large number of projects on a workspace, but only want to rebuild one or two at a time. We must therefore make this project "active":

1. Click on Project, Set Active Projects



2. And click on the checkbox next to the project you wish to make active.
3. We can now try to compile this application. Press <F7> on your keyboard.



When we try and compile the application, we get a number of errors.

```
D:\Projects\HelloWorld\com\hads1\helloWorld\helloWorld.java\helloWorld.java:16: cannot find symbol
symbol: class TrackwheelListener

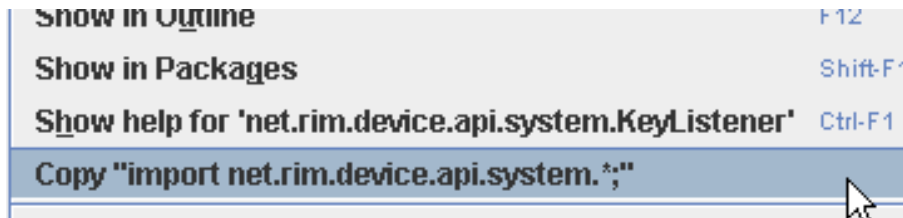
class helloWorld extends UiApplication implements KeyListener, TrackwheelListener

^
2 errors
```

Again, this is very common in the Java world. We have used a new class in our code, and the Java compiler can't compile because it's not already been defined.

We need to find out the name of the import package to add to our source code. But I for one cannot be bothered dredging through countless Java libraries looking for the correct import line. And I suspect I'm not alone in this.

If you right-click on the unknown keyword, an option appears on the context menu. This places the correct import statement on the clipboard. All you have to do, is then copy it in place at the top of the Java file between the "Package" declaration and the start of the source code.



Once you've learned this trick, life gets a lot easier.

Once you paste this import at the top of the code, another error pops up:

```
D:\Projects\HelloWorld\com\hads1\helloWorld\helloWorld.java\helloWorld.java:16: com.hads1.helloWorld.helloWorld.java.helloWorld is not abstract and does not override abstract method keyStatus(int,int) in net.rim.device.api.system.KeyListener

class helloWorld extends UiApplication implements KeyListener, TrackwheelListener
```

Basically, this means that we need to implement various functions in our code.

```
public boolean keyStatus(int keycode, int time)
{
    return false;
}
public boolean keyRepeat(int keycode, int time)
{
    return false;
}
public boolean keyUp(int keycode, int time)
{
    return false;
}
public boolean keyDown(int keycode, int time)
{
    return false;
}
public boolean keyChar(char key, int status, int time)
{
    return false;
}
public boolean trackwheelRoll(int amount, int status, int time)
{
    return false;
}
public boolean trackwheelUnclick( int status, int time )
{
    return false;
}
public boolean trackwheelClick( int status, int time )
{
    return true;
}
```

Adding these functions to our code means it'll now compile. Again, we've done nothing yet to actually make anything happen.

Let's add a couple of private member variables, and some code to our constructor:

```
private MainScreen _mainScreen;
private LabelField _statusMessage;

helloWorld()
{
    _mainScreen = new MainScreen();
    _mainScreen.setTitle( "Hello World!" );
    _mainScreen.add(new SeparatorField());
    _statusMessage = new LabelField("Starting Up");
    _mainScreen.add(_statusMessage);

    pushScreen(_mainScreen);
}
```

At this point, we're actually defining a screen object derived from MainScreen called `_mainScreen`, setting up a label field called `_statusMessage`, and adding it to the `_mainScreen`.

At this point, the screen itself is not visible. The last call - "pushScreen" - pushes the `_mainScreen` onto the UI stack, and the screen becomes invisible.

This means that (assuming it compiled cleanly), we can now debug our program by hitting <F5>.

Where can we find our application? Well, on the BlackBerry simulator, use the arrow keys and enter key to navigate to the second window. You'll find it lurking down there near the bottom:



We can now press ENTER and our application will actually appear:

Hello World!
Starting Up

And the nice thing about reusing a framework, is that several default actions are already in place.

If you use your mouse to click on a menu button or trackball, you get a default menu with "Close" defined.



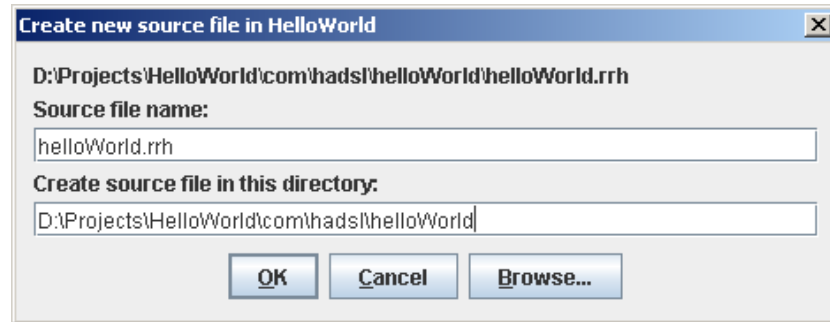
Granted, it's not a particularly interesting application yet, but now it's an application.

4. Externalizing Strings

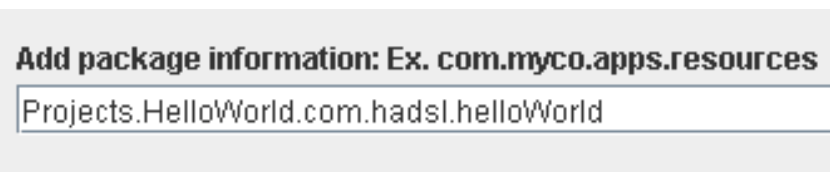
Whilst not the most exciting topic in the world, internationalization is key to successful application development. And the sooner applications have hard-coded strings externalized, the easier it is when the application is enhanced.

Thankfully, the JDE handles most of the grunt work for us.

1. Create a new file with the file extension "rrh".



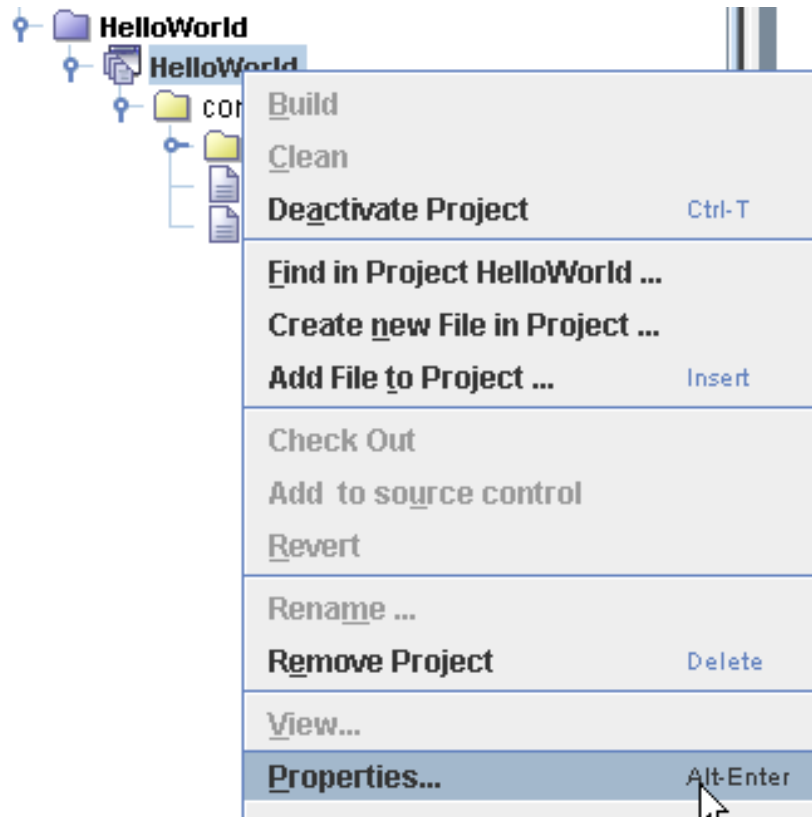
2. You will then be prompted to add this resource to the project.



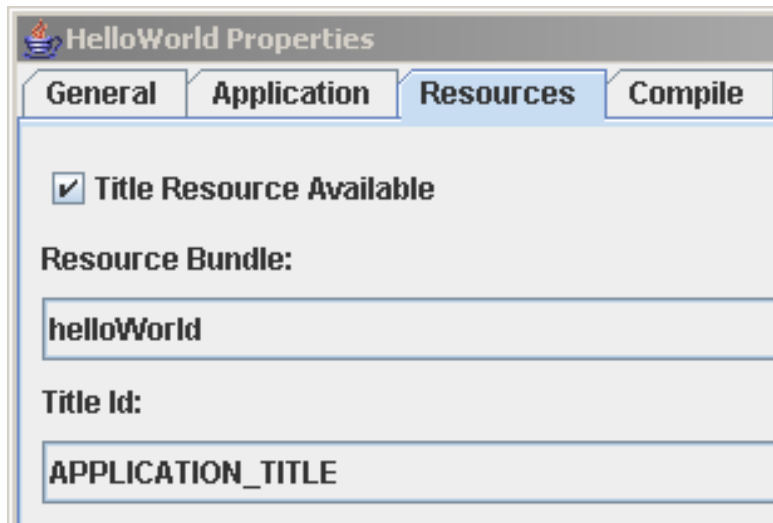
3. And then add a new file with extension "rrc".
4. We can now start entering some strings into our resource file. Double-click on the file ending in "rrh".

Keys	Values
APPLICATION_TITLE	Hello World
STATUS_STARTING	Starting Up

5. We can now tell the project that we wish to use this resource file, and that we wish to pick up the application name from the application name from the resource file. Right click on the project, and select "Properties"



6. In the project properties section, we can now direct the application to pick up the application title from the resource section:



Click on the Resources tab, select the resource bundle, then in the Title ID section, select the application title resource in the Title ID field.

We can also incorporate the resources file in our code. We can add this line to the list of private members in our class, so it is instantiated at startup. Note that in order to refer to it, we have to append the word "Resource" to our resource file name. So we use "helloWorldResource" instead of "helloWorld".

```
ResourceBundle _resources = ResourceBundle.getBundle(  
  
    helloWorldResource.BUNDLE_ID,  
    helloWorldResource.BUNDLE_NAME);
```

And in the constructor, we can now use this resource bundle.

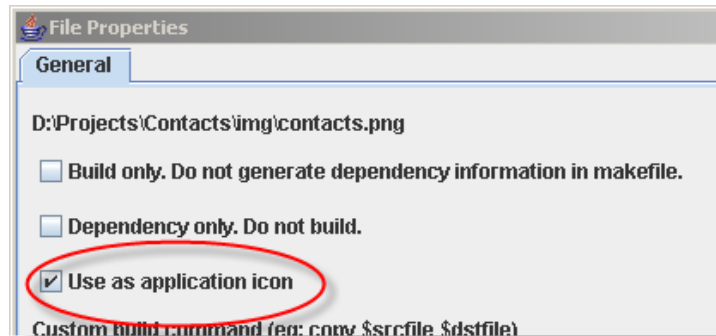
```
_mainScreen.setTitle(  
    _resources.getString(  
        helloWorldResource.APPLICATION_TITLE),  
    LabelField.ELLIPSIS | LabelField.USE_ALL_WIDTH));  
statusMessage = new LabelField(  
  
    _resources.getString(  
        helloWorldResource.STATUS_STARTUP));
```

Later on, we can create multi-lingual versions of these resources, and all string resources can have different translations depending on the language of the user's handset.

5. Adding an Application Icon

This is really simple. Find yourself a nice 43x43 GIF or single-layer PNG file, insert it into the project, and then tell it to be the application icon:

1. Copy the file to an "img" subdirectory (alongside your "com" directory) in your project folder. It can really be anywhere, but this is as good a convention as any other.
2. Right click on the project, and choose "Add file to Project".
3. Browse to the file (remembering to change the default file type on the dialog to "All Files", right?) and select it. This will add it to the project.
4. Right click on the file in the project browser, and choose "Properties".
5. Enable the option "Use as Application Icon".



6. Finished!
The following is our new "Hello World" icon.

If you look in the JDE Sample Applications directory, you will find many other professional looking icons.



6. Adding an About Page

Adding an About Page will show us how to construct a screen from scratch, and how to add the additional screen into our application menu. How do we do this?

1. First we need to add a new blank Java file to our project. Right click on the project file itself, and select "Create file in Project". Let's call it "aboutScreen.java" and place it in the "com/hadsl/helloWorld" directory as before.
2. Change the default code so that the class now looks like:

```
package com.hadsl.helloWorld;
import net.rim.device.api.ui.*;
public final class aboutScreen extends MainScreen{
    ...
}
```

3. As you can see, we've changed our class to inherit from class MainScreen. Let's go and add some private members to this screen:

```
ResourceBundle _resources = ResourceBundle.getBundle(
    helloWorldResourceResource.BUNDLE_ID,
    helloWorldResourceResource.BUNDLE_NAME);
private UiApplication app = null;
```

In this case, we're adding in our resource bundle again, and adding in an internal pointer to an instance of UiApplication.

4. Let's now go add some code to our rather naked looking constructor:

```
super();
_app = uiApp;
this.setTitle(_resources.getString(
    helloWorldResourceResource.ABOUT_SCREEN_TITLE));
```

This code calls the constructor's superclass constructor, sets up our reference to our UiApplication instance, and then sets the title for our screen.

5. However, we have not yet added code to make this screen visible. Instead of "pushing" the screen during our constructor, we shall add a 'display' function so our calling class can separate out the construction of this object, and the screen display:

```
public void display()
{
    _app.pushScreen(this);
}
```

6. If we were to somehow call this screen at this point in time, it'd just be a blank screen with our title set. Let's put some text on this screen to make it more interesting. However, let's use different font information so that the prompts and the information are visually distinct.

```

FontFamily ffPreferred = null;
Font fPreferred = null;
Font fBold = null;
try
{
    ffPreferred = FontFamily.forName("System");
    fPreferred = ffPreferred.getFont(
        Font.PLAIN, 8, Ui.UNITS_pt);
    fBold = Font.getDefault().derive(Font.BOLD,8,Ui.UNITS_pt);
}
catch (java.lang.ClassNotFoundException e)
{
    fPreferred = Font.getDefault().derive(Font.PLAIN);
    fBold = Font.getDefault().derive(Font.BOLD);
}
Bitmap myBitmap = Bitmap.getPredefinedBitmap(
    Bitmap.INFORMATION);
BitmapField myBitmapField = new BitmapField(myBitmap);
FlowFieldManager _ffm2 = new FlowFieldManager
    (VerticalFieldManager.FIELD_HCENTER);
_ffm2.add(myBitmapField);
add(_ffm2);
LabelField _version = new LabelField(_resources.getString(
    helloWorldResourceResource.APPLICATION_VERSION));
_version.setFont(fPreferred);
add(_version);
add(new SeparatorField());

LabelField _disLabel = new LabelField(_resources.getString(
    helloWorldResourceResource.ABOUT_DISCLAIMER_LABEL));
_disLabel.setFont(fBold);
add(_disLabel);

LabelField _disclaimer = new LabelField(_resources.getString(
    helloWorldResourceResource.ABOUT_DISCLAIMER));
_disclaimer.setFont(fPreferred);
_disclaimer.setEditable(false);
add(_disclaimer);
add(new SeparatorField());

```

This is slightly more interesting

- During the first section we define two fonts we wish to use - preferred and bold.
- Then we define a bitmap using a predefined bitmap.
 - We then go on to define a FlowField. This is a screen management framework which allows us to place more than one field across the screen.
 - We then add the bitmap to the flowfield, and the flowfield to the screen.
 - We then define a label field containing a version string from our resource bundle and add that to our screen.

- We then add a new SeparatorField - this is a horizontal line which helps group our fields in logical visual blocks, and add this to our screen.
- We then add a disclaimer label, then some disclaimer text, again from our Resource bundle.
- We finish by adding another separator line.

7. Our application would be very dull indeed if all it did were display label field after label field. Let's add a slightly more interesting data object:

```

LabelField _devLabel = new LabelField(
    _resources.getString(
        helloWorldResourceResource.ABOUT_DEVELOPED_BY));
_devLabel.setFont(fBold);
add(_devLabel);

ActiveAutoTextField _devBy = new ActiveAutoTextField(
    "",
    _resources.getString(
        helloWorldResourceResource.ABOUT_AUTHOR_NAME) +
    ", " +
    _resources.getString(
        helloWorldResourceResource.ABOUT_AUTHOR_COMPANY) +
    " - " +
    _resources.getString(
        helloWorldResourceResource.ABOUT_AUTHOR_COMPANY_URL));

_devBy.setFont(fPreferred);
_devBy.setEditable(false);
add(_devBy);

```

Ah. Now this is more like it! An ActiveAutoEditTextField enables the BlackBerry to take whatever you fling at it, and decide whether it's an email address, phone number, web URL or suchlike. It will then present the user with a prompt allowing the user to call, email, or open the link. Very powerful.

As you can see, both the label (in bold) and the screen prompt are again pulled in from our resource strings.

8. So far we have defined our screen (and whilst quite empty, is actually complete now). But we have not allowed the user to navigate to this screen. To do this we now open the helloWorld.java source file again, and add more code.

We start off with a private member declaration to hold our aboutScreen object in and two new menu entries that we shall work on. These are defined at the class level itself:

```

private aboutScreen _aboutScreen;

private MenuItem _aboutItem;

private MenuItem _closeItem;

```

9. In our constructor, we can now assign these elements some values:

```
// Add the About Menu
_aboutItem = new MenuItem("About", 100000, 10)
{
    public void run()
    {
        _aboutScreen = new aboutScreen(
            UiApplication.getUiApplication());
        _aboutScreen.display();
    }
};

_closeItem = new MenuItem("Close", 400000, 10)
{
    public void run()
    {
        int response = Dialog.ask(
            Dialog.D_YES_NO, "Do you want to exit....");
        if (response == -1)
        {
            //no
            //do nothing
        } else {
            // will be 4 indicating yes
            System.exit(0);
        }
    }
};
```

We're actually creating new instances of MenuItem, and assigning code for that menu option to run.

In the first instance - aboutScreen - we're also setting up our private member variable when we call it.

10. We may have defined some new menu items, but we still don't see them within the application when we run it. So clearly, we're missing an important piece of wiring. In this case, we're missing our "listening" functions which act upon the trackball being pressed. So let's add in the supporting code for that. Firstly, we need to tell the framework that we want to listen for the trackball and keyboard events (this code is added to the constructor for helloWorld):

```
_mainScreen.addKeyListener(this); //implemented here
_mainScreen.addTrackwheelListener(this); //implemented here
```

11. What's involved in creating a listener function? Well, when we added all those boilerplate functions at the start of this exercise, we actually added an empty listening function. Let's look at the expanded code for this:

```
public boolean trackwheelClick( int status, int time )
{
    Menu menu = new Menu();
    makeMenu(menu, 0);
    menu.show();
    return true;
}
```

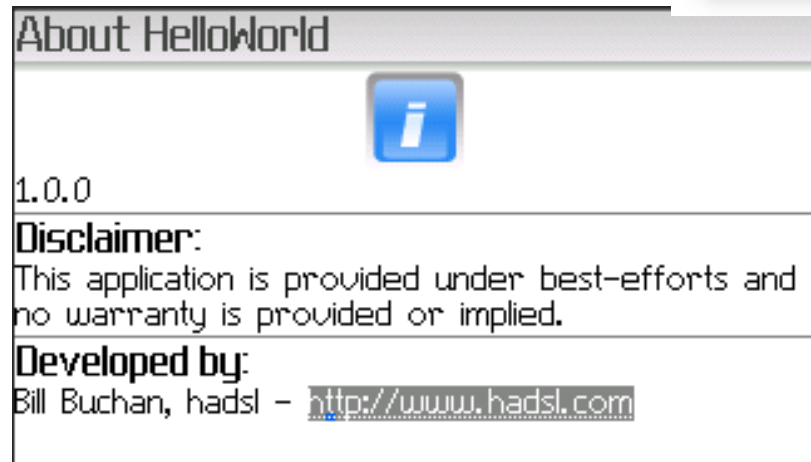
12. We now need to add this "makeMenu" function.

```
protected void makeMenu(Menu menu, int instance)
{
    menu.add(_closeItem);
    menu.add(_aboutItem);
}
```

13. Phew. Now, when we run our application, and click on the trackball, we should now see our new menu item:



14. And when we click on the About menu, we can then see our About screen in all its glory:



15. What about our URL field? Will it indeed actually open a web link when it sees a URL?

Use the trackball or arrow keys to move the cursor over the wheel, and hit the trackball. As you can see, the BlackBerry has correctly interpreted this field as a web link.



3. Summary

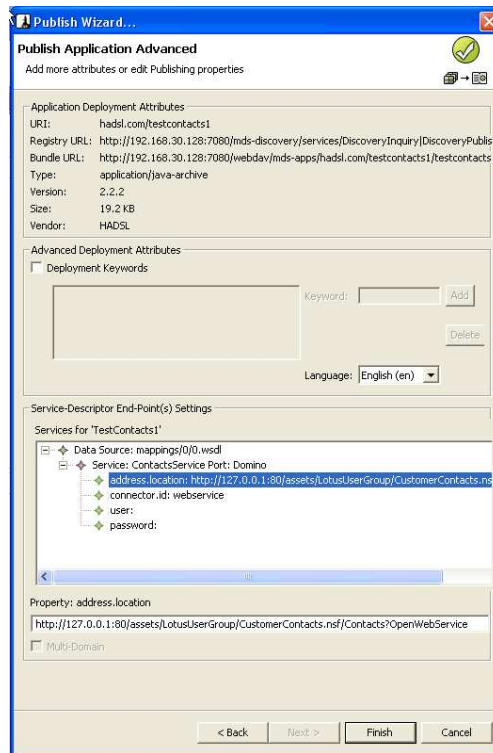
Congratulations! We have now constructed a fairly simple - but valid nevertheless - BlackBerry application which contains screens, menu linkage, smart fields and graphics.

All of these techniques now form the basis for the last (I promise!) article in this series - where we show you how to hook up this application to a Lotus Domino server, and have the BlackBerry handsets obtain their information over the air.

More importantly, you've achieved in a few short hours what normally takes others weeks to do - you've gotten started in BlackBerry JDE programming.

4. Resources

No BlackBerry Article would be complete without a trail of Breadcrumbs to follow:



- <http://www.blackberry.com/developers> - the BlackBerry developer zone. Like all on-line resources, full of lots of articles. You could (and may well) spend days in there.
- http://www.blackberry.com/knowledgecenterpublic/livelink.exe/fetch/2000/348583/796557/800451/1055819/What_Is_Sample_applications_demonstrating_BlackBerry_push_technology_Emergency_Contact_List.html?nodeid=1055822&vernum=2&cp=NLC-23 - This is an excellent "Push" application showing a Domino application pushing structured data to one or more handsets running a JDE listener task. A rare thing indeed - a well-written demonstration application that works.
- "Professional BlackBerry" by Craig James Johnston and Richard Evers is one of the few books on the market dealing with BlackBerry infrastructures and handsets in significant depth. Well recommended.
- The following white papers are required reading and are all available from the BlackBerry Developers Zone:
 - The BlackBerry Java Development Environment Development guide.
 - Deploying BlackBerry Applications.
 - Design Principles for BlackBerry Browser Applications.