Porting from BlackBerry to Series 40

This article explains what needs to be taken into account when porting BlackBerry apps to Series 40.

28 Oct 2012

Porting considerations

BlackBerry devices (up to BlackBerry OS 7.1) and Series 40 devices share the same development platform - Java ME. This makes porting from BlackBerry to Series 40 quite straightforward and transparent, especially when compared to other mobile platforms. Generic Java ME MIDlets that use the standard set of LCDUI controls, Canvas, GameCanvas, and the basic set of optional JSRs, should be possible to deploy to both BlackBerry and Series 40 after platform-specific repackaging. Of course, the behavior of LCDUI Commands and the default UI look & feel varies significantly between platforms but from the development perspective optimising a standard Java ME MIDlet for a specific platform is a trivial task. The familiarity of the programming language, Java, also plays a significant role in simplifying the porting process. Many generic Java ME libraries can be reused on both platforms “as is”.

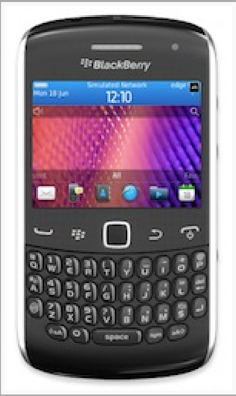
At the same time, most of the benefits that developers used to get from BlackBerry devices are specific platform integration APIs, which allow deeper utilisation of device services when compared to generic MIDlets. On top of that, the BlackBerry UI frameworks are different from the traditional Java ME architecture and much closer to the Java Swing ideology when compared to the classic LCDUI which is, with some Nokia extensions, the basis of Series 40. This means that the estimated difficulty of porting “BlackBerry Application” types (which is the term used in the BlackBerry Application Descriptor XML) is probably on the same level in as Android to Series 40 porting projects. Developers might also consider using the LWUIT library on Series 40, which provides rich set of UI components and Java Swing inspired architecture. This approach can definitely be an interesting choice especially for new projects because the Nokia release of LWUIT has been tailored for Series 40.

LWUIT for Series 40

And finally, developers should recognise the major differences between the platforms: BlackBerry devices are full smartphones with multitasking capabilities, and Series 40 devices are feature phones with much more limited resources and device capabilities. However, with the right approach and creativity, you should be able to achieve high standards of user experience for ported applications in several categories and benefit from the giant marketplace of Series 40 devices in consumer hands.

Devices especially interesting for porting projects

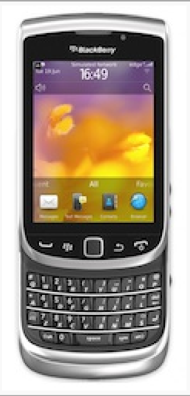
Developers with BlackBerry applications targeting traditional QWERTY keyboard devices should take a close look at these Nokia devices:

BlackBerry Bold 9900	BlackBerry Curve 9350	Nokia Asha 303	Nokia Asha 302
			
640 x 480	480 x 360	240 x 320	320 x 240
QWERTY and Capacitive Touch Screen	QWERTY	QWERTY and Capacitive Touch Screen	QWERTY
ROM/RAM 8GB/768MB, 32 GB Micro SD	ROM/RAM 512MB/512MB, 32 GB Micro SD	ROM/RAM 256MB/128MB, 32 GB Micro SD	ROM/RAM 256MB/128MB, 32 GB Micro SD

Nokia Asha 302 and Nokia Asha 303 are the fastest QWERTY Series 40 devices with 1 GHz CPU, 4 MB of Java Heap space available for 3rd party applications. The expected user base is very similar to BlackBerry Bold and Curve users, especially for the messaging and business types of apps. Nokia Asha 302 supports landscape orientation and integration with Mail for Exchange, which makes it one of the primary porting targets for traditional BlackBerry apps.

Full touch devices

With the announcement of Nokia Asha 305, Nokia Asha 306, and especially Nokia Asha 311, BlackBerry developers targeting the full touch experience should also consider porting to these devices. In October 2012, this family of full touch devices was extended with two new members Asha 308 and Asha 309.

BlackBerry Torch 9860	BlackBerry Torch 9810	Nokia Asha 311	Nokia Asha 305/306	Nokia Asha 308/309
				
800 x 480	640 x 480	240 x 400	240 x 400	240 x 400
Capacitive Touch Screen	QWERTY and Capacitive Touch Screen	Capacitive Touch Screen, Multipoint-touch (5 points)	Resistive Touch Screen, Multipoint-touch (2 points)	Capacitive Touch Screen, Multipoint-touch
ROM/RAM 4GB/768MB, 32 GB Micro SD	ROM/RAM 8GB/768MB, 32 GB Micro SD	ROM/RAM 256MB/128MB, 32 GB Micro SD	ROM/RAM 64MB/32MB, 32 GB Micro SD	ROM/RAM 128MB/64MB, 32 GB Micro SD

CPU clock speed for full touch Asha devices is in the range 200 MHz - 1 GHz. Nokia Asha 311 with 1 GHz CPU is an especially interesting device, giving that the performance is significantly faster than the performance of the more budget-oriented Nokia Asha 305/306 and Nokia Asha 308/309.

Portability of different application types

2D games	Excellent portability, source code-level compatibility. 2D game development on BlackBerry and Series 40 uses the same Java ME APIs from the javax.microedition.lcdui.game package. For devices with similar input types (touch, non-touch) porting would be possible by simply recompiling and repackaging. However, the main focus for porting will be the optimisation of graphical assets to smaller screen sizes, adaptation of different touch gestures, and adjusting to Series 40 memory requirements.
3D games	Generally portable. BlackBerry 3D gaming uses Java interfaces for OpenGL ES. Series 40 doesn’t support OpenGL ES but it features an excellent implementation of Mobile 3D Graphics (JSR-184 or M3G). This JSR is significantly simpler and optimised for low memory footprint but it is far from being primitive and it is a real 3D rendering engine with a special immediate mode, which is ideologically very similar to OpenGL. Similarly as in 2D gaming, Series 40 memory requirements impose challenges for textures and graphical assets.
Social networking	Applications with OAuth authentication are not portable as Java apps because the Series 40 platform doesn’t support embeddable web view and the custom Java ME implementation of OAuth appears to be challenging. The Series 40 platform doesn’t support multitasking and background execution, so implementing messaging notifications when the application is not running is not possible. However, if background functionality is not needed and if the social networking app is using custom authentication schemes or private APIs for well-known networks, then it is generally portable. Developers of such apps should consider exploring Nokia Web Apps, where at least the OAuth issue can be resolved.
SMS, MMS, and Bluetooth	Generally portable. BlackBerry and Series 40 both feature the Wireless Messaging API (JSR-205) and Java APIs for Bluetooth (JSR-82). Some parts can be compatible on the source code level with the correct level of

Messaging	abstraction from the UI.
Telephony and platform messaging services	Generally not portable. The Series 40 platform doesn't support integration with telephony and other these types of services for 3rd party developers. Applications for VoIP, Caller ID, Tethering, platform email, and Social Networking apps' integrations are not feasible on Series 40.
Video, audio, and image processing	Generally not portable. The Series 40 platform doesn't provide APIs for media parsing, recompressing, and image processing. It is also related to memory restrictions (2 or 4 MB Java Heap) imposed on 3rd party applications.
Photography and camera	Generally portable with Advanced Multimedia Supplements APIs (JSR-234) which is also supported on BlackBerry with source code level compatibility. Image filtering and processing full resolution images is not possible due to memory restrictions.
News readers, book readers, business, finance, comics, magazines	Generally portable. UI redesign and optimisation of graphical assets is required. The Series 40 platform doesn't support embeddable web view and book / magazine reader apps should implement custom text rendering with LCDUI components or Canvas. Comics and magazine readers should take special precautions regarding memory limitations.
Media playback and streaming	Generally portable with Mobile Media API (JSR-135). Simple audio / video playback and streaming is possible and might be compatible on source code level. Redesign is required for background audio streaming apps, which are not supported on Series 40.
Maps and location	Generally portable with Nokia Maps API for Series 40, which provides also search, geocoding, reverse geocoding, and routing functionality. Series 40 devices are not equipped with GPS, so Cell ID-based location or an external GPS device with Bluetooth connection is needed.

Memory considerations

Modern Series 40 devices impose specific memory limitations on 3rd party applications. The application JAR file size should not exceed 2 MB and Java Heap Size can be either 2 or 4 MB. For applications that use many graphical assets and custom backgrounds, especially games, it might be a challenge to pre-bundle the required resources. The smaller screen size of Series 40 may help reduce the sizes and resolutions of images; however, memory limitations should be considered carefully. A potential workaround for storing a big amount of binary resources can be the usage of RMS (Record Store) and file system. In this case the required application data can be loaded over the network and cached locally.

At the same time, Java Heap size can be a serious obstacle for certain imaging applications. For example, if the application requires displaying large images even from the device image gallery, the developer should take special care of checking the image file size to avoid potential OutOfMemory exceptions.

The Nokia Device Matrix is a very good tool for finding information about devices parameters relevant for porting projects:

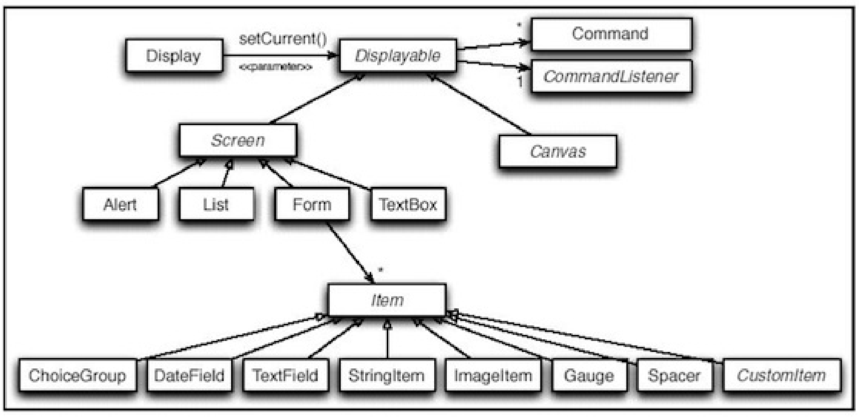
Series 40 Devices with Java Heap size 4MB

Series 40 Devices with JAR file size 2 MB

User interface

Working on the user interface of the application takes most of the development and design time. It is important to correctly map the user experience from BlackBerry to Series 40 using the right APIs and components. A typical BlackBerry application with a standard UI architecture uses the fundamental basics from net.rim.device.api.ui, a set of pre-build components from the net.rim.device.api.ui.component package, and the net.rim.device.api.ui.container package for arranging elements. Combined, those packages provide quite a sophisticated user interface and event model, which is optimised for BlackBerry smartphones. Series 40 UI development should utilise the LCDUI model which is suitable for devices with a low memory footprint.

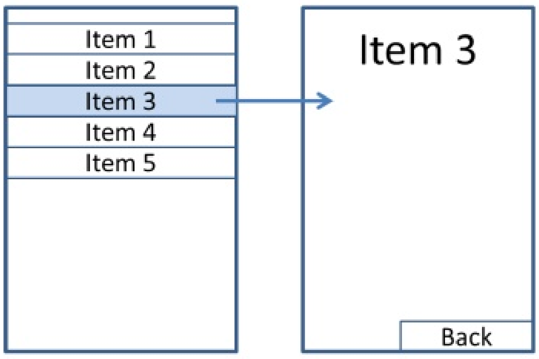
The following diagram (from the Java Developer's Library) illustrates this model:



If the BlackBerry application was created with correct decoupling of models, views, and data; rewriting the UI using a set of LCDUI classes, especially in business applications, should not be a significant issue. The biggest UI-related task that developers will face is optimization for different screen sizes and the navigation model. The screens of a BlackBerry application (UiApplication) are managed according to the stack model - you can push your application screens into the stack and the screen on top of the stack is what is visible to the user. LCDUI on Series 40 behaves differently and you can switch between screens directly. In addition to LCDUI, Nokia provides Series 40-specific extensions known as the Nokia UI API. These APIs enhance MIDP with a functionality related to touch screen gestures, canvas drawing, text editing, and kinetic scrolling.

User Interface porting example

To illustrate differences and similarities in UI programming concepts, let’s consider the following simple application example:



Application will have two screens: main screen with standard list component and second screen with text component, which will display text component with item content selected on previous screen. We will use the most standard way of tracking touch and keyboard events on BlackBerry and Series 40 devices, as well as, standard way of navigating between screens. BlackBerry application will consist of three classes:

- App.java
- AppMainScreen.java
- SimpleScreen.java

Class `App` provides basic application infrastructure as standard BlackBerry `UiApplication` class:

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

public class App extends UiApplication
{
    public static void main(String[] args)
    {
        App theApp = new App();
        theApp.enterEventDispatcher();
    }
    public App()
    {
        pushScreen(new AppMainScreen());
    }
}
```

In the main method we instantiate `App` class and by using `UiApplication` method `pushScreen()` in the constructor of the application class, we create and display main applications screen represented by `AppMainScreen` class.

```
import net.rim.device.api.command.Command;
import net.rim.device.api.command.CommandHandler;
import net.rim.device.api.command.ReadOnlyCommandMetadata;
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.ui.component.table.SimpleList;
import net.rim.device.api.ui.container.MainScreen;

public final class AppMainScreen extends MainScreen
{
 private SimpleList listField;
 private SimpleScreen simpleScreen;

    public AppMainScreen()
    {
     super( MainScreen.NO_VERTICAL_SCROLL);
        setTitle("Simple List Demo");

        listField = new SimpleList(getMainManager());
        listField.add("Helsinki");
        listField.add("Tampere");
        listField.add("London");
        listField.add("Paris");
        listField.add("Berlin");

        listField.setCommand(new Command(new CommandHandler()
        {

            public void execute(ReadOnlyCommandMetadata metadata, Object context)
            {

showSimpleScreen((String)listField.getModel().getRow(listField.getFocusRow()));
            }
        }));

    }

    private void showSimpleScreen(String message){
     if (simpleScreen == null){
      simpleScreen = new SimpleScreen();
     }
     simpleScreen.setText(message);
     UiApplication.getUiApplication().pushScreen(simpleScreen);
    }
}
```

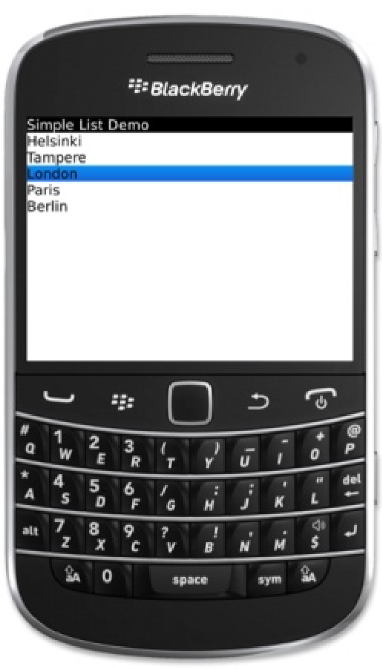
Both application screens `AppMainScreen` and `SimpleScreen` are extending `MainScreen` class which is commonly used screen container that inherits all default behavior of the standard BlackBerry application screens. Main screen displays list component based on `SimpleList` class. Adding list component to the screen and filling it up with items is straightforward operation. One interesting aspect of working with list components is tracking selection of item. Since there is wide variety of BlackBerry devices with touch and keyboard based input, traditional approach is to use both explicit mechanisms for tracking events. However, the most effective way which at the same time significantly minimizes code usage is to use `Command` class with dedicated command handler, like in the example above.

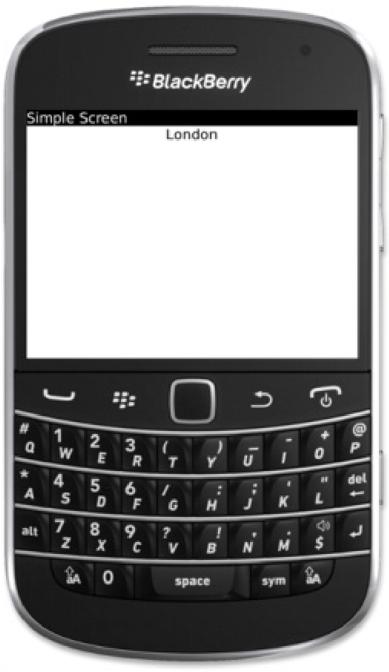
Once it's been detected, that user selected item, method `showSimpleScreen` is called with string content of the selection. At this point, there is a need for making check if our next screen class has been already created, passing string argument and pushing screen to the visible UI stack.

```
import net.rim.device.api.ui.component.LabelField;
import net.rim.device.api.ui.container.MainScreen;

public class SimpleScreen extends MainScreen {

 LabelField labelField;

 public SimpleScreen() {
  super(MainScreen.VERTICAL_SCROLL | MainScreen.VERTICAL_SCROLLBAR);

  setTitle("Simple Screen");
  labelField = new LabelField("Some text", LabelField.FIELD_HCENTER
|LabelField.FIELD_VCENTER);
  add(labelField);
 }

 public void setText(String text){
  labelField.setText(text);
 }
}
```

Actual code for making detailed screen is simple and features only custom method for providing text content for the label component.

Resulting BlackBerry application will look as displayed on screenshots below.





Complete source code for BlackBerry Java 7.1 SDK (Eclipse Plug-in) can be downloaded here:

File:SimpleListDemoBB.zip

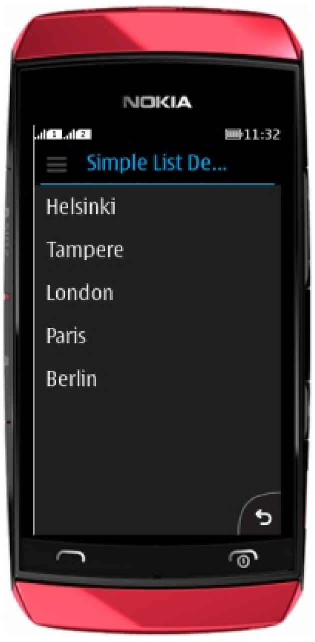
BlackBerry example above illustrates similarities in UI development between desktop Java and BlackBerry UiApplication based development. With Nokia Series 40, developers are using different approach, based on LCDUI component stack. Let’s consider recreating exactly the same application functionality for Series 40 full touch user experience.

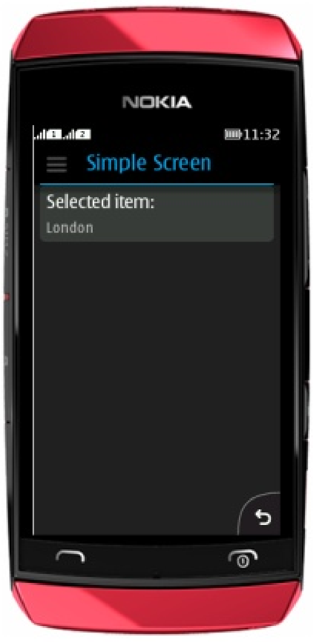
Main difference with LCDUI based midlet development that instead of subclassing container UI components, developer in most cases just instantiate them as new objects. In order to port / recreate our example, there is a need for only single Midlet class which will incorporate all functionality.

The `SimpleListDemo` class is executable Midlet and thus, should implement standard lifecycle methods `startApp()`, `pauseApp()` and `destroyApp()`. In the constructor of the Midlet class there is initialization of entire UI stack – List and Form. In LCDUI it is possible to use `List` as actual displayable component, so there is no need to have extra container. At the same time, it for building secondary screen `Form` is used with `StringItem`.

```
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.StringItem;
import javax.microedition.midlet.*;

public class SimpleListDemo extends MIDlet implements CommandListener {

    private Display display;
    private List simpleList;
    private Form simpleScreen;
    private StringItem stringItem;
    private static final Command CMD_EXIT = new Command("Exit", Command.EXIT, 1);
    private static final Command CMD_BACK = new Command("Back", Command.BACK, 1);

    public SimpleListDemo(){
        display = Display.getDisplay(this);

        simpleList = new List("Simple List Demo", List.IMPLICIT);
        simpleList.append("Helsinki", null);
        simpleList.append("Tampere", null);
        simpleList.append("London", null);
        simpleList.append("Paris", null);
        simpleList.append("Berlin", null);
        simpleList.addCommand(CMD_EXIT);
        simpleList.setCommandListener(this);

        simpleScreen = new Form("Simple Screen");
        simpleScreen.setCommandListener(this);
        stringItem = new StringItem("Selected item:", "Some text");
        simpleScreen.append(stringItem);
        simpleScreen.addCommand(CMD_BACK);
    }


    public void startApp() {

        display.setCurrent(simpleList);
    }
    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void commandAction(Command c, Displayable d) {

        if (c == List.SELECT_COMMAND) {

            stringItem.setText(simpleList.getString(simpleList.getSelectedIndex()));
            display.setCurrent(simpleScreen);
        }

        if (c == CMD_EXIT) {
            destroyApp(false);
            notifyDestroyed();
        }

        if (c == CMD_BACK) {
            display.setCurrent(simpleList);
        }

    }
}
```

In order to track selection and navigation events in standard LCDUI components, mechanism of `Commands` is used. Midlet implements `CommandListener` interface and selection logic is implemented in `commandAction` method. LCDUI doesn't maintain stack concept of screens, developer can just directly to switch between displayable components. Screenshots of functional application on full touch Series 40 device are below.

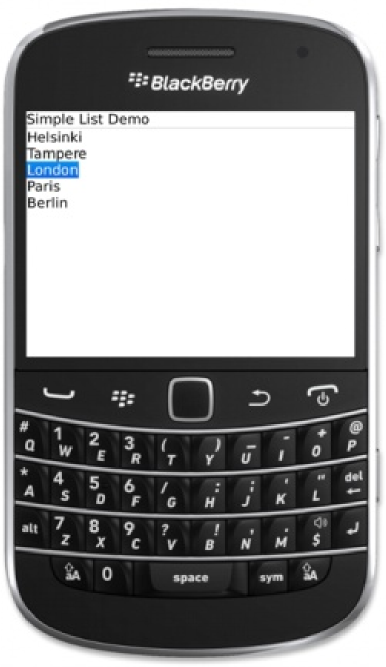




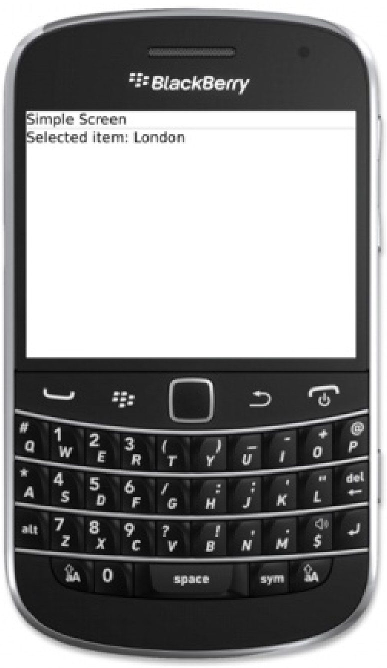
Complete downloadable code sample for NetBeans 7.2 and Nokia SDK 2.0 for Java is available here:

File:SimpleListDemoS40.zip

Interesting aspect of portability between BlackBerry and Series 40, that once project for Series 40 has been created, it can be easily ported back to BlackBerry (excluding Nokia specific JSRs). For example, following screenshots are from resulting Series 40 Midlet which was just recompiled as BlackBerry application project.



Simple List Demo
Helsinki
Tampere
London
Paris
Berlin



Simple Screen
Selected item: London

It looks identical to original BlackBerry project with exactly same behavior. Full source code of Midlet version for BlackBerry Java 7.1 SDK (Eclipse Plug-in) can be downloaded here:

File:SimpleListDemoMidletBB.zip

Adding maps to User Interface example

To make things interesting, let's now add some platform specific functionality to our application example. Our list component is using names of the cities as items, so let's modify secondary screen of the application to display actual map of the selected city.

Both BlackBerry and Series 40 developers can use platform specific Map APIs. Both require special access privileges, but in a bit different form. BlackBerry developers require acquiring special signing keys, in order to deploy applications with Map APIs on device (but not on emulator). Series 40 developers can request free authentication token for each application.

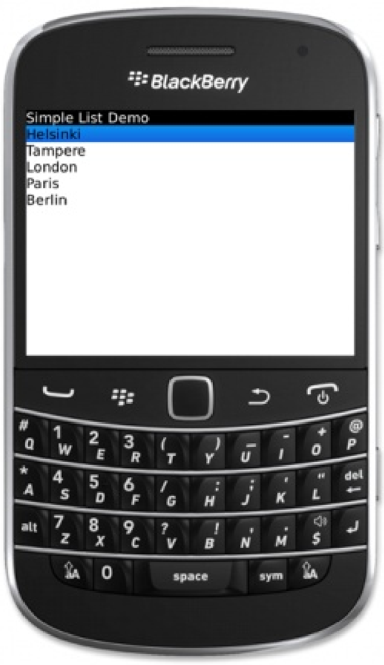
But, from development perspective, those APIs provide pretty similar (at least on basic level) functionality, so porting process is really straightforward. Let's start with modifying our BlackBerry project. First step would be to modify our detailed screen and add Map component, or `MapField` and BlackBerry terms.

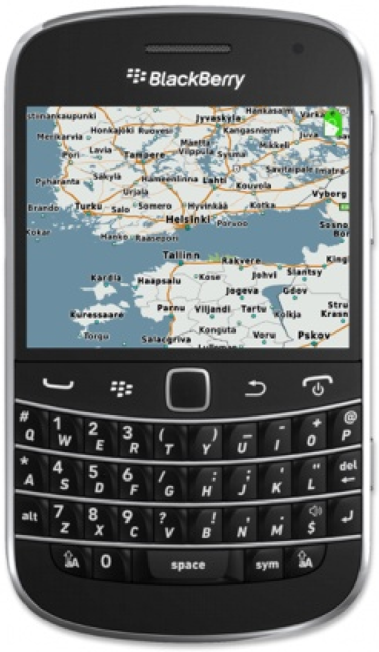
```
import net.rim.device.api.lbs.maps.model.MapPoint;
import net.rim.device.api.lbs.maps.ui.MapAction;
import net.rim.device.api.lbs.maps.ui.MapField;
import net.rim.device.api.ui.container.FullScreen;
import net.rim.device.api.ui.container.MainScreen;

public class SimpleMapScreen extends MainScreen {

 private MapField map;
 private MapAction action;

 public SimpleMapScreen() {
  super( FullScreen.DEFAULT_CLOSE | FullScreen.DEFAULT_MENU );
        map = new MapField();
        action = map.getAction();

        add(map);
 }
 public void setLocation(MapPoint point){
  action.setCenterAndZoom(point, 10);

 }
}
```

Class `SimpleMapScreen` is using `MapField` in order to display actual map. Custom method `setLocation()` takes `MapPoint` as argument and drives Map to display specified location. In our main application screen, there is also modification: `locations[]` array of MapPoints introduced with actual physical coordinates of cities in our list component. When city is selected and `Command` event is fired, before displaying `SimpleMapScreen`, new coordinates are provided.

```
import net.rim.device.api.command.Command;
import net.rim.device.api.command.CommandHandler;
import net.rim.device.api.command.ReadOnlyCommandMetadata;
import net.rim.device.api.lbs.maps.model.MapPoint;
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.ui.component.table.SimpleList;
import net.rim.device.api.ui.container.MainScreen;

public final class AppMainScreen extends MainScreen
{
 private SimpleList listField;
 private SimpleMapScreen simpleMapScreen;
 private MapPoint locations[] = new MapPoint[5];


    public AppMainScreen()
    {
     super( MainScreen.NO_VERTICAL_SCROLL);
        setTitle("Simple List Demo");

        listField = new SimpleList(getMainManager());
        listField.add("Helsinki");
        listField.add("Tampere");
        listField.add("London");
        listField.add("Paris");
        listField.add("Berlin");

        locations[0] = new MapPoint(60.171617, 24.941368);
        locations[1] = new MapPoint(61.500000, 23.766667);
        locations[2] = new MapPoint(51.507222, -0.127500);
        locations[3] = new MapPoint(48.856700,  2.350800);
        locations[4] = new MapPoint(52.500556, 13.398889);


        listField.setCommand(new Command(new CommandHandler()
        {

            public void execute(ReadOnlyCommandMetadata metadata, Object context)
            {
                showMapScreen(listField.getFocusRow());
            }
        }));
    }

    private void showMapScreen(int index){
     if (simpleMapScreen == null){
      simpleMapScreen = new SimpleMapScreen();
     }
     simpleMapScreen.setLocation(locations[index]);
     UiApplication.getUiApplication().pushScreen(simpleMapScreen);
    }
}
```

Following screenshots illustrate working application in action.



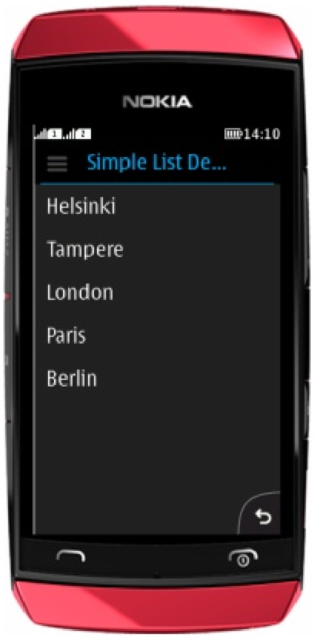


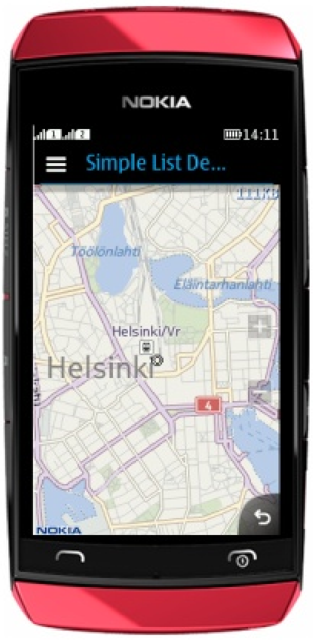
Full source code for BlackBerry project is here:

File:MapListDemoBB.zip

Modifications of Series 40 project are also quite minimal. Component used for displaying map is `MapCanvas`. It can be used as regular displayable, so there is no need for `Form` class anymore. Array for storing cities coordinates is similar to one on BlackBerry side. When `List` item is selected, new coordinates are supplied for `MapDisplay` class and actual map screen is shown.

```
import com.nokia.maps.common.ApplicationContext;
import com.nokia.maps.common.GeoCoordinate;
import com.nokia.maps.map.MapCanvas;
import com.nokia.maps.map.MapDisplay;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.StringItem;
import javax.microedition.midlet.*;

public class ListMapDemo extends MIDlet implements CommandListener {

    private Display display;
    private List simpleList;
    private MapCanvas mapCanvas;
    private MapDisplay mapDisplay;
    private GeoCoordinate locations[] = new GeoCoordinate[5];

    private static final Command CMD_EXIT = new Command("Exit", Command.EXIT, 1);
    private static final Command CMD_BACK = new Command("Back", Command.BACK, 1);

    public ListMapDemo(){
        ApplicationContext.getInstance().setAppID("<AppID>");
        ApplicationContext.getInstance().setToken("<Token>");


        locations[0] = new GeoCoordinate(60.171617, 24.941368, 0);
        locations[1] = new GeoCoordinate(61.500000, 23.766667, 0);
        locations[2] = new GeoCoordinate(51.507222, -0.127500, 0);
        locations[3] = new GeoCoordinate(48.856700,  2.350800, 0);
        locations[4] = new GeoCoordinate(52.500556, 13.398889, 0);


        display = Display.getDisplay(this);
        simpleList = new List("Simple List Demo", List.IMPLICIT);
        simpleList.append("Helsinki", null);
        simpleList.append("Tampere", null);
        simpleList.append("London", null);
        simpleList.append("Paris", null);
        simpleList.append("Berlin", null);
        simpleList.addCommand(CMD_EXIT);
        simpleList.setCommandListener(this);

        mapCanvas = new MapCanvas(display) {

            public void onMapUpdateError(String description, Throwable detail,
                boolean critical) {
                // an error has occurred during map rendering
            }

            public void onMapContentComplete() {
            }
        };

        mapDisplay = mapCanvas.getMapDisplay();
        mapDisplay.setZoomLevel(14, 0, 0);

        mapCanvas.setCommandListener(this);
        mapCanvas.addCommand(CMD_BACK);
    }


    public void startApp() {

        display.setCurrent(simpleList);
    }

    public void pauseApp() {
    }
    public void destroyApp(boolean unconditional) {
    }

    public void commandAction(Command c, Displayable d) {

        if (c == List.SELECT_COMMAND) {

            mapDisplay.setCenter(locations[simpleList.getSelectedIndex()]);
            display.setCurrent(mapCanvas);
        }

        if (c == CMD_EXIT) {
            destroyApp(false);
            notifyDestroyed();
        }

        if (c == CMD_BACK) {
            display.setCurrent(simpleList);
        }

    }
}
```





Full NetBeans project for Nokia SDK 2.0 for Java can be found here:

File:ListMapDemoS40.zip