Volume 3, Issue 1
January 2006

BlackBerry Developer Journal

main()

"No matter how much evidence exists that seers do not exist, suckers will pay for the existence of seers."

The Seer-Sucker Theory: The Value of Experts in Forecasting
J. Scott Armstrong, Published in Technology Review, June/July 1980, 16-24

Armstrong concluded that experts are generally awful at predicting change, breed the inability to accept new views, and ignore disconfirming evidence that does not fit within their preconceived notions.

In James Surowiecki's recent book, "The Wisdom of Crowds" (Anchor Publishing, 2004), the author takes The Seer Sucker Theory much further.

Surowiecki's contention is that "many are smarter than the few, and collective wisdom shapes business, economies, societies and nations." In brief, an expert or a group of experts have very specific knowledge and expertise. If you arrange for an expert, or group of experts, to provide advice in some broad areas, chances are high that their advice will be wrong. If you arrange for a highly diverse group of people to provide their opinions, they will bring together a wider mix of knowledge and experience than possible by a group of like-minded experts. By averaging out their advice and using the average to set direction, chances are higher that the direction chosen will be correct.

Surowiecki's thesis can be validated in several ways. For example, many companies have gone sour because of bad decisions, rotten management, greed and tunnel vision by a select group within. Many government policies have failed to work because they were cooked up for political reasons by individuals with similar beliefs and experience. Major private and public sector scandals erupt through the efforts of a select group of like-minded people. Pricing within stock, commodity and consumer markets are set through the efforts of a wide body of people and groups with dissimilar backgrounds. Traffic flow occurs on highways and streets throughout the world purely because of the efforts of a wide selection of unrelated people. Overall, people, when unhindered by pressure from special interest groups and the media, can make exceptional decisions on a collective basis.

The same philosophy should apply to the development of most commercial software. For example, I've created thousands of commercial and custom software applications. Of those, only a few have gone on to long-term success largely because they were developed with massive levels of user input. The products developed at the request of experts with a good idea always turned out to be a waste of time.

Before starting the wheels in motion to create a commercial application, determine what market exists, and what the end users really want. At the alpha and beta stages of development, include as many end users as possible in your testing to catch errors and help refine the look, feel and functionality. The cost of getting end-user input and involvement is low compared to the benefits that can be realized by doing it right.

As always we are interested in your comments, suggestions, letters, and anything else that you feel would be of benefit to our developer community.

Please feel free to email us at **Editor@BlackBerryDeveloperJournal.com**.

BlackBerry Developer Journal team

BlackBerry Resources

The following are third-party sites that BlackBerry developers may find beneficial in their eternal search for information, software, source code, reviews, and sage advice from like-minded souls. Please feel free to drop us a line at **Editor@BlackBerry-DeveloperJournal.com** if you would like us to consider the inclusion of your site in this column.

"Mobile PC Guide to BlackBerry" Book
by Bill Foust

ISBN: 0789733439
Paperback: 240 pages
Publisher: Que
http://www.quepublishing.com/bookstore/product.asp?isbn=0789733439&rl=1

"Professional BlackBerry" Book
by Craig J. Johnston, Richard Evers

ISBN: 0-7645-8953-9
Paperback: 336 pages
Publisher: Wiley
http://ca.wiley.com/WileyCDA/WileyTitle/productCd-0764589539.html

Important Notice: This information may contain references to third party sources of information, hardware, software, products or services, and/or third party websites (the "Third Party Information.") Any Third Party Information that is provided with or without Research In Motion Limited or its affiliated companies ("RIM") products and/or services is provided "as is." RIM makes no representation, warranty or guarantee whatsoever in relation to the Third Party Information and RIM assumes no liability whatsoever in relation to the Third Party Information even if RIM has been advised of the possibility of such damages or can anticipate such damages. The inclusion of Third Party Information herein does no imply endorsement by RIM of the Third Party Information or the third party in any way. RIM does not control and is not responsible for any Third Party Information, including without limitation, the content, accuracy, intellectual property issues, compatibility, trustworthiness, legality, decency, links or any other aspects of the Third Party Information..

Visualization as a Development Tool

Richard Evers, Editor

Visualization and goal setting are two powerful tools that can be used to create exceptional software. The basic theory is that if you want something to happen, regularly visualize (make a mental picture of) what you want in precise detail, and then set goals to gradually get what you are visualizing.

I've used these techniques with varying levels of success since childhood. The key is to be creative and positive at every turn, accept responsibility for events that have occurred in your life, be prepared to work hard, set goals that you believe you can reach, make achieving your goals your primary responsibility, and have the skills necessary to accomplish your goals. The true power of this technique is found when the visualizer does most or all of the work.

Most children use visualization and goal setting to get what they want. For example, when throwing a tantrum in a public location: they visualize, set goals, are willing to work as hard as necessary to get their way, and use their existing skills to get it. The one area where this technique falls apart is negativity. Parents may give in to the little tyrant occassionally, but over the long haul, the technique will fail.

I have successfully used visualization and goal setting for many years when developing software. For me, this works by setting goals to achieve and then visualizing achievement of those goals.

Setting goals is often the most difficult part because the entire project has to "crystallize" in my mind before I can do anything of worth. This means that I have to learn as much as possible about the project so that I can visualize how it will look, feel and work when it's finished. While waiting for crystallization to occur as the facts come in, I usually create components that are sure to be necessary within the project.

When the magic moment arrives where I can fully visualize the project, I further refine visualization to understand how everything fits together from a technical perspective. From there I visualize the code flow required to make it all happen. After that, it often becomes an exercise in visualizing the actual code, then typing what has been completed in my head. Sadly, reaching this level of visualization is not easy as it seems.

For me, this technique only works on fairly challenging projects where I can isolate myself from interruptions and remain fully focused. This means minimal interruptions from family, friends, coworkers, phone calls, email, network activity, boredom, and more. A total lockdown works best. Even better, working at night and through the wee hours of the morning in a windowless room, especially after working on the project all day, lowers my defenses enough to allow full visualization to occur.

Most developers rarely, if ever, achieve this state of focus when developing software. This is usually due to interruptions, distractions and meetings, a 9-to-5 development mentality, shared project responsibilities, bad management, poor design and analysis, a weak grasp of the language or target platform, a lack of interest, or a bad work environment.

It's often impossible to visualize a project even when conditions are perfect when the requirements or timelines are unrealistic or sketchy. At times the challenge of trying to meet unrealistic expectations can lead to breakthroughs of the highest order. For me, one of the biggest motivators in the past has been trying to accomplish goals that seem unattainable by most sane developers.

This can be shown in two projects where one seemed possible but turned out to be impossible, where the other seemed impossible but proved to be possible.

The first project involved creating a lossless compression algorithm that eclipsed all known techniques used at the time. I started with the assumption that data can be repeatedly organized into patterns that can be compressed. By repeat reorganization and compression, extreme levels of compression should be possible.

Dozens of techniques were developed to organize and compress data. At one stage I developed a series of techniques that generated extremely tiny files. Unfortunately, my decompressor proved that the overall method of combining techniques was flawed. After repeatedly failing to produce extremely tiny files, I realized that it is impossible from a mathematical perspective. The overhead associated with each data organization technique combined with limits on reduced permutations greatly curbs maximum lossless compression ratios. The up-side of this work was that my algorithms were terrific for data encryption.

The second project involved development of extremely high-speed search algorithms where size and speed were equally critical due to media and platform constraints.

I started with the usual indexing techniques and found they all were too consumptive and slow. I modified the techniques to further improve performance and reduce size, and found that the improvements were still not good enough. I developed several custom techniques to improve speed and reduce overhead, all with limited success. Finally, I developed a winning technique that used less than 20% of the storage of a standard bTree index, delivering results upwards of 1,600% faster, with no limit on key length or number of search words or phrases, and operating with minimal memory overhead. All in all, it was pretty impressive.

By the time I came across my winning search design, I was truly in the zone of visualization. No hurdle remained to stop me from fully visualizing how every piece of the various applications I wrote would fit together. For the grueling year it took me to complete this project, I kept every line of code, and every technical detail of how it worked, in my head for instant access. In that way, any difficulties could be instantly sorted out because I was a human debugger. It was strange but true, and probably common among many top developers today.

B Reel: From concept to design

Michael Clewley, Editor

Everybody loves a good flick, and there's nothing better than watching your favourite movie over and over again in the comfort of your own home. Most people have their own stockpile of favorite movies for their own personal enjoyment. How many times have you been in a conversation with someone and they ask you “oh, so what movies do you have?”. You sit there, pull out the fingers and toes, start counting and can't even get past the first hand. You've forgotten your favourite collection! Don't worry; I've been there with you, with that same dumbfounded and disappointing look. So there's no better idea for an application than a movie-tracking database. It's a good way to learn how to create a basic application, it’s fun, and it’s something you're sure to use. By the end, you should have a really cool application that includes components you can use in future applications.

Now you ask yourself, where do I begin? Well the name of the game is tracking movies, so what's the first thing that comes to mind? Here are a few things that came to my mind:

- Title
- Studio
- Format (DVD/VHS)
- Did I lend it to a friend?
- If I lent it, to who?

I think those are the key pieces of information. But thinking about it more there were some other ideas. Who knows if they'll make it in the final application though! They include:

- length of movie
- comments / notes
- year

I'm sure the list could go on and on. Each person has different needs from an application. At some point you need to stop the daydreaming and make a decision, or you'll never get anything done!

Now that we've given our data model some thought, let's turn the ideas into actual code.

Start off by doing any necessary imports and then the class declaration. We know that this movie data is going to be store on the device. Make sure to import the necessary class.

```
package com.bbdj.samples.breel;

import net.rim.device.api.util.Persistable;

final public class BReelMovieElement implements Persistable {
```

The only other piece of information to note is that we are going to make the class final because its not going to be extended and it allows the compiler to optimize the class to create smaller code.

Now we take the ideas of the data and create variables for our class.

```
//variables
private byte _lentTo[];
private byte _movieTitle[];
private byte _productionCompany[];

private int _genre;
private int _format;
private boolean _lentOut;  //false
```

Let’s examine what happened in the code above. Each piece of information has been analyzed and turned into a primitive data type that best matches the functionality. As a personal preference, I like to group variables according to type. While some prefer to group alphabetically, it makes no difference. Data that is going to be plain text has been defined as byte arrays because it's more efficient to convert a string to a byte array and store the array than it is to store a String object. The format will be a number that represents the video format. And lastly, a boolean value is being used to represent whether the movie has been lent out.

The next piece is the constructor, but we are not actually initializing anything here so we'll omit it for now. It should be noted that the variables that we have defined at the class level are automatically initialized to their default values.

Moving along, we need ways to set the data and also extract the current values. In this application there are only five data types, so we use the set and get methods for each variable. If this class had many different variables then we might want to use an alternative method of retrieving the information, such as getInt(), getString(), getBoolen() methods.

```
/**
 * @return the format of the movie
 */
/* package */ int getFormat() {
  return _format;
}

/**
 * @return True if lent out, false otherwise
 */
/* package */ boolean getLentOut() {
  return _lentOut;
}

/**
 * @return the genre of the movie
 */
/* package */ int getGenre() {
  return _genre;
}

/**
 * @return name of the person the movie was lent to
 */
/* package */ String getLentTo() {
  if (_lentTo == null || _lentTo.length == 0) {
    return "";
  }
  return new String(_lentTo);
}

/**
 * @return title of the movie
 */
/* package */ String getMovieTitle() {
  if (_movieTitle == null || _movieTitle.length == 0) {
    return "";
  }
  return new String(_movieTitle);
}

/**
 * @return production/film company name
 */
/* package */ String getProdCompany() {
  if (_productionCompany == null || _productionCompany.length == 0) {
    return "";
  }
  return new String(_productionCompany);
}
```

We now have a way of retrieving the data from the instance of the class. Everything shown above is pretty straightforward, but there might be some things that are not familiar to you. At the start of each method I have placed a comment **/* package */**. This means that the method has default permissions access and I just wanted to note that. The system does some optimization if the method modifier is omitted, and allows for a smaller file size for the application when completed. Lastly, the byte arrays values are all converted to a String object before being returned, with a check put in place to ensure that we don't return a null.

```
/**
 * @param value - number assigned to the format
 */
/* package */ void setFormat(int value) {
  _format = value;
}

/**
 * If the value is true and set it false it will also clear the lentTo field.
 * @param value true if lent out, false otherwise
 */
/* package */ void setLentOut(boolean value) {
  _lentOut = value;
  if (!_lentOut) {
    setlentTo("");
  }
}

/**
 * @param value the genre of the movie
 */
/* package */ void setGenre(int value) {
  _genre = value;
}

/**
 * @param data name of the person the movie is lent too
 */
/* package */ void setlentTo(String data) {
  _lentTo = data.getBytes();
}

/**
 * @param data movie title
 */
/* package */ void setMovieTitle(String data) {
  _movieTitle = data.getBytes();
}

/**
 * @param data production/film company
 */
/* package */ void setProdCompany(String data) {
  _productionCompany = data.getBytes();
}
```

With the ability to set the data in the object, our BReelMovieElement is now complete. The only note on the above methods would be those methods that are receiving String objects. From within the method we are converting the String object to a byte array using the getBytes() method.

With our element class now ready to roll, we need to a way to track a list of all the different movies. This list will implement the List interface, read and save the data from persistence, and sort the list as well. Let's dive in.

```
package com.bbdj.samples.breel;

import java.util.Vector;
import net.rim.device.api.collection.List;
import net.rim.device.api.system.*;
import net.rim.device.api.ui.Graphics;
import net.rim.device.api.ui.component.ListFieldCallback;
import net.rim.device.api.ui.component.ListField;
```

```
/**
 *
 */
class BReelMovieList implements List, ListFieldCallback {

  /**
   * The ID used to store the data in persistence
   */
  //com.bbdj.samples.breel.BReelMovieList
  private static final long B_REEL_MOVIE_LIST = 0x2713ab1c1a7c4c15L;

  /**
   * Comparator for the movie elements
   */
  private BReelMovieElementComparator _comparator;

  /**
   * A Vector containing the BReelMovieElement
   */
  private Vector _list;

  /**
   * The persistent store object
   */
  private PersistentObject _persist;

  /**
   * Reads the list from persistence or creates a new one if not found
   */
  public BReelMovieList() {
    _persist = PersistentStore.getPersistentObject(B_REEL_MOVIE_LIST);
    Object obj = _persist.getContents();

    if( obj == null) {
      obj = new Vector();

      //store the vector
      _persist.setContents( obj );
      _persist.commit();
    }
    _list = (Vector)obj;

    _comparator = new BReelMovieElementComparator();
  }

  /**
   * Saves the data to the store
   */
  private void commit()
  {
    _persist.commit();
  }
```

Let's cover what just happened. To start we have the usual package and import statements, along with the class declaration. The *B_REEL_MOVIE_LIST* variable is a constant that will be used as a unique ID for the persistent database. The *_list* variable will contain all of the BReelMovieElements, and *_persist* is the persistent store object. The constructor has been marked as private because we don't want to be able to instantiate the object from outside of the class. The constructor is also responsible for retrieving the data from the store. The last piece included in the code above is a commit() method which will be responsible for actually storing the vector to persistence.

```
////////////////////////////////////////////////
//       Implementation of Readable List       //
////////////////////////////////////////////////

public Object getAt(int index)
{
  return _list.elementAt(index);
}
public int getAt(int index, int count, Object[] elements, int destIndex)
{
  return 0;   //not implemented
}
public int getIndex(Object element)
{
  return _list.indexOf(element);
}
public int size()
{
  return _list.size();
}
////////////////////////////////////////////////////////
// Implementation of WriteableList Interface           //
////////////////////////////////////////////////////////

public void add( Object element )
{
  //check if the element is grouped, if not, do it now
  if( !ObjectGroup.isInGroup( element )) {
    ObjectGroup.createGroup( element );
  }

  //we are going to add an element in a sorted order
  int size = size();
  for (int cnt = 0; cnt < size; ++cnt) {
    BReelMovieElement compare = (BReelMovieElement)_list.elementAt(cnt);

    if (_comparator.compare(element, compare) < 0) {
      insertAt(cnt, element);
      return;
    }
  }

  //the element wasn't added, so it goes to the bottom of the list
  _list.addElement(element);
  commit();
}

public void insertAt( int index, Object element )
{
  //check if the element is grouped, if not, do it now
  if( !ObjectGroup.isInGroup( element )) {
    ObjectGroup.createGroup( element );
  }

  _list.insertElementAt(element, index);
  commit();
}

public void remove( Object element )
{
  _list.removeElement(element);
```

```
    commit();
  }

  public void removeAll()
  {
    _list.removeAllElements();
    commit();
  }

  public void removeAt( int index )
  {
    _list.removeElementAt(index);
    commit();
  }
```

These methods are just an implementation of the List interface for all of the writeable accessors. We start off with the readable methods, and then move to the writeable methods. With all of the writable methods you need to make sure that you commit the data to persistence. The add method is also doing a bit of extra work. We don't want to just add a new movie element into the mix of things, so we'll add it in a sorted way based on the movie title.

```
  ////////////////////////////////////////////////////
  // Implementation of ListFieldCallback Interface       //
  ////////////////////////////////////////////////////

  public void drawListRow(ListField listField, Graphics graphics, int index, int y, int width) {
    //put a check in as a precaution so that we don't throw an out of bounds exception
    int size = size();
    if (size > 0 && index < size) {
      BReelMovieElement element = (BReelMovieElement)getAt(index);
      graphics.drawText(element.getMovieTitle(), 0, y, 0, width);
    }
  }
  /**
   * Used to assit with the search
   * @param listField
   * @param index
   * @return Movie Title
   */
  public Object get(ListField listField, int index) {
    BReelMovieElement element = (BReelMovieElement)getAt(index);
    return element.getMovieTitle();
  }
  public int getPreferredWidth(ListField listField) {
    return Graphics.getScreenWidth();
  }

  /**
   * Performs a search of data in the field
   * @param listField <description>
   * @param prefix <description>
   * @param start <description>
   * @return <description>
   */
  public int indexOfList(ListField listField, String prefix, int start) {
    return listField.indexOfList(prefix,start);
  }

  public void insertItem(int index, Object element) {
  }
}
```

The last part of our list class is implementing the ListFieldCallback. We want to make the list responsible for the drawing of the elements, which is where the drawListRow() method comes into play. When an element is drawn we are choosing to just draw the movie title. Other methods worth noting are the indexOffList(), and get() methods. These two methods combined will allow simple searching of the list field, adding some extra power to the application and making it easy for the user to find what they want.

The comparator is pretty simple. The main concern is the compare() method which compares the movie titles from the sorted list.

```
package com.bbdj.samples.breel;

import net.rim.device.api.util.Comparator;

class BReelMovieElementComparator implements Comparator {
  public BReelMovieElementComparator() {
    // Do nothing.
  }

  public int compare( Object o1, Object o2 ) {
    // compare the movie titles
    BReelMovieElement element1 = (BReelMovieElement)o1;
    BReelMovieElement element2 = (BReelMovieElement)o2;

    return (StringUtilities.compareToIgnoreCase(element1.getMovieTitle(),
      element2.getMovieTitle()) );
  }

  public boolean equals( Object obj ) {
    // We don't need to know this
  return false;
  }
}
```

We have successfully built the foundation for the application, and are ready to move onto the next phase. Let's take a break and move into the user interface scene. The first thing we want to create is a method that allows us to keep a single tracking point for the list of our movies. Also down the road this class will become useful for different features, but that's for the second chapter of this story that will be published in the next issue.

```
package com.bbdj.samples.breel;

import net.rim.device.api.system.RuntimeStore;

final class BReel
{
  /**
   * ID used to store the object in the runtime store
   */
  //com.bbdj.samples.breel.BReel
  private static final long APP_ID = 0x1db8890f0df27f91L;

  /**
   * List of all the Movie Elements
   */
  private BReelMovieList _list;

  /**
   * Constructor
   */
  private BReel() {
    _list = new BReelMovieList();
```

```
  }

  /**
   * Retrives a single instance of BReel
   * @return current instance of BReel
   */
  /* package */ static BReel getInstance() {
    RuntimeStore store = RuntimeStore.getRuntimeStore();
    Object bReel = store.get(APP_ID);

    if (bReel == null) {
      bReel = new BReel();
      store.put(APP_ID, bReel);
    }

    return (BReel)bReel;
  }

  /**
   * @return The data from persistence
   */
  /* package */ BReelMovieList getList() {
    return _list;
  }
}
```

Everything here is pretty basic, with the possible exception of the getInstance(). This is a static method that retrieves the single instance of BReel so that we only work with one instance of the BReelMovieList. As you can see, the constructor is private so only the getInstance() method can invoke it. Once we have created the instance we also provide a method from pulling out the list for the screens to use.

The next class is currently a very simple class, but it is the main driver class, or the entry point for the application. When the user clicks on the icon the main method is invoked and 3...2...1...action!

```
package com.bbdj.samples.breel;

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

final class BReelApp extends UiApplication
{
  public static void main(String args[]) {
    new BReelApp().enterEventDispatcher();
  }

  public BReelApp() {
    pushScreen(new BReelScreen());
  }
}
```

Finally! We actually get to see what something is going to look like. BReelScreen at the moment is a basic screen, but it's still better than nothing. The screen itself only contains a list field that will list all of the movies in our database. On top of that we have some menu items which take care of opening, delete, or creating a new movie entry.

```
package com.bbdj.samples.breel;

import net.rim.device.api.system.GlobalEventListener;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.MainScreen;
```

```
final class BReelScreen extends MainScreen implements GlobalEventListener {

  /**
   * ID used to identify when a screen update has to take place
   */
  //com.bbdj.samples.breel.BReelScreen
  public static final long SCREEN_UPDATE = 0x7d2f34f5416d5bb5L;

  //menu item varaiables
  private MenuItem _deleteItem = new MenuItem("Delete", 500001, 600000) {
    public void run() {
      deleteMovie();
    }
  };
  private MenuItem _newItem = new MenuItem("New", 500000, 600000) {
    public void run() {
      addNewMovie();
    }
  };
  private MenuItem _openItem = new MenuItem("Open", 500001, 500000) {
    public void run() {
      openMovie();
    }
  };

  //UI Variables
  private BReelMovieList _list;
  private ListField _listField;

  /* package */ BReelScreen() {
    //set the title of the screen
    setTitle("BReel Movie Database");

    //get the move list
    _list = BReel.getInstance().getList();

    //create the list field and set the callback
    _listField = new ListField();
    _listField.setSearchable(true);
    _listField.setCallback(_list);
    _listField.setSize(_list.size());

    //add the field to the screen
    add(_listField);

    //add the listener to the screen
    UiApplication.getUiApplication().addGlobalEventListener(this);

    addMenuItem(_newItem);
  }
```

Let's take a step back from the production floor and head to the editing room to figure out how the shot went. We start with some basic declarations and imports. Note that the screen extends MainScreen, which means that our screen will have a default set of behaviours. Added to that we implement the GlobalEventListener, which will be used to update the ListField from classes that are outside of the screen but can affect the data.

A key is defined for our screen update event, and then we jump into defining the menu items for the screen and what exactly they are going to do. When a movie is highlighted we want the default menu action to be the open action, so we are giving it a lower priority value than the other two menu items.

Shifting into the constructor, the masterpiece begins. For an esthetic effect we are adding a title to the screen. Now create the list field, allow it to be searchable, set the callback and

size and add it to the screen. The last step that the constructor must do is to register itself to be a global event listener so that it gets notified when an update is necessary.

```
//make sure we remove the listener
public void close() {
  UiApplication.getUiApplication().removeGlobalEventListener(this);
  super.close();
}

public void makeMenu(Menu menu, int instance) {
  if (_list.size() > 0) {
    menu.add(_openItem);
    menu.add(_deleteItem);
  }
  super.makeMenu(menu, instance);
}
```

In these methods we are just adding to the default behaviour of the screen. When close() is called and there is only one screen on the display stack, then the screen is popped from the stack and system.exit(0) is called. Before the application is ended we want to un-register the global event listener. Overriding the makeMenu() class allows us to control what menu items are displayed to the user. This is important because you wouldn't want an open, or delete, menu item when there is nothing in the list.

```
/**
 * Gives user ability to add a new movie to the list
 */
/* package */ void addNewMovie() {
  UiApplication.getUiApplication().pushScreen(new BReelMovieElementScreen());
}

/**
 * Deletes a movie eletment from the list
*/
/* package */ void deleteMovie() {
  //get the selected item, and remove it
  int index = _listField.getSelectedIndex();
  _list.removeAt(index);

  //set the size of the list file which causes invalidate
  _listField.setSize(_list.size());
}

/* package */ void openMovie() {
  BReelMovieElement element =
    (BReelMovieElement)_list.getAt(_listField.getSelectedIndex());
  UiApplication.getUiApplication().pushScreen(
    new BReelMovieElementScreen(element, false));
}
```

Actions have now been defined for the menu items. One of the reasons that the actions have been placed in a separate method is because your could override the keychar() method on the listField or screen to capture user input. Most commonly, the enter key would be mapped to the open item, and the delete/backspace key mapped to the delete item. The add and open actions both use the same screen but just pass in different parameters for the required action.

```
////////////////////////////////////////////
//    Implementation of GlobalEventListener    //
////////////////////////////////////////////
public void eventOccurred(long guid, int data0, int data1, Object object0, Object object1) {
```

```
    if (guid == SCREEN_UPDATE) {
      _listField.setSize(_list.size());
    }
  }
}
```

The last thing that we need to worry about in this screen is the implementation of the global event listener. In this case it's simple. Just check if the guid passed into the eventOccurred() method is the same as the SCREEN_UPDATE ID. If that's the case, then size the list field accordingly which causes the list field to invalidate itself.

The climax to this plot has been reached and we are heading to the conclusion, but don't worry. There's plenty of action left to be had! The BReelMovieElementScreen provides a way for the user to create and review detailed information about any movie in the list on the BReelScreen.

```
package com.bbdj.samples.breel;

import net.rim.device.api.system.ApplicationManager;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

class BReelMovieElementScreen extends MainScreen implements FieldChangeListener
{
  //menu items
  private MenuItem _deleteItem = new MenuItem("Delete", 500000, 500000) {
    public void run() {
      doDelete();
    }
  };

  private MenuItem _saveItem = new MenuItem("Save", 500000, 500000) {
    public void run() {
      //only close the screen if the save was successful
      if (onSave()) {
        close();
      }
    }
  };

  //variables
  private BReelMovieElement _element;
  private boolean _updated;
  private boolean _isNew;

  //field variables
  BasicEditField _titleField;
  BasicEditField _prodCompanyField;
  BasicEditField _lentToField;
  ObjectChoiceField _genreChoiceField;
  ObjectChoiceField _formatChoiceField;
  ObjectChoiceField _lentChoiceField;

  BReelMovieElementScreen() {
    this(new BReelMovieElement(), true);
  }
```

More of the basics start this scene, with a few small surprises. Like the previous screen we are again extending MainScreen, but this time we are implementing a FieldChangeListener which will add some excitement to the story shortly. Basic menu declarations and variables used in the class along with the screen's fields. The last part shows a constructor which seems to be leading to something bigger.

```
BReelMovieElementScreen(BReelMovieElement element, boolean isNew) {
  _element = element;
  _isNew = isNew;

  setTitle("B Reel Movies");

  //make sure we have no null strings
  String title = _element.getMovieTitle();
  String prodCompany = _element.getProdCompany();
  String lentTo = _element.getLentTo();
  int genreChoice = _element.getGenre();
  int formatChoice = _element.getFormat();
  boolean lentChoice = _element.getLentOut();

  //add the fields to the screen according to the element data
  _titleField = new BasicEditField("Title: ", title, BasicEditField.DEFAULT_MAXCHARS,
    BasicEditField.NO_NEWLINE);
  add(_titleField);

  _prodCompanyField = new BasicEditField("Film Company: ", prodCompany,
    BasicEditField.DEFAULT_MAXCHARS,
    BasicEditField.NO_NEWLINE);
  add(_prodCompanyField);

  String genreChoices[] = {"Horror", "Drama", "Sci-Fi", "Comedy", "Action"};
  _genreChoiceField = new ObjectChoiceField("Genre: ", genreChoices, genreChoice);
  add(_genreChoiceField);

  String formatChoices[] = {"DVD", "VHS"};
  _formatChoiceField = new ObjectChoiceField("Format:", formatChoices, formatChoice);
  add(_formatChoiceField);

  String lentChoices[] = {"No", "Yes"};
  _lentChoiceField = new ObjectChoiceField("Lent Out:", lentChoices,
    (!lentChoice ? 0 : 1));
  _lentChoiceField.setChangeListener(this);
  add(_lentChoiceField);

  if (lentChoice) {
    _lentToField = new BasicEditField("Lent To: ", lentTo,
      BasicEditField.DEFAULT_MAXCHARS,
      BasicEditField.NO_NEWLINE);
    add(_lentToField);
  }

  //add the menu items
  addMenuItem(_saveItem);

  if (!_isNew) {
    addMenuItem(_deleteItem);
  }
}
```

While it might seem like there is a lot of code, everything that occurs in this constructor is pretty basic. The main purpose is to create the fields and add them to the screen with pre-populated information depending on the calling constructor. There are a few things to point out here. The *_lentToField* is only displayed if the movie is lent out, and the delete menu item is only added if we have opened a movie from the previous screen.

```
public void close() {
  if (_updated) {
    ApplicationManager.getApplicationManager().postGlobalEvent(BReelScreen.SCREEN_UPDATE);
  }
  super.close();
}

/**
 * Deletes the element from the list
 */
/* package */ void doDelete() {
  int answer = Dialog.ask(Dialog.D_DELETE);

  if (answer == Dialog.DELETE) {
    BReelMovieList list = BReel.getInstance().getList();
    list.remove(_element);
    _updated = true;
    close();
  }
}

public boolean onSave() {
  //get the data from the screen
  BReelMovieElement newElement = new BReelMovieElement();
  String title = _titleField.getText();

  //if the title is blank then we can't save
  if (title == null || title.length() == 0 ) {
    Dialog.inform("Cannot have a blank title!");
    return false;
  }

  newElement.setMovieTitle(title);
  newElement.setProdCompany(_prodCompanyField.getText());

  newElement.setGenre(_genreChoiceField.getSelectedIndex());
  newElement.setFormat(_formatChoiceField.getSelectedIndex());

  if (_lentChoiceField.getSelectedIndex() == 1) {
    newElement.setLentOut(true);
  }

  if (newElement.getLentOut()) {
    newElement.setlentTo(_lentToField.getText());
  }

  //remove the old element and add the new one
  BReelMovieList list = BReel.getInstance().getList();
  list.remove(_element);
  list.add(newElement);
  _updated = true;

  return true;
}
```

The three methods above are a combination of overriding the screen's default methods for additional functionality and adding our own method for the delete function. Closing this screen will fire an update to the BReelScreen, but only if the element has been saved, or deleted. The delete method displays a confirmation dialog so that the user doesn't accidentally delete the element. The onSave() is just responsible for pulling the information from the fields on the screen and dropping it into our BReelMovieList. It should be pointed

out that the element can't be saved if the movie title is blank. That's just some artistic flavoring, and is not necessary for your application.

```
  ////////////////////////////////////////////
  //    Implementation of FieldChangeListener    //
  ////////////////////////////////////////////

  public void fieldChanged(Field field, int context) {
    //the listener is only added the a choice field, so cast it
    ObjectChoiceField choiceField = (ObjectChoiceField)field;

    //if the choice is 0 (NO) clear the field
    if (choiceField.getSelectedIndex() == 0) {
      getDelegate().delete(_lentToField);
      _lentToField = null;
    } else if (choiceField.getSelectedIndex() == 1) { //if the choice is 1 (YES), add the field
      _lentToField = new BasicEditField("Lent To: ", "", BasicEditField.DEFAULT_MAXCHARS,
        BasicEditField.NO_NEWLINE);
      add(_lentToField);
    }
  }
}
```

That's it! Oh wait, watch the credits, and hidden in the credits is our implementation of the FieldChangeListener. The lent out field is an ObjectChoiceField which lets the user toggle between YES or NO. When the field is changed to YES, the listener is notified of the change event, and the lentToField is added to the screen. When the field is changed to NO, the field is removed from the screen.

Yes. You have reached the end of the story and the application is complete, so get your thumbs ready to type your movie collection into your BlackBerry device. And as always in the movie business these days TO BE CONTINUED…….Watch for B Reel II: Attack of the Feature Enhancements.

IPD File Format

Garry Seifried, Research In Motion

Note: *The format of IPD files described in this article may change at any time at the sole discretion of Research In Motion. Research In Motion will not provide technical support on issues related to IPD files and their format.*

The Need for Speed

One of the challenges of working with wireless applications is that delivering large amounts of data over the wireless network can be both time-consuming and expensive.

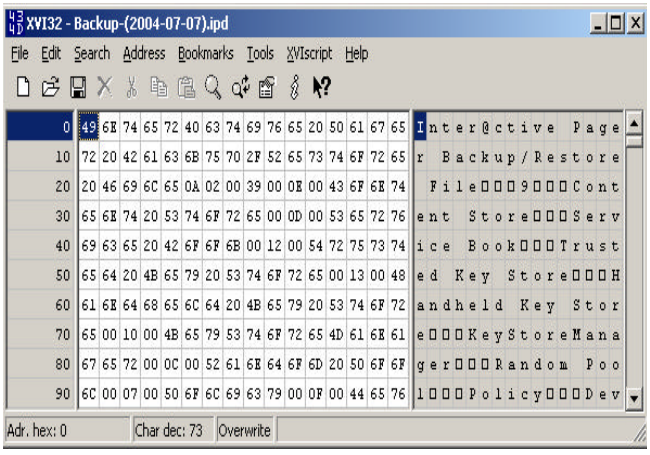
Suppose that you want to put the contact information for an organization on the BlackBerry wireless device. Normally, we only put the most essential data on the device, and wirelessly pull additional less-essential data as may be needed.

Let's assume, for the sake of this article, that the entire set of contact information is required.

BlackBerry Desktop Manager Backup/Restore

We've all used the BlackBerry Desktop Manager to perform backup of our device data - perhaps prior to performing an upgrade to the device software. Periodically, we've also needed to use that backup to restore our data. But have you ever had a look inside one of those IPD files?

That's one nasty looking file!



Using the IPD Format for Bulk Loads

What if the most effective and efficient way of getting a large amount of data onto the device were to use the IPD format? Using this format, we could programmatically create a file that could be used by the desktop manager in a restore operation.

The structure of the IPD file shown above is as follows:

```
Inter@ctive Pager Backup/Restore File
<Line feed> - 1 byte - value 0A
<Version> - 1 byte - value 02
<Number of databases in file> - 2 bytes
<Database name separator> - 1 byte - value 00
<Database name block#1>
<Database name block#2>
.
<Database name block#n>
<Database data block#1>
<Database data block#2>
.
<Database data block#n>
```

Where each database name block is of the form:

<Database name length>	2 bytes The length includes the terminating null
<Database name>	As long as the name length

And each database data block is of the form:

<Database ID>	2 bytes Zero-based position in the list of database name blocks
<Record length>	4 bytes
<Database version>	1 byte
<DatabaseRecordHandle>	2 bytes
<Record unique ID>	4 bytes
<Field length #1>	2 bytes
<Field type #1>	1 byte
<Field data #1>	As long as the field length
<Field length #m>	2 bytes
<Field type #m>	1 byte
<Field data #m>	As long as the field length

Parse that IPD

In manually parsing the file, we can see some structure that matches the binary structure just outlined:

```
Number of databases: 57
Database[ 0 ] = Content Store
Database[ 1 ] = Service Book
.
```

```
Database[ 56 ] = WTLS Options

Record for database 0:
  Database version: 0x01
  Record handle:    0x0001
  Unique ID:        0x24F07B6D
    Field:
      Length: 0x0002
      Type:   0x01
      Data:
        2F 00
    Field:
      Length: 0x0004
      Type:   0x03
      Data:
        21 00 00 20             /
    Field:
      Length: 0x0007
      Type:   0x05
      Data:
        66 6F 6C 64 65 72 00    folder

Record for database 0:
  Database version: 0x01
  Record handle:    0x0002
  Unique ID:        0x00000007
    Field:
      Length: 0x0007
      Type:   0x01
      Data:
        2F 68 6F 6D 65 2F 00    /home/
    Field:
      Length: 0x0004
      Type:   0x03
      Data:
        31 00 00 20
    Field:
      Length: 0x0007
      Type:   0x05
      Data:
        66 6F 6C 64 65 72 00    folder
```

Limits in the IPD Format

IPD files have a record length restriction of 128K bytes. To get around this limitation, we came up with a record structure that would separate the content into separate records each under the 128K limit.

The Unique ID values map to this record type mapping for the eight record types:

```
SESSION_TYPE = 1
TRACK_TYPE = 3
SPEAKER_TYPE = 5
SESSION_TRACK_TYPE = 7
TRACK_SESSION_TYPE = 9
KEYNOTE_LIST_TYPE = 10 (0A)
DAY_SESSION_TYPE = 11 (0B)
SESSION_SPEAKER_TYPE = 13 (0D)
```

This is how the records appear in the file (fields omitted):

```
Number of databases: 1
Database[ 0 ] = BBConferenceGuideData
Record for database 0:
  Database version: 0x01
  Record handle:    0x0001
  Unique ID:        0x00000001

Record for database 0:
  Database version: 0x01
  Record handle:    0x0008
  Unique ID:        0x0000000D
```

What is the relationship to a Session record as defined in this structure versus an instance of a Session record from a database? We used the fields of the IPD record for each occurrence of a session.

Writing the Length Before the Content

Since you need to output the record length before the record content, you must process the content first and then get the content length.

This is done by creating an intermediate file that contains the content. The method that closes the file must also return the content length.

After writing the content length, append the intermediate file content to the final IPD file.

This also applies to the content in a record, such as an image that needs to be processed to determine the length of the image content. There is some extra overhead in processing content twice but it is only done while creating the file where overhead doesn't impact much.

Little Endian Hex Values

Lengths are stored in little endian format. This means that the low-order byte of the number is stored at the lowest address, and the high-order byte at the highest address (the word is stored 'little-end-first').

The little-endian format applies to the following fields:

- record length
- db version
- db handle
- record type.

The following conversion is used to get the byte values:

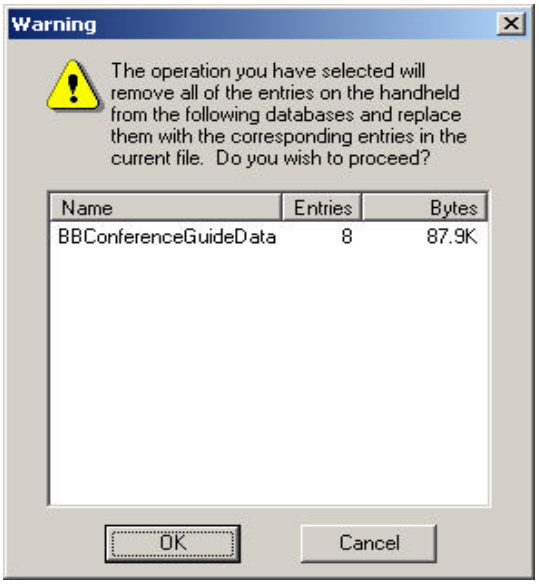
```
//Size as little-endian hex value
byte b1 = Convert.ToByte
  (size&Byte.MaxValue);
byte b2 = Convert.ToByte
  (size>>8&Byte.MaxValue);
byte b3 = Convert.ToByte
  (size>>16&Byte.MaxValue);
byte b4 = Convert.ToByte
  (size>>24&Byte.MaxValue);
```

After conversion, each byte is written in order:

```
loader.write(b1);
loader.write(b2);
loader.write(b3);
loader.write(b4);
```

Using the Simulator for Backup/Restore

Prior to BlackBerry Java Development Environment (JDE) v4.0, simulator testing configuration for backup/restore required the installation of a serial loopback driver to be defined for a port (say PORT5) as well as few other steps.



With BlackBerry JDE v4.0, the BlackBerry Device Simulator provides a direct and convenient way for backup/restore testing:

- Launch the BlackBerry Device Simulator
- Select the Simulate menu
- Check the USB Cable Connected.

Application Handling the Restore

Performing Backup/Restore involves the BlackBerry Synchronization API. The Synchronization API provides the following interfaces that need to be implemented:

SyncConverter

- Converts data between SyncObject-format on the device and a serialized format required on the desktop.

SyncCollection

- A collection of synchronization objects.

SyncObject

- An object that can be backed up and restored to the user's computer.

SyncConverter and SyncCollection Manager

SyncConverter and SyncCollection are interfaces, so we chose to implement the interfaces in a DataStoreSyncManager class.

DataStoreSyncManager implements SyncCollection

Since a SyncCollection is a collection of SyncObject used for backup/restore and synchronization, we need to provide implementations for the methods defined in the SyncCollection interface.

Notable methods include:

beginTransaction()

- Starts a transaction which is required to execute synchronizations that involve a large number of data records

endTransaction()

- Ends a transaction which is required to execute synchronizations that involve a large number of data records

getSyncConverter()

- Returns the instance of the DataStoreSyncManager

getSyncName()

- Returns the database name

getObjectCount()

- Returns the number of sync objects

getSyncObjects()

- Returns a SyncObject[]

getSyncObject(int uid)

- Returns a SyncObject by UID

getSyncVersion()

- Returns 1

removeAllSyncObjects()

- Clears the DataStore object and returns true

addSyncObject()

- Adds a SyncObject to the DataStore by using the UID to identify the data type, and using the DataStore methods for the appropriate data type (e.g. setSessionSpeaker() method)

Since we are performing a bulk-load and not a synchronization, we can provide minimal implementations as shown below for some of the methods. For a backup/restore, we can return false; for Synchronization you must provide a full implementation.

```
isSyncObectDirty () {return false; }
removeSyncObject () { return false; }
setSyncObjectDirty() {}
clearSyncObjectDirty()  {}
```

DataStoreSyncManager implements SynchConverter

This class implemented the convert(..) method that Extracts a SyncObject from the synchronization data and converts a SyncObject into synchronization data.

DataStoreSyncObject implements SyncObject

The application needs to have a class defined that represents the structure of the .IPD database records. For our example, eight record types have been defined in our database, so we effectively have eight instances of SyncObjects.

A SyncObject must have a unique ID, which is a 32-bit value that is contant for the lifetime of the object. For this UID we implement a getUID() method that returns a value for one of the record types. For example, SESSION_TYPE = 1 as defined above.

PersistentStore contains Persistable objects

Since we've gone to the trouble of getting the data to the device, we now have to save it to the PersistentStore. To do so, implement a class that extends the Persistable interface. Again, the structure of the DataStore class needs to reflect the structure of the data contained in the .IPD file.

In our example, we have eight different data records defined so we will need a DataStore class capable of managing those eight different data types.

We chose to have byte[][] for Session, Track and Speaker data as the records contained character data.

For SessionTrackMap, TrackSessionMap, DaySessionMap, SessionSpeakerMap we chose int[][] since we had relationships expressed as integer index values.

For KeyNoteSessionList we chose int[] as we only had the Session index value in the list. So we have a DataStore structure on the device that supports the .IPD format.

The implementation of methods to access the particular records and record relationships is up to the designer of the application. In our case, we designed the data to support the application UI so that we had methods for Sessions, SessionSpeakers, KeyNoteSpeakers, Tracks, etc.

Sample Code

Loader.cs

Contains the data retrieval and data output code.

```
namespace BulkLoader {
  /// <summary>
  /// Summary description for Class1.
  /// </summary>
  class Loader {
    // Constants for record types - session shown here
    public static int SESSION_NUMBER_TYPE = 0;
    public static int SESSION_TYPE = 1;

    public void doLoading() {
      // Create output file and content
      byte dbVersion = 1;
      short dbHandle = 0;

      // Final output written to Loader1
      Loader loader1 = new Loader();
      loader1.createFile("BulkLoad-Output.ipd");

      // Prefix data for every IPD file
      loader1.loadFile("BulkLoad-Prefix1.ipd");

      // Data access provided by DBReader
      DBReader rdr = new DBReader();

      // Intermediate data written to BulkLoad-Data.ipd files
      Loader loader = new Loader();
      loader.createFile("BulkLoad-Data.ipd");
      dbHandle++;

      // An ArrayList of SessionInfo objects turned into SessionContent
      ArrayList list = rdr.GetSessions();
      int listSize = list.Count;
      SessionContent sessions = new SessionContent(listSize);
```

```
      for (int i=0;i<listSize;i++) {
        SessionInfo info = (SessionInfo) list[i];
        Session data = new Session(info);
        sessions.AddMember(i, data);
      }

      if (listSize>0) {
        sessions.WriteInfo(loader);
      }

      // Get the size so far and output
      int bytes = loader.closeFile();
      OutputData(loader1, bytes, dbVersion, dbHandle, SESSION_TYPE);
    }

    /**
     * Create output from the data
     */
    private void OutputData(Loader loader, int bytes, byte dbVersion,
      short dbHandle, int recordType) {
    // Database type
    short dbase = 0;
    loader.Write(dbase);

    // Add the size of the Prefix2 portion which is 7 bytes
    bytes+=7;

    // Write the size as little-endian hex values
    byte b1 = Convert.ToByte(bytes&Byte.MaxValue);
    byte b2 = Convert.ToByte((bytes>>8)&Byte.MaxValue);
    byte b3 = Convert.ToByte((bytes>>16)&Byte.MaxValue);
    byte b4 = Convert.ToByte((bytes>>24)&Byte.MaxValue);
    loader.Write(b1);
    loader.Write(b2);
    loader.Write(b3);
    loader.Write(b4);

    // DBVersion - append as little-endian hex values
    b1 = Convert.ToByte(dbVersion&Byte.MaxValue);
    loader.Write(b1);

    // DBHandle - append as little-endian hex values
    b1 = Convert.ToByte(dbHandle&Byte.MaxValue);
    b2 = Convert.ToByte((dbHandle>>8)&Byte.MaxValue);
    loader.Write(b1);
    loader.Write(b2);

    // RecordType - append as little-endian hex values
    b1 = Convert.ToByte(recordType&Byte.MaxValue);
    b2 = Convert.ToByte((recordType>>8)&Byte.MaxValue);
    b3 = Convert.ToByte((recordType>>16)&Byte.MaxValue);
    b4 = Convert.ToByte((recordType>>24)&Byte.MaxValue);
    loader.Write(b1);
    loader.Write(b2);
    loader.Write(b3);
    loader.Write(b4);

    // Data Content
    loader.loadFile("BulkLoad-Data.ipd");
  }
}
```

Content.cs

Contains definitions for the IPD file content: Session, Tracks, Speakers.

Session items are shown here.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace BulkLoader {

  /**
   * MapContent defines the Record Group: Length, Type, length value that prefixes every record.
   */
  public abstract class MapContent {
    short length;
    protected byte type;     // Record type
    protected int[] members; // Used for Record length

    public MapContent() {}


    public void SetContentLength(int len) {
      length = Convert.ToInt16(len);
    }

    public void AddContentMember(int pos, int item) {
      if (members != null) {
        members[pos] = item;
      }
    }

    /// <summary>
    /// Our objects need to know how to write themselves
    /// </summary>
    /// <param name="loader"></param>
    protected void WriteContentMember(Loader loader) {
      loader.Write(length);
      loader.Write(type);
      int j = members.Length;

      for (int i=0; i<j;i++) {
        loader.Write((int) members.GetValue(i));
      }
    }
  }

  /**
   * SessionContent, Session and SessionInfo manage Sessions.
   */
  public class SessionContent : MapContent {
    Session[] info;

    /// <summary>
    /// Ctor defines the size of Session[]
    /// </summary>
    public SessionContent(int num) {
      type = Convert.ToByte(Loader.SESSION_NUMBER_TYPE);
      info = new Session[num];
      members = new int[1];
    }
```

```
    /// <summary>
    /// Add an item to the list
    /// </summary>
    public void AddMember(int pos, Session data) {
      info[pos] = data;
    }

    /// <summary>
    /// Our objects need to know how to write themselves
    /// </summary>
    /// <param name="loader"></param>
    public void WriteInfo(Loader loader) {
      // Content setup
      SetContentLength(4);
      AddContentMember(0, info.Length);
      WriteContentMember(loader);
      IEnumerator e = info.GetEnumerator();

      while (e.MoveNext()) {
        Session data = (Session) e.Current;
        data.WriteInfo(loader);
      }
    }
  }

  /// <summary>
  /// Session provides the infoType, length and SessionInfo
  /// </summary>
  public class Session {
    short infoLength;
    byte infoType = Convert.ToByte(Loader.SESSION_TYPE);
    SessionInfo info;

    public Session(SessionInfo data) {
      info = data;
      infoLength = Convert.ToInt16(info.Length());
    }

    /// <summary>
    /// Our objects need to know how to write themselves
    /// </summary>
    /// <param name="loader"></param>
    public void WriteInfo(Loader loader) {
      loader.Write(infoLength);
      loader.Write(infoType);
      info.WriteInfo(loader);
    }
  }

  /// <summary>
  /// SessionInfo contains the details of a session
  /// </summary>
  public class SessionInfo {
    long start;
    long duration;
    byte keynote;
    string info;

    public SessionInfo(long start, long duration, byte keynote, string info) {
      this.start = start;
      this.duration = duration;
      this.keynote = keynote;
      this.info = info;
    }
```

```
    /// <summary>
    /// The length is the size of the SessionInfo object
    /// </summary>
    /// <returns></returns>
    public short Length() {
      return Convert.ToInt16(8 + 8 + 1 + info.Length);
    }

    /// <summary>
    /// Our objects need to know how to write themselves
    /// </summary>
    /// <param name="loader"></param>
    public void WriteInfo(Loader loader) {
      loader.Write(start);
      loader.Write(duration);
      loader.Write(keynote);
      loader.Write(info);
    }
  }
```

Loader.cs

This class is responsible for output of data using a BinaryWriter.
Methods exist to write various data types: char, int, byte, short, long, String.

```
using System;
using System.IO;
namespace BulkLoader {

/// <summary>
/// Loader manages Binary files.
/// </summary>
public class Loader {
  // Our writer
  BinaryWriter writer = null;

  // Output length counter
  int bytes = 0;

  public Loader() {}

  /// <summary>
  /// Create BinaryWriter from a filename
  /// </summary>
  /// <param name="fileName"></param>
  public void createFile(String fileName) {
    try {
      writer =  new BinaryWriter(File.Open(fileName, FileMode.Create));
    } catch (Exception e) {
      Console.Out.WriteLine("Exception: " + e.Message);
    }
  }

  /// <summary>
  /// Load a file
  /// </summary>
  /// <param name="fileName"></param>
  public void loadFile(String fileName) {
    if (fileName.Length==0){
      return;
    }

    loadFile(fileName, false);
  }
```

```
/// <summary>
/// Load a file, counting the bytes of the file
/// </summary>
/// <param name="fileName"></param>
/// <param name="countBytes"></param>
public void loadFile(String fileName, bool countBytes) {
  if (fileName.Length==0){
    return;
  }

  BinaryReader rdr = null;

  Try {
    rdr =  new BinaryReader(File.OpenRead(fileName));
    byte b;

    for (;;){
      b = rdr.ReadByte();
      writer.Write(b);

      if (countBytes){
        bytes++;
      }
    }
  } catch (EndOfStreamException) {
    ; // do nothing
  }
  finally {
    if (rdr != null) rdr.Close();
  }
}

/// <summary>
/// Close an intermediate file and return the file size
/// </summary>
/// <returns>file size</returns>
public int closeFile() {
  if (writer != null) {
    try {
      writer.Close();
    } catch (Exception e) {
      Console.Out.WriteLine("Exception: " + e.Message);
    }
  }
  return bytes;
}

/// <summary>
/// Write a char - 1 byte
/// </summary>
/// <param name="val"></param>
public void Write(char val) {
  writer.Write(val);
  bytes+=1;
}

/// <summary>
/// Write an int - 4 bytes
/// </summary>
/// <param name="val"></param>
public void Write(int val) {
  writer.Write(val);
  bytes+=4;
}
```

```
/// <summary>
/// Write a short - 2 bytes
/// </summary>
/// <param name="val"></param>
public void Write(short val) {
  writer.Write(val);
  bytes+=2;
}

/// <summary>
/// Write a byte - 1 byte
/// </summary>
/// <param name="val"></param>
public void Write(byte val) {
  writer.Write(val);
  bytes+=1;
}

/// <summary>
/// Write a long - 8 bytes
/// </summary>
/// <param name="val"></param>
public void Write(long val) {
  writer.Write(val);
  bytes+=8;
}

/// <summary>
/// Write a String - string length
/// </summary>
/// <param name="val"></param>
public void Write(string val) {
  Write(val, val.Length);
}

/// <summary>
/// Write a String - string length
/// </summary>
/// <param name="val", "len" ></param>
public void Write(string val, int len) {
  StringReader rdr = new StringReader(val);

  for (int i=0;i<len;i++) {
    int c = rdr.Read();
    writer.Write(Convert.ToByte(c));
    bytes++;
  }
}
```

The BlackBerry Graphical User Interface: Part 2 - Direct Screen Drawing

Mark Sohm, Research In Motion

In part one of this two part series we explored the many displayable objects that are present in the BlackBerry API set. These include Screens, Managers and Fields. By using these components and containers, it is possible to create an interface that looks and feels like a standard application for a BlackBerry wireless device. However, there are cases where you may want to deviate from the norm to provide a custom field, a custom image or some kind of animation. Part two of this series explores the use of the Graphics class (net.rim.device.api.ui.Graphics) to draw directly to a screen or field.

There are two main areas where the Graphics class is typically used: at the field level and the screen level. Drawing at the field level allows you to create a unique field such as a custom looking button or a graph. Drawing at the screen level allows full control of the screen and what is presented to the user. This approach is typically used for applications that make use of animation or multimedia content, such as games and graphic demos.

The Graphics class

Whether creating a custom screen or a custom field, the bulk of the work of the Graphics class is performed in the paint method of the Field class (remember that the Screen class extends Field). A device invokes the paint method when a field, or part of the field, is to be redrawn.

The paint method can be overridden to allow the creation of a custom field, or to modify an existing field. One of the simplest things we can do by overriding the paint method is to change the colour used. For example we can change the text colour of a RichTextField to green by doing the following:

```
//Green - The format for colour is 0x00RRGGBB.
Long myColour = 0x00008800;

RichTextField colourChange =
  new RichTextField
    ("The quick brown fox jumps over the
      lazy dog.")
{
  public void paint(Graphics graphics)
  {
    //Change the colour of the text in
    //the RichTextField to green.
    graphics.setColor(myColour);
    //call super.paint() to carry out the
    //default painting of the field
    super.paint(graphics);
  }
};
```

We can also change values used within the paint method between calls to paint to create animation. Continuing from the sample above we could do the following:

```
//Change the colour to red.
myColour = 0x00FF0000;
colourChange.invalidate();
```

You can force a field, or part of a field, to be redrawn by invoking one of its invalidate methods. Calling the invalidate() method will mark the entire field as invalid by instructing the entire field to be redrawn. Calling invalidate(int x, int y, int width, int height) allows you to mark a certain region of a field as invalid. This can speed up painting if only a certain area of a field has changed, as only the region specified will be redrawn. This can provide a substantial performance increase for large fields, or fields that are redrawn in rapid succession such as when performing animation.

Drawing graphics with Graphics

Now it's time to create some graphics with Graphics! The approach you take to create your graphics should vary based on their usage pattern. Overriding the paint method for a field or screen that changes often - such as during an animation - works great but is less desirable when creating an image that may rarely change but will be reused throughout an application. Here is an example of creating a dynamic Bitmap that is used to populate a BitmapField. In this example we create a Bitmap that contains a large purple square and a green circle.

```
//Instantiate a bitmap 100x100 pixels in size.
Bitmap myBitmap = new Bitmap(100, 100);

Graphics myGraphics = new Graphics(myBitmap);

//Change the colour to purple.
myGraphics.setColor(0x00550055);

//Draw a filled rectangle starting at 0,0
//that is 100 pixels square.
myGgraphics.fillRect(0, 0, 100, 100);

//Change the colour to green.
myGraphics.setColor(0x00004400);

//Draw a filled circle that has a radius of 25.
myGraphics.fillArc(50, 50, 25, 25, 0, 360);

//Instantiate a BitmapField with the
//created bitmap.
BitmapField myBitmapField =
  new BitmapField(myBitmap);
```

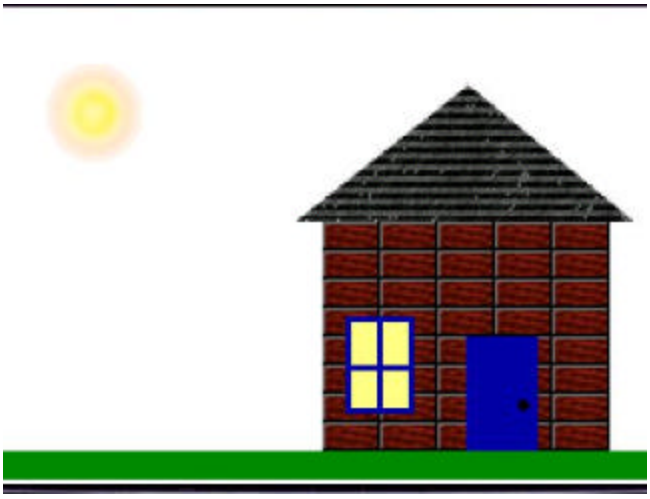
This technique allows us to store the bitmap and display it again in other areas at a later time. If the image is fairly detailed and requires multiple calls to several of the graphics methods, you can save some processing time by creating the image once and reusing it. This could be faster than recreating an image every time by using graphics methods from within the paint method and allows you to reuse the existing bitmap image.

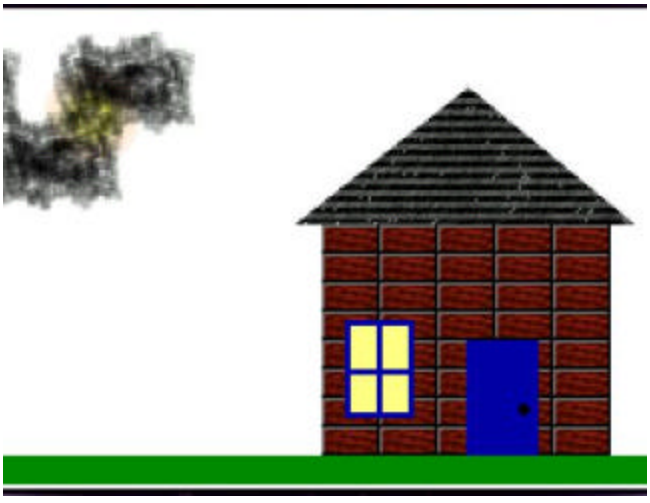
Although ideal for images that remain fairly constant, this approach is not optimal for highly dynamic content.

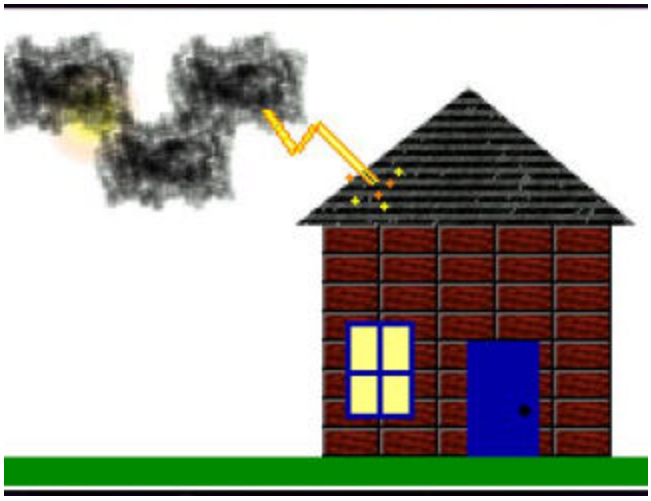
Animation!

Animation used by a game or multi-media application can push any device to its limits. Whether you are using a desktop PC, laptop or a mobile device such as a BlackBerry device you want to ensure that you squeeze the most out of the platform. The remainder of this article will explore using the Graphics class and other BlackBerry APIs to create and optimize an animated sequence.

The following images depicts some of the animation we are going to dissect. The complete source code for this application is available for download.







The calculations and drawing of the animation take place in the AnimationScreen class. AnimationScreen extends FullScreen and contains a thread called AnimationThread. AnimationThread controls what is drawn in each frame, per-

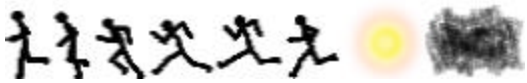
forms all of the manipulation calculations and calls invalidate to repaint the screen.

There are three png images included in the application.

One image contains a single brick used to build the house:

Another holds the roof:

A third contains the sun, cloud and each frame of the runner (there are 6 runner images in total). The sun, cloud and runner are all of the same height, which allows us to easily place them side-by-side in a rectangular image.



As discussed in the BlackBerry Developer Journal article (Volume 2, Issue 1, Jan-2005), Rooster Revealed!, this allows for a size reduction. It reduces the overhead (image headers, meta-data, etc.) that would exist if all 8 elements were in their own image and helps reduce the application size. When everything is complete the total size of the application including images is only 17.6 kb. The images are defined as members of the AnimationScreen class.

```
private final static Bitmap _imageItems =
  Bitmap.getBitmapResource("imageItems.png");
private final static Bitmap _brick =
  Bitmap.getBitmapResource("brick.png");
private final static Bitmap _roof =
  Bitmap.getBitmapResource("roof.png");
```

The paint method of FullScreen - which is inherited from the Screen, then Field class - is overridden to allow us to draw directly to the screen. Let's take a look at some of the Graphics methods that are used to draw the scene.

The first thing we do in the paint method is obtain the screen dimensions. The dimensions can be used to scale aspects of the animation and make use of all available screen real estate. The animation is designed to not leave any blank areas along the top, bottom or sides of the screen on high resolution BlackBerry devices but still fit on the screen on BlackBerry devices that have a smaller screen resolution.

```
int width = Graphics.getScreenWidth();
int height = Graphics.getScreenHeight();
```

We also define four integers that are used in counters, for loops and as holders for calculations for x and y values used in the application.

```
int count, count2;
int tempX, tempY;
```

Storing these in a defined variable can be more efficient than calculating them inline, as the inline calculation will create a temporary int variable for each calculation. This would result in the creation of a lot of garbage that can add up quickly in a loop, and even faster in a loop inside the paint method that is called repeatedly during the animation. In the sample code below we also save two arithmetic operations by storing the result in a variable. An example of both the recommended and improper methods follow.

Note: *The following code example is not part of this application*

```
//Inline calculations (BAD!):
graphics.fillRect(0, width - 10, width, 10);
graphics.fillRect(0, width - 10, width, 20);
graphics.fillRect(0, width - 10, width, 30);
...

//Use defined variables (GOOD!):
tempY = width - 10;
graphics.fillRect(0, tempY, width, 10);
graphics.fillRect(0, tempY, width, 20);
graphics.fillRect(0, tempY, width, 30);
...
```

Moving back to the animation, the first element we draw is the grass. We set the drawing colour to green and use the fillrect method to draw a painted rectangle at the bottom of the screen. The rectangle is 10 pixels high and the width of the screen, and is placed at the bottom of the screen.

```
//Set the colour to green
graphics.setColor(0x00008800);

//Draw the grass.
tempY = height - 10;
graphics.fillRect(0, tempY, width, 10);
```

The next element drawn is the sun. The sun is placed at the same coordinates regardless of the screen resolution. The sun is part of the imageItems bitmap which also contains the cloud and all 6 runner frames. The sun is 43 pixels wide, 40 pixels tall and starts 170 pixels from the left of the image. We use the drawBitmap method and only draw the portion of the imageItems bitmap that contains the sun.

```
graphics.drawBitmap
  (20, 10, 37, 40, _imageItems, 170, 0);
```

Now it is time to draw the house. The first item we tackle is the house wall. Since we are using an image that contains a single brick, it must be drawn multiple times to make up the entire house wall. This is done using two loops that repeatedly call the drawBitmap method. Here we pre-populate the count variable and decrement in each loop. Comparing to 0 is faster than comparing to another value, and pre-decrementing count (--count) is also faster than post-decrementing (count--).

```
for(count = 8; count > 0; --count)
{
  for (count2 = 5; count2 > 0; --count2)
  {
    //The x coordinate.
```

```
    tempX = 220 - count2 * 20;

    //The y coordinate.
    tempY = height -10 - count * 10;

    //Draw a brick.
    graphics.drawBitmap
      (tempX, tempY, 20, 10, _brick, 0, 0);
  }
}
```

The remainder of the house (roof, window and door) is drawn next. The roof is drawn using drawBitmap, the door using fillRect and fillArc and the window using fillRect. These methods have been used in the sample on page one of this article, so the code is not shown in this article. If you would like to see how they are drawn, the complete source code is available for download. The clouds and runner are also drawn using drawBitmap, so we'll move past them and onto the lightning.

The lightning is not shown on every frame. Before starting to draw the lightning we check to ensure the drawLightning variable is true (this value is changed in the AnimationThread that controls the animation sequence). There are multiple calls used to make up the bolt of lightning, but the only new one is the drawLine method.

```
graphics.drawLine(100, 30, 110, 45);
```

The drawLine method accepts four ints. The first two are the x and y coordinates for the start point of the line and the second two are the x and y coordinates for the end point of the line.

Now that the paint method was drawing what was required and the AnimationThread was set up to control the sequence, it was time to see how everything looked. Part of the goal of this application was also to see how efficient we could make the application and see how quickly the animation could run. I think this goal was realized. It started off chugging along, but after all optimizations were complete I actually had to slow it down by inserting some sleep cycles. Let's see what we can do to speed up the frame rate on a BlackBerry device.

Optimizing to get the maximum frame rate

The first step in optimizing the application was to ensure that the recommendations from the Java Bits and Pieces articles from previous BlackBerry Developer Journal issues had been applied. With these in place the animation was able to move along at an average rate of 17 frames per second on a BlackBerry 7100 Series device running BlackBerry Device Software v4.0. Now it was time to refine the graphics to get things drawing faster.

The first step implemented was to control what is drawn on the screen each frame and only redraw elements that are changing. This meant adding controls to allow the application to toggle each element from being drawn or not. For example, in most frames the sun, clouds and house don't change. Therefore it is a waste of processing to redraw them every frame. The drawBitmap method calls were taking the most time, so they were first on the list to optimize.

By controlling on what frames the sun and clouds were drawn - removing them from any frames where they don't change - the animation sped up from 17 to 25 frames per second, a significant increase! After adding controls for the house, we moved up to 34 frames per second, shaving 400 ms off the total animation time. Not bad when all of the animation calculations took less than 2 seconds!

After making changes to control what was drawn on each frame, we have to dig deeper to see what we can do to squeeze an even higher frame rate out of the BlackBerry device. By optimizing the paint method to only draw what is needed, we save time on calls we are making to the Graphics methods.

However, the entire screen is still being redrawn every time the invalidate method is called. In many frames, only a small area of the screen is changing, such as the clouds moving across the sky or the runner running off the screen. By making use of the invalidate(int x, int y, int width, int height) method we can specify a rectangular region of the screen to invalidate. Areas outside of this region will not be redrawn, which can save us some more calculation time. This change isn't appropriate in all frames, such as the full screen yellow flashes during the lightning strike, but can be used in most frames.

After analyzing each frame, the generic invalidate methods were replaced with invalidate (int, int, int, int). The screen coordinates were relatively easy to figure out, as they matched coordinates already used in the paint method. After making this change the calculation time was dropped to just over a second, achieving an amazing 51 frames per second!

Watching the show

With the animation sequence complete and optimized, the only thing left was to sit back and watch what's been created. As you can see, the BlackBerry API set does allow for the creation of some interesting graphical or animated content. All it takes is a bit of testing and tweaking to optimize your content and really make things fly. Hopefully you can now take the concepts detailed in this article and create some truly amazing content. After all, there is only so much my stick-man-level art capabilities can do.

Please email your comments, suggestions and editorial submissions to Editor@BlackBerryDeveloperJournal.com

Using WBXML Parsing to Send Data to Wireless Devices

Rohit Gupta, Research In Motion

The growing popularity and corporate usage of BlackBerry means that an increasing number of organizations are providing wireless access to corporate data for their mobile workers. Although this can have productivity benefits, the cost of transmitting information wirelessly is a consideration when organizations decide which applications and information to make available to their BlackBerry users.

XML is useful for representing complex data in a simplified way that can be displayed in any application. Unfortunately, XML is not always practical to use in wireless environments because XML files tend to be large documents. Transmitting large documents over wireless networks can be costly and time-consuming for users who are downloading information on a wireless device. Wireless binary XML (WBXML) offers a cost-saving alternative to XML documents. WBXML replaces the customized XML tags with a binary value, which significantly decreases the file size. WBXML incorporates the advantages of XML without the disadvantages of the large file size.

Understanding XML

XML is designed to provide more flexible and adaptable information identification. It is extensible because it is not a fixed format like HTML. Instead, XML is a meta language - a language for describing other languages - that allows you to design your own customized markup language for different types of documents.

Advantages of using XML to code data

With XML, application developers can represent complex data in a simple manner. They can completely define their data in XML because it provides fully customized tags. There are no pre-defined or hard-coded tags like HTML. Some of the advantages of using XML to represent data include:

- XML, like Java, is viewed as open source. This means that any application can open an XML document and display its contents.
- XML is as easy to read and write as HTML.
- XML files take approximately the same amount of time to render as HTML files.
- XML provides developers with more customization options and access to more tools than HTML.
- XML seamlessly integrates with multiple applications, making XML a standard type for data definition documents.

Limitations of using XML in wireless application development

XML documents can be large, which reduces its value for sending data to users' wireless devices. An optimal solution involves sending rich data using the quickest and most cost efficient method possible. Wireless binary XML is one solution that helps to reduce the file size and save organizations transmission time and costs.

Understanding WBXML

WBXML provides a way to send documents quickly and more cost effectively while maintaining the advantages of using XML. WBXML is a compact representation of XML and is part of the presentation logic in Wireless Application Protocol (WAP). WBXML significantly improves the efficiency of transmitting XML over narrow bandwidth networks where data size is of paramount importance.

Most of the WBXML format is not human readable; it contains bytes or octets in raw (non-textual or non-encoded) and hexadecimal form. Two hexadecimal numbers represent 1 byte.

Binary form	Hexadecimal form
0100 1000	0x48
1100 1110 0111 1101	0xCE 0x7D

Advantages of using WBXML to send data wirelessly

Using WBXML for sending data wirelessly offers the following advantages:

- Because the tags in WBXML are represented as a binary value, you can save transmission time when sending the document to a wireless device.
- Binary XML documents are considerably smaller than XML documents. This reduces the amount of airtime it takes to send a WBXML document wirelessly, which may result in a lower cost to transmit it.
- Binary XML parsing is about half the size of an XML parsing. The BlackBerry MDS Services creates a WBXML document by taking an XML document and creating two additional corresponding files: a binary XML file and a codebook.

Note: *Refer to the BlackBerry MDS Services log that is included with this article for an example of a codebook.*

About codebooks

The codebook is used after the BlackBerry device receives the binary XML document. A codebook holds the table value for the binary tags. It is usually stored on the device since it is lightweight and the values are matched once the binary document is received. A codebook contains the following three types of tables:

Codebook table type	Description
Tag table	Defines the element values
Start table	Defines the name of the attributes
Value table	Defines the value of the attribute

Codebook creation

Certain conditions must be met before the BlackBerry MDS Services can successfully create a codebook from an existing XML document.

- The XML document needs an external DTD/XSD document because the codebook is generated from the DTD/XSD and the binary XML file is generated from the XML itself.
- The address for the XML document must be referred to in URL format.
- The address in the XML document pointing to the DTD/XSD must be in URL format, not as a local file path.

For example, use: http://localhost:8000/content/text.dtd

… instead of c:/sample/text.dtd

Note: *The codebook is returned from the BlackBerry MDS Services in a 64-bit string type. You must put that string in a 64bit array before you can manipulate it.*

Codebook storage

There are two options for storing the codebook. It can either be persistently stored on the device, or it can be created each time the application is invoked by the user. Do not use the second scenario unless the memory on the device is close to being full. Creating a redundant codebook and sending it every time breaks down the function of WBXML, since the amount of airtime used is not significantly lowered.

The net.rim.device.api.xml.jaxp API

The net.rim.device.api.xml.jaxp is a new API found in the BlackBerry Java Development Environment (JDE) v4.0. The classes found in this API are described in the following table.

Class	Description
DomInternalRepresentation	Represents an XML document and is used to implement the W3C DOM
SAXParserImpl	Using the specified default handler uses this class to parse the content of a given Input Stream instance as XML
WBXMLCodeBookCreation Handler	Extends the default handler and deals with the content of a codebook after it is created
WBXMLParser	Handles all WBXML documents and their attributes
WBXMLWriter	Extends the default handler and implements an XML writer as a SAX parser handler. The main purpose of this class is to handle the WBXML document once it has been parsed
XMLParser	Handles all XML documents that need to be parsed and their attributes
XMLWriter	Same as WBXMLWriter but handles XML documents

Development steps for using WBXML

The following steps guide you through the process of creating an application that can be parsed using WBXML. Refer to the code sample on the BlackBerry Developer Journal site for clarification as you move through the steps.

Step 1: Make sure that the XML file is up to code and fully valid

This step includes verifying that the corresponding DTD/XSD files are created correctly and that the code in the XML file is also correct. There are multiple third-party software products available that can build a DTD on its own after the XML is validated. The XML and its corresponding DTD/XSD must be stored on a web server. The BlackBerry MDS Services needs a URL to find the WBXML file. The address in the XML that points to the DTD/XSD must be a URL in order for the BlackBerry MDS Services to find the codebook.

Note: *If you get a SAX parser exception, make sure the XML document is valid.*

Step 2: Build object classes

These classes include all object references that are found in the XML document. In the sample, there is only one object called Contact. Building object classes makes it easier to control the data that is shown on the BlackBerry device and makes it easier to add and subtract data types. A well-formed XML document contains, at most, one or two major object types and multiple attributes and minor objects within them. For example, in the sample, there is one object type with multiple attributes in it.

Object	Element	Example
Name	First name Last name	John Doe
Phone number	Area code Remaining digits of phone number	414 555-1234

These object classes make the rest of the development easier and less confusing.

Step 3: Handle the parsed file

The default handler dictates what to do with the data after it is parsed and provides a place where the SAX parser can store the parsed data. The default handler is a public class, but it responds like an interface because most of its important functions return null values. You must construct a class that extends the default handler. You can use one of the given classes, like WBXMLWriter, but it has limitations. It does not have enough functions to produce correct output or to correctly handle the parsed data for the sample application. The default handler class can be developed in multiple ways, but the two main functions are startElement() and endElement(). Their default value is to return null but this is where the finding of the data can be done. The function startElement() does something at the start of a tag or element; the endElement() function does something at the end of a tag.

In the sample, at the start of each of the elements, you set the type of data so that you know which data to get and then store under the appropriate setting in the object class. The method characters() returns the data value of an element, including its white space and empty characters. It is important to use the trim method so that white space and empty characters are not collected and displayed. In the sample, the data was extracted and displayed, leaving the class small in size.

Note: *The default handler has more functionality than used in the sample. See the BlackBerry API v4.0 for more information.*

Step 4: Persistent storage of the codebook

In most cases, the persistent classes are small and easy to create. Some developers include persistent classes in their main Java application class, but it is usually easier to create them as an external class. Because most applications involve persistent classes, not much detail is included here. The sample includes an object of a persistent object, which holds the same attributes as a codebook. Because codebooks are not persistent objects, they cannot be stored directly on the device, but must be instantiated as a persistent object. In most cases, this exact persistent class can be used in your development. Even with modifications, a persistent class is relatively simple to create.

Step 5: Develop the application

The main class

Now that you have taken care of the entire external element surrounding the development for WBXML, you are ready to develop the core Java application. This part of the development is similar to developing other applications, so only problematic issues are discussed.

Looking at the attached code, initially you can see all the member variables being defined, the main function is starting the application, and the constructor is setting the title, creating a screen and pushing it and instantiating the connection thread. This activity is all being done in the main class.

Note: *Looking at the code, you can see that UiApplication.getUiApplication().invokeAndWait(new Runnable()) was used multiple times. Only one thread at a time can gain access to an interface component. If you are getting a UI exception check whether you are invoking any screen variables without using this function.*

Converting 64-bit codebook to string

The function getCodeBookFromString takes a 64-bit string containing the codebook. Using the WBXML parser, it creates a codebook handler. This function can be reused in multiple applications that require a codebook. It was created as a standard function because the BlackBerry MDS Services will always have 64-bit string value of the codebook. It is

easier to create a codebook handler than the codebook itself; the string is parsed using the WBXML parser. Remember that the WBXML parser takes an input stream containing a WBXML document and parses it to a default handler. In this case, the default handler is the codebook handler, so now you can use the codebook to parse data.

Adding contacts

The addContact function is customized for each parsing application. This function takes the object type Contact and outputs its relevant data. This function works in collaboration with the contact class. The contacts are being fetched from a vector array, which were stored in it by the default handler class Parser.

Creating a separate thread

The thread class makes the request to the BlackBerry MDS Services for the conversion of XML into WBXML and its corresponding codebook through an HTTP header request. If this is the first time the device is receiving the codebook, you can store it using persistent store. If the codebook already exists, you can reuse the stored codebookas shown in the IF and ELSE statement found in this class.

If the codebook does not exist, make the X-RIM-UseCode-Book true and embed. If the codebook exists, make it true. There is no check to see if the codebook is up to date, this is just a sample, that much detail is not necessary at this point.

From the CodebookHandler, retrieve the tables that are needed for SAX parser, parse the matching XML with the codebook, and retrieve the data. The other HTTP headers are setting the transcoded content to be XML, for the given document, the BlackBerry MDS Services creates its WBXML and codebook. The other header is telling the BlackBerry MDS Services to accept the type of WBXML and the codebook documents. The last step involves doing the necessary functions to the data or displaying it.

Note: *You must use all of the HTTP headers for the WBXML parsing to work. You can have more than the requirement header, but these headers are the minimum requirement.*

Summary

There are two major constraints to sending data wirelessly: the time to render the document and the cost of sending it. With WBXML parsing, you can help minimize these two constraints while still transmitting all of the necessary data. Mobile workers can access critical data at a more cost-effective and faster rate than by using XML documents.

WBXML development does not have to be more difficult than XML or HTML development. Developers with skills in application development can quickly learn how to use WBXML to send data across wireless networks.

You can download the sample from the Developer Journal site to use as a guide in development, and can also enhance it when creating your own custom application.

Visit http://www.blackberry.com/developers for more information about application development for BlackBerry. See the APIs for the BlackBerry JDE v4.0 for more information on additional functions that were not used or discussed for this sample but that might be helpful as you develop your own applications.

Please email your comments, suggestions and editorial submissions to Editor@BlackBerryDeveloperJournal.com

Object Grouping

Mike Kirkup, Research In Motion

One of the most common memory related errors encountered by application developers is an exhaustion of object handles.

There are two types of object handles defined in the Java operating system for BlackBerry today:

- Transitive
- Persistent

Transitive object handles, more commonly referred to simply as "object handles", are consumed for each object in the system.

Persistent object handles, on the other hand, are consumed only when an object is persisted. That is, a persisted object consumes both a transitive and persistent object handle.

There are a fixed number of object handles in the system where the number of persistent object handles is roughly half the number of transitive object handles. For developers, it is important to conserve object handles as much as possible especially when persisting objects to the limited number. Depending on the data structure selection it does not take too many stored records to exhaust the number of object handles.

Consider a record that contains 10 String fields representing such items as name, phone number, address, etc. This record will consume 11 persistent object handles - one for the record object and one for each string. If 3000 records are persisted, 33,000 persistent object handles will be consumed, which exceeds the current number of persistent object handles on a 16MB device.

Flash Memory	Persistent Object Handles	Object Handles
8 MB	12,000	24,000
16 MB	27,000	56,000
32 MB	65,000	132,000

An API has been exposed in the BlackBerry JDE v4.0 to provide the capability of Object Grouping. Using the net.rim.device.api.system.ObjectGroup class, a developer can consolidate the object handles for an object into one group. Using the example above, if you group the record you will only use one object handle for the record instead of 11. The object handles for the String fields will be consolidated under the record object handle.

However, there are two downsides to the use of the ObjectGroup API.

1. When an object is grouped it is considered read-only. This does not mean that the object cannot be changed, but rather that you need to ungroup the object before making any changes. Once the changes are complete, regroup the object using the normal group method. If you attempt to modify a grouped object without ungrouping it first, an ObjectGroupReadOnlyException will be thrown.
2. There is a performance penalty applied when an object is ungrouped. The system will create a copy of the grouped object and allocate handles to each of the objects inside that group. Therefore, ungrouping of objects should only be done when necessary. For the curious at heart, this explains why many applications on the BlackBerry have a separate View and Edit menu item. The View menu item does not ungroup the entry while the Edit menu item does ungroup the item.

The sample code associated with this article showcases how an application can take advantage of object grouping for their data storage on the device with the common address book implementation. The sample code also shows how easy it can be to attempt to modify a grouped object which the system will not allow.

Class: net.rim.device.api.system.ObjectGroup

An ObjectGroup represents a collection of objects contained within the same filesystem record. A reference to any member of the group from outside of the group will prevent the entire group from being garbage collected. ObjectGroup is a signed class.

ObjectGroup.createGroup()

Description:

- Recursively groups an object, and everything it references to, into one orphan filesystem record. References between objects in a group are rewritten as relative references which do not consume an object handle. This version will record in the event log if a grouping failed.

Syntax:

- void createGroup(Object obj)

Parameters:

- The object to group

Returns:

- Nothing

Throws:

- *ObjectGroupTooBigException* if there are too many object to fit in 64k
- *ObjectGroupReadOnlyException* if any attempt is made to modify an object in a group.

ObjectGroup.createGroupIgnoreTooBig()

Description:

- Recursively groups an object, and everything it references to, into one orphan filesystem record. References between objects in a group are rewritten as relative references which do not consume an object handle. This version will record in the event log if a grouping failed.

Syntax:

- void createGroupIgnoreTooBig(Object obj)

Parameters:

- The object to group

Returns:

- Nothing

Throws:

- *ObjectGroupReadOnlyException* if any attempt is made to modify an object in a group.

ObjectGroup.expandGroup()

Description:

- Return a new object which is a read-write clone of an object group

Syntax:

- Object expandGroup(Object obj)

Parameters:

- The object to ungroup

Returns:

- A read-write clone of an object group

Throws:

- Nothing

ObjectGroup.isInGroup()

Description:

- Test if object is grouped

Syntax:

- boolean isInGroup(Object obj)

Parameters:

- The object of the group

Returns:

- True is object is grouped else returns False

Throws:

- Nothing

Sample Source Code

```
/*
 * ObjectGroupingDemoApplication.java
 *
 * © Research In Motion Limited, 2003-2005
 */

package com.rim.samples.device.objectgroupingdemo;

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

/**
 * The application class contains the starting point
 * for this demo and shows how grouping and ungrouping
 * should work.  As well, it shows how an exception is
 * thrown if the object is grouped and the application
 * attempts to modify it.
 */
public class ObjectGroupingDemoApplication extends Application
{
  public static void main( String[] args )
  {
    AddressBook addressBook = AddressBook.getInstance();
```

```
    // Create some sample address book entries.
    AddressBookRecord record1 = new AddressBookRecord( "Mr.", "Tony", "Hawk" );
    AddressBookRecord record2 = new AddressBookRecord();
    record2.setTitle( "Mrs." );
    record2.setFirstName( "Greta" );
    record2.setLastName( "Shaw" );

    addressBook.add( record1 );
    addressBook.add( record2 );

    addressBook.remove( record2 );
    addressBook.update( record1, record2 );

    // Ensure that the size of the address book is correct.
    if( addressBook.size() != 1 ) {
      throw new RuntimeException();   // We should only have one record left.
    }

    // Now, try to update a record entry from the address book directly without ungrouping.
    AddressBookRecord myRecord = addressBook.getRecord( 0 );

    // Show that myRecord is currently in a group.
    boolean isGrouped = ObjectGroup.isInGroup( myRecord );
    System.out.println( "Object is grouped? " + isGrouped );

    // Try to update the record and catch the resulting exception.
    // Note that this is a runtime exception so we need to explicitly catch it.
    try {
      myRecord.setFirstName( "Elizabeth" );
    } catch( ObjectGroupReadOnlyException e ) {
      System.out.println( "ObjectGroupReadOnlyException caught as expected: " + e );
    }

    // Clear up the address book to remove remaining entries.
    addressBook.remove( myRecord );
  }
}
```

```
/*
 * AddressBookRecord.java
 *
 * © Research In Motion Limited, 2003-2005
 */

package com.rim.samples.device.objectgroupingdemo;

import net.rim.device.api.system.*;
import net.rim.device.api.util.*;

/**
 * This class represents the record object for the address book
 * record sample code.  It is a persistent object and is intended
 * to showcase how grouping would be applied to this object.
 */
public class AddressBookRecord implements Persistable
{
  private String _title;          // The title of the address book record.
  private String _firstName;      // The first name of the address book record.
  private String _lastName;       // The last name of the address book record.

  /**
   * Create an empty version of the AddressBookRecord.
   */
```

```
  public AddressBookRecord()
  {
    _title = "";
    _firstName = "";
    _lastName = "";
  }

  /**
   * Create a new AddressBookRecord with the specified title, first name and last name.
   * @param title the title of the AddressBookRecord.  It cannot be null.
   * @param firstName the first name of the AddressBookRecord.  It cannot be null.
   * @param lastName the last name of the AddressBookRecord.  It cannot be null.
   * @throws IllegalArgumentException if any of the parameters are null.
   */
  public AddressBookRecord( String title, String firstName, String lastName )
  {
    if( title == null || firstName == null || lastName == null ) {
      throw new IllegalArgumentException();
    }
    _title = title;
    _firstName = firstName;
    _lastName = lastName;
  }

/**
 * Returns the title of the AddressBookRecord.
 * @return the titel of the AddressBookRecord.
 */
  public String getTitle()
  {
    return _title;
  }

  /**
   * Returns the first name of the AddressBookRecord.
   * @return the first name of the AddressBookRecord.
   */
  public String getFirstName()
  {
    return _firstName;
  }

  /**
   * Returns the last name of the AddressBookRecord.
   * @return returns the last name of the AddressBookRecord.
   */
  public String getLastName()
  {
    return _lastName;
  }

  /**
   * Sets the title of the AddressBookRecord.
   * @param title the new title of the AddressBookRecord.
   */
  public void setTitle( String title )
  {
    _title = title;
  }

  /**
   * Sets the first name of the AddressBookRecord.
   * @param firstName the new first name of the AddressBookRecord.
   */
  public void setFirstName( String firstName )
```

```
  {
    _firstName = firstName;
  }

  /**
   * Sets the last name of the AddressBookRecord.
   * @param lastName the new last name of the AddressBookRecord.
   */
  public void setLastName( String lastName )
  {
    _lastName = lastName;
  }

  // Javadocs copied from super class.
  public boolean equals( Object obj )
  {
    if( obj == this ) return true;
    if( obj instanceof AddressBookRecord ) {
      AddressBookRecord other = (AddressBookRecord)obj;
      if( !other._title.equals( _title )) return false;
      if( !other._firstName.equals( _firstName )) return false;
      if( !other._lastName.equals( _lastName )) return false;
      return true;
    }
    return false;
  }
}
```

```
/*
 * AddressBook.java
 *
 * © Research In Motion Limited, 2003-2005
 */

package com.rim.samples.device.objectgroupingdemo;

import net.rim.device.api.system.*;
import net.rim.device.api.util.*;

/**
 * This class represents the AddressBook implementation
 * where one can add, remove, update and traverse the
 * different address book records stored in the address
 * book.
 */
public class AddressBook
{
  // com.rim.samples.device.objectgroupingdemo.AddressBook.PERSIST
  private static final long PERSIST = 0xcf76f65979a526eaL;
  // com.rim.samples.device.objectgroupingdemo.AddressBook.ADDRESS_BOOK
  private static final long ADDRESS_BOOK = 0xdc33b15c18be898fL;

  // Reference to the PersistentObject for our address book.
  private PersistentObject _persist;
  // The array of address book records that make up the address book.
  private AddressBookRecord[] _records;

  /**
   * Simple constructor for the class that will initialize the
   * _records array using the data stored in the persistent object.
   * This method is marked private because no other class should be
   * able to instantiate the AddressBook.
   */
  private AddressBook()
```

```
{
  _persist = PersistentStore.getPersistentObject( PERSIST );
  _records = (AddressBookRecord[])_persist.getContents();
  if( _records == null ) {
    _records = new AddressBookRecord[0];
    _persist.setContents( _records );
  }
}

/**
 * Returns the singleton instance of the AddressBook.
 * @return the singleton instance of the AddressBook.
 */
public static AddressBook getInstance()
{
  RuntimeStore rs = RuntimeStore.getRuntimeStore();
  AddressBook addressBook = (AddressBook)rs.get( ADDRESS_BOOK );
  if( addressBook == null ) {
    addressBook = new AddressBook();
    rs.put( ADDRESS_BOOK, addressBook );
  }
  return addressBook;
}

/**
 * Adds the address book record.
 * @param record the address book record to add.
 * Note: This method does not perform any duplicate detection.
 */
public void add( AddressBookRecord record )
{
  if( record == null ) {
    throw new IllegalArgumentException();
  }

  // Be sure to group the record before adding it to the address book.
  ObjectGroup.createGroup( record );
  Arrays.add( _records, record );
  _persist.commit();
}

/**
 * Updates the oldRecord in the address book with the contents specified
 * in the newRecord.
 * @param oldRecord the record to update in the address book.
 * @param newRecord the record to use for the data to update the oldRecord.
 */
public void update( AddressBookRecord oldRecord, AddressBookRecord newRecord )
{
  if( oldRecord == null || newRecord == null ) {
    throw new IllegalArgumentException();
  }

  // The obvious implementation for updating an address book record is
  // to remove the oldRecord and add the new record.  The following two
  // lines would accomplish this task.
  //
  // Arrays.remove( _records, oldRecord );
  // Arrays.add( _records, newRecord );
  //
  // However, this is both inefficient (traverse the array, resize the array twice, etc)
  // and doesn't help demonstrate grouping and ungrouping.  So, in this implementation
  // we will simply modify the title, first name and last name in the specific
  // old record with the data from the new record.
```

```
    // Ensure that the oldRecord is actually contained in our address book.
    int index = Arrays.getIndex( _records, oldRecord );
    if( index == -1 ) {
      // Item not found.
      throw new IllegalArgumentException();
    }

    // Ungroup the old record.
    AddressBookRecord ungroupedRecord = (AddressBookRecord)ObjectGroup.expandGroup( _records[index] );

    ungroupedRecord.setTitle( newRecord.getTitle() );
    ungroupedRecord.setFirstName( newRecord.getFirstName() );
    ungroupedRecord.setLastName( newRecord.getLastName() );

    ObjectGroup.createGroup( ungroupedRecord );
    _records[index] = ungroupedRecord;
    _persist.commit();
  }

  /**
   * Removes the address book reecord from the address book.
   * @param record the record to remove.
   */
  public void remove( AddressBookRecord record )
  {
    if( record == null ) {
      throw new IllegalArgumentException();
    }
    Arrays.remove( _records, record );
    _persist.commit();
  }

  /**
   * Returns the address book record specified by the index.
   * @param index the index into the list of address book
   * records.
   * @return the address book record specified by the index if
   * that corresponds to a valid index.
   * @throws IllegalArgumentException if the index is invalid.
   */
  public AddressBookRecord getRecord( int index )
  {
    if( index < 0 || index >= _records.length ) {
      throw new IllegalArgumentException();
    }
    return _records[ index ];
  }

  /**
   * Returns the size of the address book.
   * @return the number of records currently stored
   * in the address book.
   */
  public int size()
  {
    return _records.length;
  }
}
```

The Object versus Service dilemma: Making room for OO in Web Services

Marco Adragna, Research In Motion

Can we look at a Web Service as if it is an Object?

I asked a similar question in a W3C forum. The 30+ replies I received from W3C members gave me the measure of how hot the Object versus Services topic is. My question pointed at Grid Services as an example of object-oriented concepts used in a web service environment. Is such a practice wrong?

A Grid Service is "*a Web Service that conforms to a set of conventions (interface and behaviour)*" [1]. Grid Service experts have been working with W3C toward the inclusion of interface (exportType) inheritance in Web Service Description Language (WSDL). This feature was present in the WSDL 1.2 working draft of the 11 June 2003 release and has been added to WSDL 2.0. The focus has subsequently shifted to the possibility of including Service Data Elements in WSDL. Service Data Elements can be compared to attributes of an object-oriented interface, described in an Interface Definition Language. Such data describes externally observable behaviour, such as the terminationTime of a Grid Service instance. The notions of stateful resources and limited lifetime mark the difference between Grid Services and "traditional" Web Services.

The need for a limited lifetime is only natural when we make a comparision with objects. Resources of any provider are bounded. We can't create an infinite number of instances with an eternal lifetime, so we either provide a method to destroy an object or we give it a limited lifetime.

Over the last few decades, a great wealth of knowledge has been accumulated on object-oriented design. This knowledge is partly condensed in the form of Design Patterns and is widely available for businesses both as mature software development environments and as skilled OO software developers. It could be argued that Web Service aims at being more solid than traditional distributed object architectures. On the other hand, with decades of object orientation behind us, it is only natural to ask how a relatively new technology such as Web Services relates to OO. The rise of Grid Services have made such questions more urgent, but the Object-Service dilemma existed even before them.

The Wizard promise: Click here to publish your class as a Web Service.

If we look at commonly available software development environments, we see that is possible to create a Web Service by specifying which class or which methods are to be available as a "XML Web Service". In a typical Enterprise JavaBeans (EJB) 2.1 implementation, a stateless session bean can be published as a Web Service through its container. Using the Java Web Server (JWS) facility of Apache Simple Object Access Protocol (SOAP) processor Axis, all public methods of a Java class can be published as web services. In Microsoft .Net environment, the attribute WebMethod produces similar results. Given a development environment and a class to publish, a WSDL document can be automatically produced and the service can be requested by anyone who knows and has the privileges to access the related Uniform Resource Identifiers (URI). Cross-platform distributed object architectures have became a reality. Or not?

The Harsh Truth: Interoperable Web Services neither talk about objects nor can think about them.

Following an object-oriented approach, we want to provide requesters of the Diary Web Service with a getCurrentDay() method. This method is intended to return a remote object-reference to an object of the class Day. This object represents the current day in my Diary and encapsulates all information about my appointments. Using the remote reference, the requester can invoke methods of this object. It all seems to work, but if the requester is on a different platform (e.g. provider on .NET and requester on Tomcat/Axis) it is likely to break down. Web services do not offer a standard mechanism for handling remote object-reference. A custom solution could be found, but this would create the very sort of tight coupling that Web Services are meant to avoid.

We have seen that passing an object by reference is not a good idea. Maybe we can pass it by value? Microsoft .Net Framework ver.1 allows Web Services to pass objects by value. Objects are serialized in XML using the XMLSerializer class. In reality, only the public state of objects gets serialized. This XML serialization does not convert methods or private properties. There is no guarantee that an object serialized on the server will get deserialized into an object of the same type on the client. In general, SOAP can transport XML-serialized objects, but there's no guarantee that the receiver will be able to deserialize them correctly.

Someone might say, "Who needs those objects anyway, we'll do it all with Web Services", "Web services are designed to be coarser grained than objects" and "Web services are layered on top of an object oriented system". Yes, but can they express the Classifier/Instance relationship that exist between a class and its objects? Can Web Services "think" about objects?

What objects are really about.

"Software engineering is the application of scientific principles to (1) the orderly transformation of a problem into a working software solution and (2) the subsequent maintenance of that software until the end of its useful life'' [2]. A Software Engineer is called when there is a generically problematic situation that might be solved by computing means. The first step is to analyse the problem, gaining knowledge of the problem and its domain. The second step is to describe the external behaviour of the software/hardware system that might solve the problem. Successive iterations bring further refinement. Partitioning and Abstraction are two of the very few principles used in the process of structuring Problem Analysis and Behavioural Requirements. If the process of turning a problem analysis into a working software solution is to be simple, then the software technology used to implement the solution needs to fully support the primitives of problem analysis. Partitioning express the 'aggregation/part' structural relationship, whilst abstraction captures the concept of 'general/specific' or 'example of' or 'instance of'. Where are such concepts in Web Services? Composition languages (e.g. WS-BPEL) allow the creation of a new Web Service from existing ones. This process defines an aggregation/part relationship between the new composition and single web services that compose it. The next step could be fully supporting Abstraction and the concept of "Instance Of".

Conclusion

Can we look at a Web Service as if it was an Object? In the last two years, the standardization effort of international organisations such as W3C, OASIS and Globus has brought those two worlds much closer. Specifications such as WS-Notification and features like WSDL interface inheritance bring OO concepts to the Web Service World. This trend is likely to continue. In the meantime, cross-platform interoperability is still a challenge and the object-service dilemma remains to be solved.

References

1. S.Tuecke (ANL), K. Czajkowski (USC/ISI), I Foster (ANL), J. Frey (IBM), S. Graham (IBM), C. Kesselman (USC/ISI), T. Maquire (IBM), T. Sandholm (ANL), D. Snelling (Fujitsu Labs), P. Vanderbilt (NASA) – 27th June 2003 - Open Grid Services Infrastructure (OGSI) - Version 1.0

   During 2004 and 2005 the WS-Resource Framework has refactored interfaces and concepts of the OGSI V1.0 specification in a manner that exploits recent developments in Web services architecture.

2. A. M. Davis – 1993
   Software Requirements, Objects, Functions and States
   Prentice Hall

3. W3C, GLOBUS and OASIS web service related specifications

Would you like some testing with that?

Emad Shihab, Research In Motion

Although developers and testers work closely together, many software developers do not have a clear understanding of what testers do. This article will detail some of the common issues facing developers and testers in the industry.

Software testing

Software testing has become a crucial part of the software development cycle, and a lot of the important groundwork for testing is provided by the development teams. The quality of the testing performed depends on the information communicated to the testers by the developers regarding the functionality of the software.

The majority of test suites generated by testers are based on feature specification documents provided to them to software development teams provide. The creation of test suites can be easy if all the feature information is thorough and accurate. Of course it is the tester's job to run these test suites, but as a tester, I enjoy running a complete and accurate test suite far more than an incomplete or inaccurate test suite.

How are test suites created?

The all-important first step is to create feature specification documents for the different features. Without those, you might as well go to the beach.

A tester then examines the specifications and creates the necessary test suites based on the type of the feature. The feature can be broken down into different categories (GUI, low level functionality, etc.)

The generated test suites are then verified by a group of two or three other testers. After implementing any changes to the test suite suggested by the verification group, the test suite is finalized and ready to be used.

Why are so many logged bugs not true errors?

The main cause of this problem is that software changes rapidly and many testers do not have the time to update their test suites. This causes inconsistencies when new features are tested and causes 'false' bugs to be logged.

In addition, testing environments play a big role, especially when testing wireless applications, which is the case for most BlackBerry testers. Radio frequency noise and carrier outages are examples of "real world" factors that can affect wireless applications. Testers must take these factors into account when testing wireless applications.

This bug does not seem to be fixed although the developer says it is?

As a tester I cannot speak for developers, but I can convey my side of the story. The main reason that I send back bugs as 'not fixed' is because the developers have verified the issue in a way that is inaccessible or unknown by a tester. This means that when the developers verified issue 'X', they looked at some logs that an average tester cannot read or understand. How well testers verify bugs is very much dependent on the developers. A brief description of how the developer tested the fix would help the tester greatly. Even a brief description of the steps undertaken by the developer would assure the tester that the rights steps are being followed. This would result in saving valuable time for both parties, rather than sending issues back and forth.

I cannot stress how important the developer's role is when it comes to testing software. Having effective communication between the software testing and development teams can sometimes make the difference between a reliable and on-time product or a product that is late and suffers from quality issues.

API Spotlight: BlackBerry does GPS

Richard Evers, Editor

My family took a trip to California last year. During our return journey from Napa Valley, we lost our way and ended up in a cemetery in a very lonely section of California.

I powered up my personal GPS, hung it out of the window for an eternity to sync with the satellites, got a street level map of the region, and used it to find the highway heading back to San Francisco.

Without question, my GPS is one of the best investments I've ever made. With GPS in hand, the fear of getting lost while traveling is largely a thing of the past.

The GPS-enabled BlackBerry 7520 Wireless Handheld™ implementation includes traditional Autonomous mode (direct sync with satellites), Assisted mode, and Cellsite mode. Each mode has strengths and weaknesses that will be discussed in this article.

How it Works

Autonomous GPS works by syncing with four or more satellites to determine latitude, longitude and altitude.

Latitude is the angular distance north or south of the equator measured by lines circling the earth parallel to the equator measured from 0 degrees to 90 degrees.

Longitude is the angular distance east or west of the prime meridian (Greenwich Meridian) as measured by lines perpendicular to the parallels and converging at the poles 0 degrees to 180 degrees. Greenwich Meridian runs through "the primary transit" instrument (the main telescope) at the Royal Observatory in Greenwich, England.

Altitude is the current elevation above sea level.

GPS accuracy can be improved through use of the Wide Area Augmentation System (WAAS). WAAS is a Satellite-Based Augmentation System (SBAS) that calculates the errors in the GPS signal at several monitoring stations, and then transmits error correction messages from geostationary satellites to GPS receivers.

Statement from the FAA

WAAS is based on a network of approximately 25 ground reference stations that covers a very large service area. Signals from GPS satellites are received by wide area ground reference stations (WRSs). Each of these precisely surveyed reference stations receive GPS signals and determine if any errors exist. These WRSs are linked to form the U.S. WAAS network. Each WRS in the network relays the data to the wide area master station (WMS) where correction information is computed. The WMS calculates correction algorithms and assesses the integrity of the system. A correction message is prepared and uplinked to a geosynchronous satellite via a ground uplink system (GUS). The message is then broadcast from the satellite on the same frequency as GPS (L1, 1575.42MHz) to receivers on board aircraft (or hand-held receivers) which are within the broadcast coverage area of the WAAS. These communications satellites also act as additional navigation satellites for the aircraft, thus, providing additional navigation signals for position determination.

The WAAS will improve basic GPS accuracy to approximately 7 meters vertically and horizontally, improve system availability through the use of geostationary communication satellites (GEOs) carrying navigation payloads, and to provide important integrity information about the entire GPS constellation.

In practice, WAAS correction can improve accuracy to less than 7 meters in most commercial GPS units. GPS accuracy is better at night when the ionospheric errors have decreased. Accuracy of 1.8 meters has been documented in WAAS-capable devices at night.

The easiest way to understand how GPS works is to initially determine positioning from a two-dimensional perspective.

For example, four amazing birds (Raven, Robin, Jay and Martin) are gathered together in a park. Raven is lost and asks the other birds for help. Robin tells Raven that he's 15 km away from Lancaster, PA (as the bird flies). Jay tells Raven that he's 7 km away from Leola, PA, and Martin tells Raven that he's 8 km away from Strasburg, PA.

Raven whips out his trusty map and geometry compass, and (carefully using his wings as if they were hands) draws a circle around Lancaster with a scaled down radius of 15 km, draws a circle around Leola with a scaled down radius of 7 km, and draws a final circle around Strasburg with a scaled down radius of 8 km. The intersection of the three circles shows that Raven is perched in a park in Bird in Hand, PA.

To bring this into the realm of GPS, you would need three reference points represented by spheres where each point would be the distance from the caller to the satellite. Earth would act as the final sphere for this example. The final earthbound intersection point of the spheres would provide the latitude, longitude and altitude. Additionally, directional movement and speed could be mathematically determined after each refresh cycle.

Two things must be known before calculating GPS values:

1. The location of each of the satellites
2. The distance between the GPS unit and each of four or more satellites

The GPS clock, which is separate from the BlackBerry handheld clock, must also be regularly synced to the atomic clocks on the satellites before trying to determine distance. Constant syncing with atomic clocks is a standard feature of all GPS units.

There are 24 active GPS satellites in high orbit at all times in predictable locations with 3 backup satellites. Satellites are solar powered, 17 feet across when the solar panels are extended, weigh roughly 2,000 pounds, transmit up to about 50 watts, and are built to remain in use for 10 years. New satellites are constantly being built and launched by the U.S. Department of Defense to ensure that the network remains viable at all times.

Satellite location is initially determined by receiving the satellite identifier and then performing a lookup in a local GPS positioning almanac to determine where the satellite should be orbiting at that moment in time. The U.S. Department of Defense closely monitors the position of each satellite and reprograms the satellites to transmit location adjustments to all GPS units on a regular basis, and also repositions the satellites from time to time to return to normal orbit.

The distance between the GPS unit and a satellite is determined through use of a pseudo-random pattern. This pattern is generated and transmitted by the satellite at the same time as the GPS unit generates the same pattern. The satellite's pattern will start to be received later than the initial time of generation, which means that the GPS unit can determine how far away the satellite is by multiplying the length of delay by the speed of light. This is why it is critical to sync to the atomic clock, and why some calculation errors still occur because the GPS clock will never be in perfect sync with the atomic clocks. Other factors come into play to further degrade the calculation such as signal obstructions on earth and in the atmosphere.

With location and distance known, a calculation can be made to determine where the four spheres intersect. It's possible for three spheres to intersect even if the underlying values are wrong, but four spheres cannot intersect unless the calculations are accurate. Longitude and latitude can be determined with three satellites, but the fourth satellite is needed to determine altitude and reduce errors. Distance calculations are made in real time using the clock settings of the GPS unit, therefore all calculations will be proportionately incorrect and close enough to intersect the four spheres.

Once the GPS unit has fully synced with the satellites to determine position, it's easy to determine direction, actual and average speed, trip duration and length, estimated time of arrival, and traveled path by comparing and accumulating differences between sampling periods.

The greatest strengths of autonomous GPS is that it is accurate, free to use, and performs rapid updates after being fully synced with four satellites.

The greatest weaknesses are:

- It can take several minutes to fully sync with four or more satellites the first time it is powered up in a new area
- Power consumption can be fairly heavy for a wireless device depending on the sampling ratio. It's often recommended to use a car charger when using a GPS unit for driving directions.
- Connectivity is often hard to maintain because satellite signal strength of 50 watts from a distance of 12 miles is so weak that GPS units tend to lose connection when indoors, and when traveling through or around physical obstructions. The radio signal (1575.42 MHz in the UHF band for civilian use) travels by line of sight, which means that it will pass through clouds, glass and plastic but will not pass through most solid objects such as buildings, overpasses, earth, automobile bodies and mountains. It even has a hard time passing through water.
- Accuracy is hard to maintain because signal delay errors can be introduced in the atmosphere through reflection off tall buildings and rocks and through use of the internal GPS clock. Additionally, orbital errors can be encountered where the reported satellite location is inaccurate.
- Buildings, terrain, electronic interference and more can restrict satellite "visibility".
- Poor placement of satellites (e.g. located in the line of sight or in a tight grouping) can degrade accuracy.

Supported Modes

The GPS implementation for BlackBerry includes Autonomous mode, Assisted mode, and Cellsite mode. If you've managed to read this far, then you are pretty knowledgeable about Autonomous mode. We'll cover the other modes here.

Cellsite Mode

Provides the GPS coordinate of the serving cellsite node. This mode provides the least accurate location information in the fastest possible time. It is useful when rapid emergency assistance is required and broad strokes positioning will suffice. It uses cell tower location to calculate approximate positioning. Accuracy is very poor in rural areas where cell tower density is sparse. Conversely, dense urban areas with dense concentrations of cell towers will improve accuracy, often times to within a few city blocks.

Assisted Mode

Uses the network in an assisted mode. It is more accurate than cellsite mode but takes more time to obtain location information due to wireless latency and server response times.

It works by providing some information to a remote server for processing. The device makes a request for "assist data" from the server on the carrier network which has location information. The server will respond with "assist data" (which includes ephemeris, ionospheric modeling, etc). The greatest benefit of assisted GPS (aGPS) over autonomous GPS is the response time. Initial aGPS responses can return in under 10 seconds depending on network latency. Autonomous GPS takes longer to determine the initial position, but is fast for all subsequent positions.

GPS for BlackBerry

The Chip

The current implementation of the BlackBerry 7520™ uses a SiRFstarIIe/LP chipset to provide GPS services. Note that the chip used to provide GPS services may change over time.

This chip set supports Satellite Based Augmentation System (SBAS), which encompasses WAAS and the European Geostationary Navigation Overlay Service (EGNOS). It also provides support for Coast Guard Maritime Differential GPS (DGPS).

The chip cold starts in 45 seconds, uses less than 175 mW at full power, and will automatically support a mode to reduce power to under 60 mW and under 20 mA.

The manufacturer states that positional accuracy is less than 10 meters for autonomous GPS, less than 5 meters when WAAS is enabled, and less than 2.5 meters when enabled for Beacon DGPS. The chip is accurate beneath 60,000 feet, and under 1,000 knots (1,152 mph or 1,850 kph).

Source: **http://www.sirf.com/products-ss2eLP.html**

GPS coding in brief

```
try {
  // Create a Criteria object for
  // defining selection criteria
  Criteria cr = new Criteria();

  //
  // setAddressInfoRequired()
  // true: if location provider should be able
  // to determine textual address information
  // there is no guarantee that AddressInfo
  // is available in any mode
  // Default = false
  //
  // Note: in the current implementation,
  // setAddressInfoRequired() has no impact
  // on results
  //
// cr.setAddressInfoRequired(false);

  //
  // setAltitudeRequired()
  // true: if location provider should be able
  // to determine device altitude
  // AUTONOMOUS GPS is only mode that could
  // provide this information
  // Default = false
  //
  // Note: in the current implementation,
  // setAltitudeRequired() has no impact
  // on results
  //
//  cr.setAltitudeRequired(false);

  //
  // setPreferredResponseTime()
  // value: response time or timeout if the
  // response isn't provided in time
  // default: NO_REQUIREMENT
  //
  // Note: in the current implementation,
  // setPreferredResponseTime() has no impact
  // on results
  //
// cr.setPreferredResponseTime(NO_REQUIREMENT);

  //
  // setSpeedAndCourseRequired()
  // true: location provider should be able to
  // determine speed and course
  // Speed and course should be available in
  // all three modes
  // Default = false
  //
  // Note: in the current implementation,
  // setSpeedAndCourseRequired() has no
  // impact on results
  //
// cr.setSpeedAndCourseRequired(false);


  //
  // setCostAllowed()
  // Default = true
  // true: if there can be a cost associated
  // with determining location information
  // ASSISTED GPS is only mode that could
  // incur a direct cost
  //
  cr.setCostAllowed(true);

  //
  // setHorizontalAccuracy()
  // Default = NO_REQUIREMENT
  // value: the desired horizonal accuracy
  // in meters
  // Use NO_REQUIREMENT or a larger value (500
  // meters) for assisted and cellsite
  // set a lower value (such as 10 meters)
  // for autonomous
  //
  cr.setHorizontalAccuracy(NO_REQUIREMENT);

  //
  // setPreferredPowerConsumption()
```

```
  // Default = NO_REQUIREMENT
  // POWER_LEVEL_LOW autonomous, cellsite
  // POWER_LEVEL_MEDIUM autonomous, assisted
  // POWER_LEVEL_HIGH autonomous, assisted
  // NO_REQUIREMENT autonomous, assisted
  //
  cr.setPreferredPowerConsumption
    (NO_REQUIREMENT);

  //
  // setVerticalAccuracy()
  // Default = NO_REQUIREMENT
  // value: the desired vertical accuracy
  // in meters
  // Use NO_REQUIREMENT or a larger value
  // (500 meters) for assisted and cellsite
  // set a lower value (such as 10 meters)
  // for autonomous
  //
  cr.setVerticalAccuracy(NO_REQUIREMENT);

  //
  // set up a location provider instance
  //
  LocationProvider lp =
    LocationProvider.getInstance(cr);

  //
  // Get location, one minute timeout
  // leave up to 180 seconds for autonomous mode
  //
  Location l = lp.getLocation(60);

  Coordinates c = l.getQualifiedCoordinates();

  if ( c != null ) {
    // use coordinate information
    double latitude  = c.getLatitude();
    double longitude = c.getLongitude();
    float  altitude  = c.getAltitude();
  }

  } catch (LocationException e ) {
    // Not able to retrieve location information
    ...
  }
}
```

The API

BlackBerry supports JSR-179: Location API for Java ME, and uses the javax.microedition.location package which consists of the following functionality:

Interface: LocationListener

A listener that receives events associated with a particular LocationProvider. Applications implement this interface and register it with a LocationProvider to obtain regular position updates. If location updates cannot be provided at the defined interval, then an update can be sent to the listener that contains an 'invalid' Location instance. Applications are responsible for any possible synchronization needed in the listener methods. The listener methods must return quickly and should not perform any extensive processing. The method calls are intended as triggers to the application. An application should do any necessary extensive processing in a separate thread and only use these methods to initiate processing.

Method LocationListener.locationUpdated

Syntax:

```
void locationUpdated
(
  LocationProvider provider,
  Location location
)
```

Description:

Called by the LocationProvider to which this listener has been registered. This method will be called periodically according to the interval defined when registering the listener to provide updates of the current location. The parameters are the source of the event (provider), and the new position (location).

Method: LocationListener.providerStateChanged

Syntax:

```
void providerStateChanged
(
  LocationProvider provider,
  int newState
)
```

Description:

Called by the LocationProvider to which this listener has been registered when the state of the LocationProvider changes.

Provided state changes are delivered to the application as soon as possible after the state of a provider changes. The timing of these events are not related to the period of the location updates.

If the application is subscribed to receive periodic location updates, it will continue to receive these regardless of the state of the LocationProvider. If the application wishes to stop receiving location updates for an unavailable provider, it should de-register itself from the provider.

The parameters are the source of the event (provider), and the new state of the LocationProvider (newState), which is a constant value defined in the LocationProvider class as shown below:

- AVAILABLE
- OUT_OF_SERVICE
- TEMPORARILY_UNAVAILABLE

Interface: ProximityListener

A listener to events associated with detecting proximity to some registered coordinates. Applications implement this interface and register it with a static method in LocationProvider to obtain notfications when proximity to registered coordinates has been detected.

This listener is called when the device enters the proximity of the registered coordinates. The proximity is defined as the proximity radius around the coordinates combined with the horizontal accuracy of the current sampled location.

The listener is called only once when the device enters the proximity of the registered coordinates. The registration with these coordinates is cancelled when the listener is called. If the application wants to be notified again about these coordinates, it must re-register the coordinates and the listener.

Method: ProximityListener.monitoringStateChanged

Syntax:

```
void monitoringStateChanged
  (boolean isMonitoringActive)
```

Description:

Called to notify that the state of the proximity monitoring has changed. These state changes are delivered to the application as soon as possible after the state of the monitoring changes. The ProximityListener always remains registered until the application explicitly removes it with LocationProvider.removeProximityListener or when the application exits.

The parameter is a boolean (isMonitoringActive) indicating the new state of the proximity monitoring. A value of true indicates that the proximity monitoring is active.

Method: ProximityListener.proximityEvent

Syntax:

```
void proximityEvent
(
  Coordinates coordinates,
  Location location
)
```

Description:

After registering this listener with the LocationProvider, this method will be called the current location of the deviceis within the defined proximity radius of the registered coordinates.

The parameters are the registered coordinates to which proximity has been detected (coordinates), and the current location of the device (location).

Class: AddressInfo

This class holds textual address information about a location. Typically the information is a street address where the information is divided into fields (e.g. street, postal code, city, etc.). Defined field constants can be used to retrieve field data. If the value of a field is not available, it is set to null.

The names of the fields use terms and definitions that are commonly used in the United States. Addresses for other countries should map these to the closest corresponding entities used in that country.

This class is only a container for the information. The getField method returns the value set for the defined field using the setField method.

Fields	Notes
STREET	
EXTENSION	Unit, flat, apartment number
POSTAL_CODE	Postal or zip code
CITY	Village, town or city
COUNTY	
STATE	State or province
COUNTRY	
COUNTRY_CODE	Two-letter ISO 3166-1 code
DISTRICT	Municipal district
BUILDING_NAME	
BUILDING_FLOOR	
BUILDING_ROOM	
BUILDING_ZONE	
CROSSING1	Street in a crossing
CROSSING2	Street in a crossing
URL	
PHONE_NUMBER	

Constructor

```
AddressInfo()
```

Method: AddressInfo.getField

Syntax:

```
java.lang.String getField(int field)
```

Description:

Returns the value of an address field or null if the field is not available.

The parameter is the ID of the field to be retrieved.

This method will throw java.lang.IllegalArgumentException if the parameter field ID is not one of the constant values defined in this class

Method: AddressInfo.setField

Syntax:

```
void setField(int field, java.lang.String value)
```

Description:

Sets the value of an address field.

The parameters are the ID of the field to be set and the new value for the field. null is used to indicate that the field has no content.

This method throws java.lang.IllegalArgumentException if the parameter field ID is not one of the constant values defined in this class

Class: Coordinates

Represents coordinates as latitude-longitude-altitude values. The latitude and longitude values are expressed in degrees using floating point values. The degrees are in decimal values (rather than minutes/seconds). The coordinates are given using the WGS84 datum., which is a set of conventions, constants and formulae. For more information:

http://www.gps.gov.uk/guide4.asp

This class also provides convenience methods for converting between a string coordinate representation and the double representation used in this class.

Field	Description
DD_MM_SS	Identifier for string coordinate representation Degrees, Minutes, Seconds and decimal fractions of a second
DD_MM	Identifier for string coordinate representation Degrees, Minutes, decimal fractions of a minute

Constructor

```
public Coordinates
(
  double latitude,
  double longitude,
  float altitude
)
```

Constructs a new Coordinates object with the values specified. The latitude and longitude parameters are expressed in degrees using floating point values. The degrees are in decimal values (rather than minutes/seconds). The coordinate values always apply to the WGS84 datum. The Float.NaN value can be used for altitude to indicate that altitude is not known.

Parameters:

latitude

- Latitude of the location. Valid range: [-90.0, 90.0].
- Positive values indicate northern latitude and negative values southern latitude.

longitude

- Longitude of the location. Valid range: [-180.0, 180.0).
- Positive values indicate eastern longitude and negative values western longitude.

altitude

- Altitude of the location in meters, defined as height above WGS84 ellipsoid.
- Float.NaN can be used to indicate that altitude is not known.

The constructor throws java.lang.IllegalArgumentException if an input parameter is out of the valid range.

Method: Coordinates.getLatitude

Syntax:

```
public double getLatitude()
```

Description

Returns the latitude component in degrees of this coordinate. Positive values indicate northern latitude and negative values southern latitude. The latitude is given in WGS84 datum.

Method: Coordinates.getLongitude

Syntax:

```
public double getLongitude()
```

Description

Returns the longitude component in degrees of this coordinate. Positive values indicate eastern longitude and negative values western longitude. The longitude is given in WGS84 datum.

Method: Coordinates.getAltitude

Syntax:

```
public float getAltitude()
```

Description

Returns the altitude component in meters of this coordinate. Altitude is defined to mean height above the WGS84 reference ellipsoid. 0.0 means a location at the ellipsoid surface, negative values mean the location is below the ellipsoid surface, Float.NaN that no altitude is not available.

Method: Coordinates.setAltitude

Syntax:

```
public void setAltitude(float altitude)
```

Description:

Sets the geodetic altitude for this point. The parameter passed is the altitude of the location in meters, defined as height above the WGS84 ellipsoid. 0.0 means a location at the ellipsoid surface, negative values mean the location is below the ellipsoid surface, Float.NaN that no altitude is not available

Method: Coordinates.setLatitude

Syntax:

```
public void setLatitude(double latitude)
```

Description:

Sets the geodetic latitude for this point. Latitude is given as a double expressing the latitude in degrees in the WGS84 datum. The parameter passed is the latitude component of this location in degrees. Valid range: [-90.0, 90.0].

This method will throw java.lang.IllegalArgumentException if the latitude is out of the valid range.

Method: Coordinates.setLongitude

Syntax:

```
public void setLongitude(double longitude)
```

Description:

Sets the geodetic longitude for this point. Longitude is given as a double expressing the longitude in degrees in the WGS84 datum.

The parameter passed is the longitude of the location in degrees. Valid range: [-180.0, 180.0)

This method throws java.lang.IllegalArgumentException if longitude is out of the valid range.

Method: Coordinates.convert

Syntax:

```
public static double convert
  (java.lang.String coordinate)
```

Description:

Converts a String representation of a coordinate into float representation as used in this API. There are two string syntaxes supported:

1. Degrees, minutes, seconds and decimal fractions of seconds. This is expressed as a string complying with the following Bibliotheque national de France (BNF) definition where the degrees are within the range [-179, 179] and the minutes and seconds are within the range [0, 59], or the degrees is -180 and the minutes, seconds and decimal fractions are 0:

```
coordinate =
  degrees ":" minutes ":" seconds "." decimalfrac
  | degrees ":" minutes ":" seconds
  | degrees ":" minutes
degrees = degreedigits | "-" degreedigits
degreedigits =
  digit | nonzerodigit digit | "1" digit digit
minutes = minsecfirstdigit digit
seconds = minsecfirstdigit digit
decimalfrac = 1*3digit
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
nonzerodigit = "1"|"2"|"3"|"4|"5"|"6"|"7"|"8"|"9"
minsecfirstdigit = "0"|"1"|"2"|"3"|"4"|"5"
```

2. Degrees, minutes and decimal fractions of minutes. This is expressed as a string complying with the following BNF definition where the degrees are within the range [-179, 179] and the minutes are within the range [0, 59], or the degrees is -180 and the minutes and decimal fractions are 0:

```
coordinate = degrees ":" minutes "." decimalfrac
  | degrees ":" minutes
degrees = degreedigits | "-" degreedigits
degreedigits =
  digit | nonzerodigit digit | "1" digit digit
minutes = minsecfirstdigit digit
decimalfrac = 1*5digit
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
nonzerodigit = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
minsecfirstdigit = "0"|"1"|"2"|"3"|"4"|"5"
```

For example, for the double value of the coordinate 61.51d, the corresponding syntax #1 string is "61:30:36" and the corresponding syntax #2 string is "61:30.6"

The single parameter passed is a String in either of the two representation specified above.

The method returns a double value with decimal degrees that matches the String representation given as the parameter.

This method will throw:

java.lang.IllegalArgumentException

- The coordinate input parameter does not comply with the defined syntax for the indicated type or if the inputType is not one of the two constant values defined in this class

java.lang.NullPointerException

- The coordinate string is null

Method: Coordinates.convert

Syntax:

```
public static java.lang.String convert
(
  double coordinate,
  int outputType
)
```

Description

Converts a double representation of a coordinate with decimal degrees into a string representation.

Parameters:

coordinate

- A double representation of a coordinate

outputType

- Type ID of the string representation wanted for output The constant DD_MM_SS identifies the syntax #1 and the constant DD_MM identifies the syntax #2.

This method returns a String representation of the coordinate in a format based on the parameter.

The method will throw java.lang.IllegalArgumentException if the outputType is not one of the two constant values defined in this class or if the coordinate value is not within the range [-180.0, 180.0) or is Double.Na.

Method: Coordinates.distance

Syntax:

```
public float distance(Coordinates to)
```

Description:

Calculates the geodetic distance in meters between the two points according to the ellipsoid model of WGS84. Altitude is left out of the calculations.

This method will throw java.lang.NullPointerException if the parameter is null.

Method: Coordinates.azimuthTo

Syntax:

```
public float azimuthTo(Coordinates to)
```

Description:

Calculates the azimuth between the two points according to the ellipsoid model of WGS84. The azimuth is relative to true north. The Coordinates object on which this method is called is considered the origin for the calculation and the Coordinates object passed as a parameter is the destination which the azimuth is calculated to. When the origin is the North pole and the destination is not the North pole, this method returns 180.0. When the origin is the South pole and the destination is not the South pole, this method returns 0.0. If the origin is equal to the destination, this method returns 0.0. The implementation shall calculate the result as exactly as it can. However, it is required that the result is within 1 degree of the correct result.

This method will throw java.lang.NullPointerException if the parameter is null.

Class: Criteria

The criteria used for the selection of the location provider is defined by the values in this class. Instances of Criteria are used by the application to indicate criteria for choosing the location provider in the LocationProvider.getInstance method call.

The default values for the criteria fields are specified below in the table. The default values are always the least restrictive option that will match all location providers. Default values:

Criteria field	Default value
Horizontal accuracy	NO_REQUIREMENT
Vertical accuracy	NO_REQUIREMENT
Preferred response time	NO_REQUIREMENT
Power consumption	NO_REQUIREMENT
Cost allowed	true (allowed to cost)
Speed and course required	false (not required)
Altitude required	false (not required)
Address info required	false (not required)

The implementation of this class only retains the values that are passed in using the set* methods. It does not try to validate the values of the parameters in any way. Applications may set any values, even negative values, but the consequence may be that no matching LocationProvider can be created.

NO_REQUIREMENT

- No requirements for the parameter

POWER_USAGE_HIGH

- High power consumption allowed

POWER_USAGE_LOW

- Only low power consumption allowed

POWER_USAGE_MEDIUM

- Average power consumption allowed

Horizontal Accuracy	Vertical Accuracy	Cost	Power Consumption	Resulting Mode
Required - Set a value	Required - Set a value	false Not allowed	N/A	GPS_AID_MODE_AUTONOMOUS
Required - Set a value	Required - Set a value	true Allowed	POWER_USAGE_LOW, POWER_USAGE_MEDIUM or NO_REQUIREMENT	GPS_AID_MODE_AUTONOMOUS
Required - Set a value	Required - Set a value	true Allowed	POWER_USAGE_HIGH	1st fix - GPS_AID_MODE_ASSIST Subsequent fixes - GPS_AID_MODE_AUTONOMOUS
NO_REQUIREMENT	NO_REQUIREMENT	false Not allowed	POWER_USAGE_MEDIUM or POWER_USAGE_HIGH or NO_REQUIREMENT	GPS_AID_MODE_AUTONOMOUS
NO_REQUIREMENT	NO_REQUIREMENT	true Allowed	POWER_USAGE_MEDIUM or NO_REQUIREMENT	GPS_AID_MODE_ASSIST
NO_REQUIREMENT	NO_REQUIREMENT	true Allowed	POWER_USAGE_HIGH	1st fix - GPS_AID_MODE_ASSIST Subsequent fixes - GPS_AID_MODE_AUTONOMOUS
NO_REQUIREMENT	NO_REQUIREMENT	true Allowed	POWER_USAGE_LOW	GPS_AID_MODE_CELLSITE

Constructor

```
Criteria()
```

Method: Criteria.getHorizontalAccuracy

Syntax:

```
int getHorizontalAccuracy()
```

Description:

Returns the horizontal accuracy in meters that has been set in this Criteria.

Method: Criteria.getPreferredPowerConsumption

Syntax:

```
int getPreferredPowerConsumption()
```

Description:

Returns the preferred power consumption level.

- NO_REQUIREMENT
- POWER_LEVEL_LOW
- POWER_LEVEL_MEDIUM
- POWER_LEVEL_HIGH

Method: Criteria.getPreferredResponseTime

Syntax:

```
int getPreferredResponseTime()
```

Description:

Returns the preferred maximum response time in milliseconds.

Method: Criteria.getVerticalAccuracy

Syntax:

```
int getVerticalAccuracy()
```

Description:

Returns the vertical accuracy value in meters that has been set in this Criteria.

Method: Criteria.isAddressInfoRequired

Syntax:

```
boolean isAddressInfoRequired()
```

Description:

Returns true if the location provider should be able to determine textual address information.

Method: Criteria.isAllowedToCost

Syntax:

```
boolean isAllowedToCost()
```

Description:

Returns the preferred cost setting: true if allowed to cost, false if it must be free of charge

Method: Criteria.isAltitudeRequired

Syntax:

```
boolean isAltitudeRequired()
```

Description:

Returns true if the location provider should be able to determine altitude.

Method: Criteria.isSpeedAndCourseRequired

Syntax:

```
boolean isSpeedAndCourseRequired()
```

Description:

Returns true if the location provider should be able to determine speed and course.

Method: Criteria.setAddressInfoRequired

Syntax:

```
void setAddressInfoRequired
  (boolean addressInfoRequired)
```

Description:

Sets whether the location provider should be able to determine textual address information. The default is false. Setting this criteria to true implies that a location provider should be selected that is capable of providing the textual address information. This does not mean that every returned location instance necessarily will have all the address information filled in, though.

Method: Criteria.setAltitudeRequired

Syntax:

```
void setAltitudeRequired
  (boolean altitudeRequired)
```

Description:

Sets whether the location provider should be able to determine altitude. Default is false. If set to true, the LocationProvider is required to be able to normally determine the altitude.

Method: Criteria.setCostAllowed

Syntax:

```
void setCostAllowed(boolean costAllowed)
```

Description:

Sets the preferred cost setting where requests for location determination is allowed to incur financial cost to the user of the terminal. The default condition is true.

Method: Criteria.setHorizontalAccuracy

Syntax:

```
void setHorizontalAccuracy(int accuracy)
```

Description:

Sets the desired maximum horizontal accuracy preference in meters. The default is NO_REQUIREMENT, meaning no preference on horizontal accuracy.

Method: Criteria.setPreferredPowerConsumption

Syntax:

```
void setPreferredPowerConsumption(int level)
```

Description:

Sets the preferred maximum level of power consumption.

- NO_REQUIREMENT (default)
- POWER_LEVEL_LOW
- POWER_LEVEL_MEDIUM
- POWER_LEVEL_HIGH

Method: Criteria.setPreferredResponseTime

Syntax:

```
void setPreferredResponseTime(int time)
```

Description:

Sets the desired maximum response time preference in milliseconds. This value is typically used to determine a location method that is able to produce the location information within the defined time. The value is also used as a timeout value if the result cannot be produced within the defined time. The default is NO_REQUIREMENT, meaning no response time constraint.

Method: Criteria.setSpeedAndCourseRequired

Syntax:

```
void setSpeedAndCourseRequired
  (boolean speedAndCourseRequired)
```

Description:

Sets whether the location provider should be able to determine speed and course. The default is false. If set to true, the LocationProvider is required to be able to normally determine the speed and course.

Method: Criteria.setVerticalAccuracy

Syntax:

```
void setVerticalAccuracy(int accuracy)
```

Description:

Sets the desired vertical accuracy preference in meters. The default is NO_REQUIREMENT, meaning no preference on vertical accuracy.

Class: Landmark

Represents a landmark, which is a known location with a name. A landmark has a name by which it is known to the end user, a textual description, QualifiedCoordinates and optionally AddressInfo.

This class is only a container for the information. The constructor does not validate the parameters passed in but just stores the values, except that the name field is never allowed to be null. The get* methods return the values passed in the constructor.

Constructor

```
Landmark
(
  // Name of the landmark
  java.lang.String name,
  // Description of the landmark.
  // May be null if not available
  java.lang.String description,
  // The Coordinates of the landmark.
  // May be null if not known.
  QualifiedCoordinates coordinates,
  // The textual address information of the
  // landmark. May be null if not known.
  AddressInfo addressInfo
)
```

The constructor will throw java.lang.NullPointerException if the name is null.

Method: Landmark.getAddressInfo

Syntax:

```
AddressInfo getAddressInfo()
```

Description:

Gets the AddressInfo of the landmark.

Method: Landmark.getDescription

Syntax:

```
java.lang.String getDescription()
```

Description:

Gets the landmark description, null if not available.

Method: Landmark.getDescription

Syntax:

```
java.lang.String getName()
```

Description:

Gets the landmark name.

Method: Landmark.getQualifiedCoordinates

Syntax:

```
QualifiedCoordinates getQualifiedCoordinates()
```

Description:

Gets the QualifiedCoordinates of the landmark, null if not available.

Method: Landmark.setAddressInfo

Syntax:

```
void setAddressInfo(AddressInfo addressInfo)
```

Description:

Sets the AddressInfo of the landmark.

Method: Landmark.setDescription

Syntax:

```
void setDescription
  (java.lang.String description)
```

Description:

Sets the description of the landmark. Pass null to indicate that description is not available.

Method: Landmark.setName

Syntax:

```
void setName(java.lang.String name)
```

Description:

Sets the name of the landmark.

Method: Landmark.setQualifiedCoordinates

Syntax:

```
void setQualifiedCoordinates
  (QualifiedCoordinates cooordinates)
```

Description:

Sets the QualifiedCoordinates of the landmark.

Class: LandmarkStore

Provides methods to store, delete and retrieve landmarks from a persistent landmark store. There is one default landmark store and there may be multiple other landmark stores with different names. All landmark stores must be shared between all Java ME applications. Named landmark stores have unique names in this API.

The Landmarks have a name and may be placed in a category or several categories. The category is intended to group landmarks that are of similar type to the end user, for example, restaurants, museums, etc. The landmark names are strings that identify the landmark to the end user. The category names describe the category to the end user. The lan-

guage used in the names may be any and depends on the preferences of the end user. The names of the categories are unique within a LandmarkStore. However, the names of the landmarks are not guaranteed to be unique. Landmarks with the same name can appear in multiple categories or even several Landmarks with the same name in the same category.

The Landmark objects returned from the getLandmarks methods in this class shall guarantee that the application can read a consistent set of the landmark data valid at the time of obtaining the object instance, even if the landmark information in the store is modified subsequently by this or some other application.

The Landmark object instances can be in two states:

- Initially constructed by an application
- Belongs to a LandmarkStore

A Landmark object belongs to a LandmarkStore if it has been obtained from the LandmarkStore using getLandmarks or if it has been added to the LandmarkStore using addLandmark. A Landmark object is initially constructed by an application when it has been constructed using the constructor but has not been added to a LandmarkStore using addLandmark.

The landmark stores created by an application and landmarks added in landmark stores persist even if the application itself is deleted from the device.

Accessing the landmark store may cause a SecurityException if the calling application does not have the required permissions. The permissions to read and write (including add and delete) landmarks are distinct. An application having permission to read landmarks wouldn't necessarily have the permission to delete them.

Method: LandmarkStore.addCategory

Syntax:

```
void addCategory(java.lang.String categoryName)
```

Description:

Adds a category to this LandmarkStore.

This method will throw:

java.lang.IllegalArgumentException

- A category with the specified name already exists

java.lang.NullPointerException

- The parameter is null

javax.microedition.location.LandmarkException

- This LandmarkStore does not support adding new categories

java.io.IOException

- An I/O error occurs or there are no resources to add a new category

java.lang.SecurityException

- The application does not have the permission to manage categories

Method: LandmarkStore.addLandmark

Syntax:

```
void addLandmark
(
  Landmark landmark,
  java.lang.String category
)
```

Description

Adds a landmark to the specified category in the landmark store. Pass null as the category to indicate that the landmark does not belong to a category. Both parameters have a maximum length of 32 characters.

This method will throw:

java.lang.SecurityException

- The application is not allowed to add landmarks.

java.lang.IllegalArgumentException

- The landmark has a longer name field than the implementation can support or if the category defined in the landmark is not one of the categories supported by this LandmarkStore.

java.io.IOException

- An I/O error happened when accessing the landmark database or if there are no resources available to store this landmark.

java.lang.NullPointerException

- The landmark parameter is null.

Method: LandmarkStore.createLandmarkStore

Syntax:

```
static void createLandmarkStore
  (java.lang.String storeName)
```

Description:

Creates a new landmark store with a specified name.All LandmarkStores are shared between all Java ME applications and may be shared with native applications.

This method will throw:

java.lang.IllegalArgumentException

- The name is too long or if a landmark store with the specified name already exists

java.lang.NullPointerException

- The parameter is null

javax.microedition.location.LandmarkException

- The implementation does not support creating new landmark stores

java.io.IOException

- The landmark store couldn't be created due to an I/O error

java.lang.SecurityException

- The application does not have permissions to create a new landmark store

Method: LandmarkStore.deleteCategory

Syntax:

```
void deleteCategory
  (java.lang.String categoryName)
```

Description

Removes a category from this LandmarkStore. This method will not remove any of the landmarks, only the associated category information from the landmarks. If a category with the supplied name does not exist in this LandmarkStore, the method returns silently with no error.

This method will throw:

java.lang.NullPointerException

- The parameter is null

javax.microedition.location.LandmarkException

- This LandmarkStore does not support deleting categories

java.io.IOException

- An I/O error occurs

java.lang.SecurityException

- The application does not have the permission to manage categories

Method: LandmarkStore.deleteLandmark

Syntax:

```
void deleteLandmark(Landmark lm)
```

Description:

Removes a landmark from all categories and deletes the information from this LandmarkStore. The Landmark instance passed in as the parameter must be an instance that belongs to this LandmarkStore. If the Landmark is not found in this LandmarkStore, then the request is silently ignored and the method call returns with no error.

This method will throw:

java.lang.SecurityException

- The application is not allowed to delete the landmark

javax.microedition.location.LandmarkException

- The landmark instance passed as the parameter does not belong to this LandmarkStore

java.io.IOException

- An I/O error happened when accessing the landmark store

java.lang.NullPointerException

- The parameter is null

Method: LandmarkStore.deleteLandmarkStore

Syntax:

```
static void deleteLandmarkStore
  (java.lang.String storeName)
```

Description:

Delete a landmark store with a specified name. All the landmarks and categories defined in the named landmark store are irrevocably removed. If a landmark store with the specified name does not exist, this method returns silently without any error.

This method will throw:

java.lang.NullPointerException

- The parameter is null (the default landmark store can't be deleted)

javax.microedition.location.LandmarkException

- The implementation does not support deleting landmark stores

java.io.IOException

- The landmark store couldn't be deleted due to an I/O error

java.lang.SecurityException

- The application does not have permissions to delete a landmark store

Method: LandmarkStore.getCategories

Syntax:

```
java.util.Enumeration getCategories()
```

Description:

Returns the category names that are defined in this LandmarkStore. The language and locale used for these names depends on the implementation and end user settings. The names shall be such that they can be displayed to the end user and have a meaning to the end user.

This method returns an java.util.Enumeration containing Strings representing the category names. If there are no categories defined in this LandmarkStore, an Enumeration with no entries is returned.

Method: LandmarkStore.getInstance

Syntax:

```
static LandmarkStore getInstance
  (java.lang.String storeName)
```

Description:

Gets a LandmarkStore instance for storing, deleting and retrieving landmarks. There must be one default landmark store and there may be other landmark stores that can be accessed by name. If null is passed, the default landmark store will be returned. Throws java.lang.SecurityException if the application does not have a permission to read landmark stores.

Method: LandmarkStore.getLandmarks

Syntax:

```
java.util.Enumeration getLandmarks()
```

Description:

Lists all landmarks stored in the database and returns an java.util.Enumeration object containing Landmark objects representing all the landmarks stored in this LandmarkStore. The method will throw a java.io.IOException if an I/O error happened when accessing the landmark database.

Method: LandmarkStore.getLandmarks

Syntax:

```
java.util.Enumeration getLandmarks
(
  java.lang.String category,
  double minLatitude,
  double maxLatitude,
  double minLongitude,
  double maxLongitude
)
```

Description:

Lists all the landmarks that are within an area defined by bounding minimum and maximum latitude and longitude. The bounds are considered to belong to the area.

If minLongitude <= maxLongitude, then this area covers the longitude range [minLongitude, maxLongitude].

If minLongitude > maxLongitude, then this area covers the longitude range [-180.0, maxLongitude] and [minLongitude, 180.0).

For latitude, the area covers the latitude range [minLatitude, maxLatitude].

This method will throw java.io.IOException if an I/O error happened when accessing the landmark database, and will throw java.lang.IllegalArgumentException if the minLongitude or maxLongitude is out of the range [-180.0, 180.0), or minLatitude or minLongitude is out of the range [-90.0,90.0], or if minLatitude > maxLatitude

Method: LandmarkStore.getLandmarks

Syntax:

```
java.util.Enumeration getLandmarks
(
  java.lang.String category,
  java.lang.String name
)
```

Description:

Gets the Landmarks from the storage where the category and/or name matches the given parameters. The category of the landmark is passed, or null which implies a wildcard that matches all categories. The name of the desired landmark is also passed where null implies a wildcard that matches all the names within the category indicated by the category parameter.

This method returns an Enumeration containing all the matching Landmarks or null if no Landmark matched the given parameters.

This method throws java.io.IOException if an I/O error happened when accessing the landmark database.

Method: LandmarkStore.listLandmarkStores

Syntax:

```
static java.lang.String[] listLandmarkStores()
```

Description:

Lists the names of all the available landmark stores.The default landmark store is obtained from getInstance by passing null as the parameter. The null name for the default landmark store is not included in the list returned by this method. If there are no named landmark stores, other than the default landmark store, this method returns null.

This method returns an array of landmark store names, and will throw:

java.lang.SecurityException

- The application does not have the permission to access landmark stores.

java.io.IOException

- An I/O error occurred when trying to access the landmark stores.

Method: LandmarkStore.removeLandmarkFromCategory

Syntax:

```
void removeLandmarkFromCategory
(
  Landmark lm,
  java.lang.String category
)
```

Description:

Removes the named landmark from the specified category.

The Landmark instance passed in as the parameter must be an instance that belongs to this LandmarkStore.

If the Landmark is not found in this LandmarkStore in the specified category or if the parameter is a Landmark instance that does not belong to this LandmarkStore, then the request is silently ignored and the method call returns with no error. The request is also silently ignored if the specified category does not exist in this LandmarkStore.

The landmark is only removed from the specified category but the landmark information is retained in the store. If the landmark no longer belongs to any category, it can still be obtained from the store by passing null as the category to getLandmarks.

This method will throw:

java.lang.NullPointerException

- At least one parameter is null.

java.io.IOException

- An I/O error happened when accessing the landmark store.

java.lang.SecurityException

- The application is not allowed to the landmark.

Method: LandmarkStore.updateLandmark

Syntax:

```
void updateLandmark(Landmark lm)
```

Description:

Updates the information about a landmark. This method only updates the information about a landmark and does not modify the categories the landmark belongs to. The Landmark instance passed in as the parameter must be an instance that belongs to this LandmarkStore. This method can't be used to add a new landmark to the store.

This method will throw:

java.lang.NullPointerException

- The parameter is null.

javax.microedition.location.LandmarkException

- The landmark instance passed as the parameter does not belong to this LandmarkStore or does not exist in the store any more.

java.io.IOException

- An I/O error happened when accessing the landmark store.

java.lang.SecurityException

- The application is not allowed to update the landmark.

Class: Location

Represents the standard set of basic location information. This includes the time stamped coordinates, accuracy, speed, heading, and information about the positioning method used for the location, plus an optional address.

The location method is indicated using a bit field. The individual bits are defined using constants in this class. This bit field is a bitwise combination of the location method technology bits (MTE_*), method type (MTY_*) and method assistance information (MTA_*). All other bits in the 32 bit integer than those that have defined constants in this class are reserved and MUST not be set by implementations (i.e. these bits must be 0).

A Location object may be either 'valid' or 'invalid'. The validity can be queried using the isValid method. A valid Location object represents a location with valid coordinates. An invalid Location object doesn't have valid coordinates, but the extra info that is obtained from the getExtraInfo method can provide information about the reason why it was not possible to provide a valid Location. The periodic location updates to the LocationListener may return invalid Location objects if it isn't possible to determine the location. This class is only a container for the information.

Field	Description
MTA_ASSISTED	Location method is assisted by the other party
MTA_UNASSISTED	Location method is unassisted
MTE_ANGLEOFARRIVAL	Location method Angle of Arrival for cellular / terrestrial RF system.
MTE_CELLID	Location method Cell-ID for cellular (in GSM, this is the same as Cell Global Identity (CGI))
MTE_SATELLITE	Location method using satellites (for example, Global Positioning System (GPS))
MTE_SHORTRANGE	Location method Short-range positioning system (for example, Bluetooth LP)

Field	Description
MTE_TIMEDIFFERENCE	Location method Time Difference for cellular / terrestrial RF system (for example, Enhanced Observed Time Difference (E-OTD) for GSM)
MTE_TIMEOFARRIVAL	Location method Time of Arrival (TOA) for cellular / terrestrial RF system
MTY_NETWORKBASED	Location method is of type network based
MTY_TERMINALBASED	Location method is of type terminal based

Constructor

```
Location()
```

Method: Location.getAddressInfo

Syntax:

```
AddressInfo getAddressInfo()
```

Description:

Returns null because the RIM implementation does not support textual address.

Method: Location.getCourse

Syntax:

```
float getCourse()
```

Description:

Returns the device's course in degrees relative to true north. The value is always in the range [0.0,360.0) degrees. Returns Float.NaN if the course is not known.

Method: Location.getExtraInfo

Syntax:

```
java.lang.String getExtraInfo
  (java.lang.String mimetype)
```

Description:

Returns extra information about the location. This method is intended to provide location method specific extra information that applications that are aware of the used location method and information format are able to use.

A MIME type is used to identify the type of the extra information when requesting it. It returns the extra information as a String encoded according to format identified by the MIME type.

When the MIME type is "application/X-java-location-nmea", the returned string shall be a valid sequence of NMEA sentences formatted according to the syntax specified in the NMEA 0183 v3.1 specification. These sentences should represent the set of NMEA sentences that are related to this location at the time this location was created.

A sample returned string of getExtraInfo("application/X-jsr179-location-nmea"):

```
$GPGGA,140234,26:08.76784,N,-80:15.22240,W,1,
  6,,7.0,M,,
$GPGLL,26:08.76784,N,-80:15.22240,W,140234,A
```

When the MIME type is "application/X-java-location-lif", the returned string shall contain an XML formatted document containing the "pd" element defined in the LIF Mobile Location Protocol TS 101 v3.0.0 as the root element of the document.

A sample returned string of getExtraInfo("application/X-jsr179-location-lif"). Note that the following string has been formatted for display purposes:

```
lif:
<pd>
  <time>1105044947940</time>
  <shape>
    <Point>
      <coord><X>-23.67</X><Y>34.45</Y></coord>
    </Point>
  </shape>
  <alt>456.0</alt>
  <alt_acc>23.0</alt_acc>
  <speed>20</speed>
  <direction>120</direction>
</pd>
```

When the MIME type is "text/plain", the returned string shall contain textual extra information that can be displayed to the end user.

Method: Location.getLocationMethod

Syntax:

```
int getLocationMethod()
```

Description:

Returns information about the location method used. The returned value is a bitwise combination (OR) of the method technology, method type and assistance information. The method technology values are defined as constant values named MTE_* in this class, the method type values are named MTY_* and assistance information values are named MTA_*.

For example, if the location method used is device based, network assisted Enhanced Observed Time Difference (E-OTD), the following value would be returned:

```
0x00050002 ( = MTY_TERMINALBASED | MTA_ASSISTED
  | MTE_TIMEDIFFERENCE)
```

If the location is determined by combining several location technologies, the returned value may have several MTE_* bits set.

If the used location method is unknown, the returned value may have all the bits set to zero.

Only bits that have defined constants within this class are allowed to be used. Other bits are reserved and must be set to 0.

Method: Location.getQualifiedCoordinates

Syntax:

```
QualifiedCoordinates getQualifiedCoordinates()
```

Description:

Returns the coordinates of this location and their accuracy, or null if not known.

Method: Location.getSpeed

Syntax:

```
float getSpeed()
```

Description:

Returns the device's current ground speed in meters per second (m/s) at the time of measurement, or Float.NaN if the speed is not known. The speed is always a non-negative value. Note that unlike the coordinates, speed does not have an associated accuracy because the methods used to determine the speed typically are not able to indicate the accuracy.

Method: Location.getTimestamp

Syntax:

```
long getTimestamp()
```

Description:

Returns the time stamp at which the data was collected. This timestamp should represent the point in time when the measurements were made. The time returned is the local device time in milliseconds using the same clock and same time representation as System.currentTimeMillis().

Method: Location.isValid

Syntax:

```
boolean isValid()
```

Description:

Returns whether this Location instance represents a valid location with coordinates or an invalid one where all the data, especially the latitude and longitude coordinates, may not be present.

A valid Location object contains valid coordinates whereas an invalid Location object may not contain valid coordinates but may contain other information via the getExtraInfo() method to provide information on it was not possible to provide a valid Location object.

This method returns a boolean value with true indicating that this Location instance is valid and false indicating an invalid Location instance

Class: LocationProvider

This is the starting point for applications using this API and represents a source of the location information. A LocationProvider represents a location-providing module, generating Locations.

Applications obtain LocationProvider instances (classes implementing the actual functionality by extending this abstract class) by calling the factory method. It is the responsibility of the implementation to return the correct LocationProvider-derived object.

Applications that need to specify criteria for the location provider selection, must first create a Criteria object, and pass it to the factory method. The methods that access the location related information shall throw SecurityException if the application does not have the relevant permission to access the location information.

RIM Implementation Note

Textual address is not supported. If an application invokes LocationProvider.getInstance(criteria) with a Criteria instance that requires AddressInfo, it returns null.

AVAILABLE

- Availability status code: the location provider is available

OUT_OF_SERVICE

- Availability status code: the location provider is permanently unavailable. Permanent unavailability means that the method is unavailable and the implementation is not able to expect that this situation would change in the near future. An example is when using a location method implemented in an external device and the external device is detached.

TEMPORARILY_UNAVAILABLE

- Availability status code: the location provider is temporarily unavailable. Temporary unavailability means that the method is unavailable due to reasons that can be expected to possibly change in the future and the provider to become available. An example is not being able to receive the signal because the signal used by the location method is currently being obstructed, e.g. when deep inside a building for satellite based methods. However, a very short transient obstruction of the signal should not cause the provider to toggle quickly between TEMPORARILY_UNAVAILABLE and AVAILABLE

Constructor

```
protected  LocationProvider()
```

Empty constructor to help implementations and extensions. This is not intended to be used by applications. Applications should not make subclasses of this class and invoke this constructor from the subclass.

Method: LocationProvider.addProximityListener

Syntax:

```
static void addProximityListener
(
  ProximityListener listener,
  Coordinates coordinates,
  float proximityRadius
)
```

Description:

Adds a ProximityListener for updates when proximity to the specified coordinates is detected.

If this method is called with a ProximityListener that is already registered, the registration to the specified coordinates is added in addition to the set of coordinates it has been previously registered for. A single listener can handle events for multiple sets of coordinates.

If the current location is known to be within the proximity radius of the specified coordinates in meters, the listener shall be called immediately.

Detecting the proximity to the defined coordinates is done on a best effort basis. Due to the limitations of the methods used to implement this, there are no guarantees that the proximity is always detected; especially in situations where the device briefly enters the proximity area and exits it shortly afterwards, it is possible that it will be missed. It is optional to provide this feature as it may not be reasonably implementable with all methods used to implement this API.

If the implementation is capable of supporting the proximity monitoring and has resources to add the new listener and coordinates to be monitored but the monitoring can't be currently done due to the current state of the method used to implement it, this method shall succeeed and the monitoringStateChanged method of the listener shall be immediately called to notify that the monitoring is not active currently.

This method throws:

javax.microedition.location.LocationException

- The platform does not have resources to add a new listener and coordinates to be monitored or does not support proximity monitoring at all

java.lang.IllegalArgumentException

- The proximity radius is 0 or negative* or Float.NaN

java.lang.NullPointerException

- The listener or coordinates parameter is null

java.lang.SecurityException

- The application does not have the permission to register a proximity listener

Method: LocationProvider.getInstance

Syntax:

```
static LocationProvider getInstance
  (Criteria criteria)
```

Description:

This factory method is used to get an actual LocationProvider implementation based on the defined criteria. Pass null if the default provider is requested. If no concrete LocationProvider could be created that typically can match the defined criteria but there are other location providers not meeting the criteria that could be returned for a more relaxed criteria, null is returned to indicate this. The LocationException is thrown, if all supported location providers are out of service.

A LocationProvider instance is returned if there is a location provider meeting the criteria in either the available or temporarily unavailable state. If a LocationProvider meeting the criteria can be supported but is currently out of service, it shall not be returned.

When this method is called with a Criteria that has all fields set to the default values (i.e. the least restrictive criteria possible), it will return a LocationProvider if there is any provider that isn't in the out of service state. Passing null as the parameter is equal to passing a Criteria that has all fields set to the default values, i.e. the least restrictive set of criteria.

This method only makes the selection of the provider based on the criteria and is intended to return it quickly to the application. Any possible initialization of the provider is done at an implementation dependent time and must not block the call to this method.

This method may return the same LocationProvider instance as has been returned previously from this method to the calling application, if the same instance can be used to fulfil both defined criteria. Note that there can be only one LocationListener associated with a LocationProvider instance.

This method throws javax.microedition.location.LocationException if all LocationProviders are unavailable.

Method: LocationProvider.getLastKnownLocation

Syntax:

```
static Location getLastKnownLocation()
```

Description:

Returns the latest known location. This is the best estimate for the previously known location. null is returned if there isn’t any previous location information.

Applications can use this method to obtain the last known location and check the timestamp and other fields to determine if this is recent enough and good enough for the application to use without needing to make a new request for the current location.

This method throws java.lang.SecurityException if the calling application does not have a permission to query the location information.

Method: LocationProvider.getLocation

Syntax:

```
abstract Location getLocation(int timeout)
```

Description:

Retrieves a Location with the constraints given by the Criteria associated with this class. If no result could be retrieved, a LocationException is thrown. If the location can't be determined within the timeout period specified in the parameter, the method shall throw a LocationException.

If the provider is temporarily unavailable, the device will wait and try to obtain the location until the timeout expires. If the provider is permanently unavailable, then the LocationException is thrown immediately.

Note that the individual Location returned might not fulfill exactly the criteria used for selecting this LocationProvider. The Criteria is used to select a location provider that typically is able to meet the defined criteria, but not necessarily for every individual location measurement.

The single parameter passed is a timeout value in seconds. -1 is used to indicate that a default timeout value for this provider should be used.

This method throws:

javax.microedition.location.LocationException

- The location couldn't be retrieved or if the timeout period expired

java.lang.InterruptedException

- The operation is interrupted by calling reset() from another thread

java.lang.SecurityException

- The calling application does not have a permission to query the location information

java.lang.IllegalArgumentException

- The timeout == 0 or timeout < -1

Method: LocationProvider.getState

Syntax:

```
abstract int getState()
```

Description:

Returns the current state of this LocationProvider. The return value shall be one of the availability status code constants defined in this class.

Method: LocationProvider.removeProximityListener

Syntax:

```
static void removeProximityListener
  (ProximityListener listener)
```

Description:

Removes a ProximityListener from the list of recipients for updates. If the specified listener is not registered or if the parameter is null, this method silently returns with no action.

Method: LocationProvider.reset

Syntax:

```
abstract void reset()
```

Description:

Resets the LocationProvider.

All pending synchronous location requests will be aborted and any blocked getLocation method calls will terminate with InterruptedException.

Applications can use this method e.g. when exiting to have its threads freed from blocking synchronous operations.

Method: LocationProvider.setLocationListener

Syntax:

```
abstract void setLocationListener
(
  LocationListener listener,
  int interval,
  int timeout,
  int maxAge
)
```

Description:

Adds a LocationListener for updates at the defined interval. The listener will be called with updated location at the defined interval. The listener also gets updates when the availablilty state of the LocationProvider changes.

Passing an interval of -1 selects the default interval which is dependent on the location method used. Passing an interval of 0 registers the listener to only receive provider status updates and not location updates at all.

Only one listener can be registered with each LocationProvider instance. Setting the listener replaces any possibly previously set listener. Setting the listener to null cancels the registration of any previously set listener.

The first location result will be obtained when the listener is registered and will provide the location to the listener as soon as it is available. Subsequent location updates will happen at the defined interval after the first one. If the specified update interval is smaller than the time it takes to obtain the first result, the listener shall receive location updates with invalid Locations at the defined interval until the first location result is available.

The timeout parameter determines a timeout that is used if it's not possible to obtain a new location result when the update is scheduled to be provided. This timeout value indicates how many seconds the update is allowed to be provided late compared to the defined interval. If it's not possible to get a new location result (interval + timeout) seconds after the previous update, the update will be made and an invalid Location instance is returned. This is also done if the reason for the inability to obtain a new location result is due to the provider being temporarily unavailable or out of service. For example, if the interval is 60 seconds and the timeout is 10 seconds, the update must be delivered at most 70 seconds after the previous update and if no new location result is available by that time the update will be made with an invalid Location instance.

The maxAge parameter defines how old the location result is allowed to be provided when the update is made. This allows the implementation to reuse location results if it has a recent location result when the update is due to be delivered. This parameter can only be used to indicate a larger value than the normal time of obtaining a location result by a location method. The normal time of obtaining the location result means the time it takes normally to obtain the result when a request is made. If the application specifies a time value that is less than what can be realized with the used location method, the implementation shall provide as recent location results as are possible with the used location method. For example, if the interval is 60 seconds, the maxAge is 20 seconds and normal time to obtain the result is 10 seconds, the implementation would normally start obtaining the result 50 seconds after the previous update. If there is a location result otherwise available that is more recent than 40 seconds after the previous update, then the maxAge setting to 20 seconds allows to return this result and not start obtaining a new one.

Parameters:

listener

- The listener to be registered.
- If set to null the registration of any previously set listener is cancelled.

interval

- The interval in seconds.
- -1 is used for the default interval of this provider.
- 0 is used to indicate that the application wants to receive only provider status updates and not location updates at all.

timeout

- Timeout value in seconds
- Must be greater than 0.
- The default timeout for this provider is used if the value is -1.
- If the interval is -1 to indicate the default, the value of this parameter has no effect and the default timeout for this provider is used.
- This parameter has no effect if the interval is 0.

maxAge

- Maximum age of the returned location in seconds
- Must be greater than 0 or equal to -1 to indicate that the default maximum age for this provider is used.
- If the interval is -1 to indicate the default, the value of this parameter has no effect and the default maximum age for this provider is used.
- This parameter has no effect if the interval is 0.

This method throws:

java.lang.IllegalArgumentException

- If interval < -1, or if (interval != -1) and (timeout > interval or maxAge > interval or (timeout < 1 and timeout != -1) or (maxAge < 1 and maxAge != -1))

java.lang.SecurityException

- The calling application does not have a permission to query the location information

Class: Orientation

The Orientation class represents the physical orientation of the device. Orientation is described by azimuth to north (the horizontal pointing direction), pitch (the vertical elevation angle) and roll (the rotation of the device around its own longitudinal axis).

The device providse the compass bearing, pitch and roll information. Most commonly, this class will be used to obtain the current compass direction.

It is up to the device to define its own axes, but it is generally recommended that the longitudinal axis is aligned with the bottom-to-top direction of the screen. This means that the pitch is positive when the top of the screen is up and the bottom of the screen down (when roll is zero). The roll is positive when the device is tilted clockwise looking from the direction of the bottom of the screen, i.e. when the left side of the screen is up and the right side of the screen is down (when pitch is zero).

No accuracy data is given for Orientation.

This class is only a container for the information. The constructor does not validate the parameters passed in but just retains the values. The get* methods return the values passed in the constructor.

Constructor:

```
Orientation
(
  float azimuth,
  boolean isMagnetic,
  float pitch,
  float roll
)
```

Constructs a new Orientation object with the bearing, pitch and roll parameters specified.

The values are expressed in degress using floating point values.

If the pitch or roll is undefined, the parameter should be given as Float.NaN.

Parameters:

bearing

- The bearing relative to true or magnetic north.
- Valid range: [0.0, 360.0)

isMagnetic

- true if the bearing is relative to the magnetic field or the Earth
- false if the bearing is relative to true north and gravity

pitch

- The pitch of the terminal in degrees.
- Valid range: [-90.0, 90.0]

roll

- The roll of the terminal in degrees.
- Valid range: [-180.0, 180.0)

Method: Orientation.getCompassAzimuth

Syntax:

```
float getCompassAzimuth()
```

Description:

Returns the device's surface orientation in degrees relative to either magnetic or true north. The value is always in the range [0.0, 360.0) degrees. The isOrientationMagnetic() method indicates whether the returned bearing is relative to true north or magnetic north.

Method: Orientation.getOrientation

Syntax:

```
static Orientation getOrientation()
```

Description:

Returns the device's current orientation or null if it can't be determined.

This method throws:

javax.microedition.location.LocationException

- The implementation does not support orientation determination

java.lang.SecurityException

- The calling application does not have permission to query the orientation

Method: Orientation.getPitch

Syntax:

```
float getPitch()
```

Description:

Returns the device's tilt in degrees defined as an angle in the vertical plane orthogonal to the ground, and through the longitudinal axis of the device. The value is always in the range [-90.0, 90.0] degrees, or Float.NaN if not available. A negative value means that the top of the device is pointing towards the ground.

Method: Orientation.getRoll

Syntax:

```
float getRoll()
```

Description:

Returns the device's rotation in degrees around its own longitudinal axis, or Float.NaN if not available. The value is always in the range [-180.0, 180.0) degrees. A negative value means that the device is orientated anti-clockwise from its default orientation.

Method: Orientation.isOrientationMagnetic

Syntax:

```
boolean isOrientationMagnetic()
```

Description:

Returns a boolean value that indicates whether this Orientation is relative to the magnetic field of the Earth or relative to true north and gravity. If this method returns true, the compass bearing and pitch are relative to the magnetic field of the Earth. If this method returns false, the compass bearing is relative to true north and pitch is relative to gravity.

Class: QualifiedCoordinates

The QualifiedCoordinates class represents coordinates as latitude-longitude-altitude values that are associated with an accuracy value.

Fields inherited from class javax.microedition.location.Coordinates:

- DD_MM
- DD_MM_SS

Constructor

```
QualifiedCoordinates
(
  double latitude,
  double longitude,
  float altitude,
  float horizontalAccuracy,
  float verticalAccuracy
)
```

Constructs a new QualifiedCoordinates object with the values specified. The latitude and longitude parameters are expressed in degrees using floating point values. The degrees are in decimal values (rather than minutes/seconds).

The coordinate values always apply to the WGS84 datum.

The Float.NaN value can be used for altitude to indicate that altitude is not known.

Parameters:

latitude

- The latitude of the location.
- Valid range: [-90.0, 90.0]

longitude

- The longitude of the location.
- Valid range: [-180.0, 180.0)

altitude

- The altitude of the location in meters, defined as height above WGS84 ellipsoid.
- Float.NaN can be used to indicate that altitude is not known.

horizontalAccuracy

- The horizontal accuracy of this location result in meters.
- Float.NaN can be used to indicate that the accuracy is not known.
- Must be greater or equal to 0.

verticalAccuracy

- The vertical accuracy of this location result in meters.
- Float.NaN can be used to indicate that the accuracy is not known.
- Must be greater or equal to 0.

This method throws java.lang.IllegalArgumentException if an input parameter is out of the valid range.

Method: QualifiedCoordinates.getHorizontalAccuracy

Syntax:

```
float getHorizontalAccuracy()
```

Description:

Returns the horizontal accuracy of the location in meters (1-sigma standard deviation). A value of Float.NaN means the horizontal accuracy could not be determined.

The horizontal accuracy is the RMS (root mean square) of east accuracy (latitudinal error in meters, 1-sigma standard deviation), north accuracy (longitudinal error in meters, 1-sigma).

Method: QualifiedCoordinates.getHorizontalAccuracy

Syntax:

```
float getVerticalAccuracy()
```

Description:

Returns the accuracy of the location in meters in vertical direction (orthogonal to ellipsoid surface, 1-sigma standard deviation). A value of Float.NaN means the vertical accuracy could not be determined.

Method: QualifiedCoordinates.setHorizontalAccuracy

Syntax:

```
void setHorizontalAccuracy
  (float horizontalAccuracy)
```

Description:

Sets the horizontal accuracy of the location in meters (1-sigma standard deviation). Must be greater or equal to 0. A value of Float.NaN means the horizontal accuracy could not be determined.

The horizontal accuracy is the RMS (root mean square) of east accuracy (latitudinal error in meters, 1-sigma standard deviation), north accuracy (longitudinal error in meters, 1-sigma).

Method: QualifiedCoordinates.setVerticalAccuracy

Syntax:

```
void setVerticalAccuracy
  (float verticalAccuracy)
```

Description:

Sets the accuracy of the location in meters in vertical direction (orthogonal to ellipsoid surface, 1-sigma standard deviation). The number must be greater or equal to 0. A value of Float.NaN means the vertical accuracy could not be determined.

Exception: LandmarkException

The LandmarkException is thrown when an error related to handling landmarks has occurred.

Constructors

`LandmarkException()`

- Constructs a LandmarkException with no detail message.

`LandmarkException(java.lang.String s)`

- Constructs a LandmarkException with the specified detail message.

Exception: LocationException

The LocationException is thrown when a location API specific error has occurred. The detailed conditions when this exception is thrown are documented in the methods that throw this exception.

Constructors

`LocationException()`

- Constructs a LocationException with no detail message.

`LocationException(java.lang.String s)`

- Constructs a LocationException with the specified detail message.

In closing ...

It would be hard to beat the sample device and server-side code that comes with the BlackBerry JDE v4.01 update. As such, download and install the latest update, adjust the code to reflect the server where you will deploy the sample server component, compile then deploy the sample application on your BlackBerry 7520 handheld. After that, compile and deploy the GPS server sample, and give it a test spin.

Please email your comments, suggestions and editorial submissions to Editor@BlackBerryDeveloperJournal.com