May 2004

BLACKBERRY DEVELOPER JOURNAL

main() 2

Java Bits & Pieces 3

C++ Bits & Pieces 6

Storing Data Persistently 7

- How to create, read, write, delete and otherwise manipulate persistent data using the BlackBerry API and MIDP RecordStore

FONTS 12

- Details on how to programmatically use fonts

C++ Edit Field in Over-ride 14

- How to create and use a custom edit field

Give Me A Sign 16

- BlackBerry application code signing 101

Let me introduce you to ... 18

- ... the Plazmic Content Development Kit!!

Gaming 101 - My First Experience 19

- Tips and techniques on how to develop a game for the BlackBerry with complete Java source code

© 2004 Research In Motion Limited. All rights reserved. The BlackBerry and RIM families of related marks, images and symbols are the exclusive properties and trademarks of Research In Motion Limited. RIM, Research In Motion, "Always On, Always Connected", the 'envelope in motion' symbol, BlackBerry, BlackBerry Wireless Handheld, BlackBerry Enterprise Server, and the BlackBerry logo are registered with the U.S. Patent and Trademark Office and may be pending or registered in other countries. All other brands, product names, company names, trademarks and service marks are the properties of their respective owners. The specifications and features contained in this document are subject to change without notice.

main()

A statement is made in Joseph Weizenbaum's book, “Computer Power and Human Reason” that has great relevance:

“The computer programmer is a creator of universes for which he alone is the lawgiver. Universes of virtually unlimited complexity can be created in the form of computer programs. Moreover, and this is a crucial point, systems so formulated and elaborated act out their programmed scripts. They compliantly obey their laws and vividly exhibit their obedient behavior. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or field of battle and to command such unswervingly dutiful actors or troops.”

This rings true for most software engineers. We strive to create digital universes that obey laws that we put into place. If we fail to design or implement our universes well, failures minor to catastrophic will occur.

BlackBerry has created a small, powerful digital universe within a handheld that has permanently changed our world. With each revision, the universe gets larger, more stable, and definitely better.

As a developer of BlackBerry applications, you are on the ground floor to catch the wave of BlackBerry's success. As sales increase and technology evolves, users will need an ever-changing kaleidoscope of software, software that only you can create.

Are you ready to create new universes of your own? We are, and we'll try our best each issue to share tricks and techniques, and much more, to help you along.

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@blackberrydeveloper.com**.

BlackBerry Developer Journal team

Java Bits & Pieces

Checking for a 0-length string

Inefficient:

```
if( str.equals( "" ) ) {
```

Efficient:

```
if( str.length() == 0 ) {
```

Loops

This loop:

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

... executes more quickly than this loop:

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

This loop executes more quickly than either of the above, and should be used when direction is not important:

```
for( int i = v.size()-1; i >= 0; --i ) {
```

CAUTION: Include a comment if the size of v can change:

```
// vector's size can change during loop
for ( int i = 0; i < v.size(); ++i ) {
```

Class & instance variable defaults

In Java, instance variables within a class are always set to a default value. This does not apply to variables declared within a method, which must be explicitly initialized before use.

From the Java Language Specification:

Each class variable, instance variable, or array component is initialized with a default value when it is created (§15.9, §15.10):

Type	Default Value
byte	(byte)0
short	(short)0
int	0
long	0L
float	Positive 0.0f
double	Positive 0.0d
char	null ‘\u0000’
boolean	false
class, interface and array types	Null

So, writing:

```
private boolean _foo = false;
```

... is redundant and has a negative impact on code size and performance.

Instead, use:

```
private boolean _foo;
```

If you like the commenting effect, use this form instead:

```
private boolean _foo; // = false
```

object instanceof <type> is false if object is null

Both of these:

```
if ( obj != null && obj instanceof Foo ) {
if ( obj == null || !( obj instanceof Foo )) {
```

... can be reduced to:

```
if ( obj instanceof Foo ) {}
if ( ! ( obj instanceof Foo ) ) {}
```

... producing smaller and faster code.

String literals

Avoid:

```
new String( "literal" );
```

The expression "literal" is an instance of *java.lang.String* so creating another object via:

```
new String( "literal" );
```

... is wasting time and space. The only use of the *String(String)* constructor should be in the Sun Technology Compatibility Kit (TCK), since the new copy of the string is indistinguishable from the original except that it has a different reference. Recall that *java.lang.String* cannot be modified after they are created, so creating carbon copies often isn't necessary.

Another case takes this form:

```
new String( "found " + n + " item(s)" );
```

Once again, a useless extra *String* is created. Use this form instead:

```
String foo = "found " + n + " item(s)";
```

Division by 2

Code that divides a positive number by 2:

```
midpoint = width / 2;
```

... should use shift-right by one instead:

```
midpoint = width >> 1;
```

Note: *This only works with positive integers.* Negative numbers require divide

instanceof keyword

The compiler has a special case construct to improve the performance of this common idiom:

```
if( x instanceof A ) {
  A y = (A)x;
  ...
}
```

In simple terms, if 'x' is a of type 'A', then interpret 'x' as being of type 'A' and store it in 'y'. Where 'y' is declared to be of type 'A'.

Close inspection of what javac generates for this construct reveals byte code like this:

```
load x
instanceof A  // is 'x' of type 'A'?
if false goto label

load x
checkcast A  // interpret 'x' as type 'A'
store y
...

label:
```

It turns out that *instanceof* and *checkcast* have to do almost exactly the same work. The main difference is that *instanceof* accepts a reference on the stack and produces a *Boolean* on the stack. *checkcast* accepts a reference on the stack and either leaves it there, now of the checked type, or throws a *ClassCastException*. The bottom line is that all the type checking work is done twice.

Another difference between *instanceof* and *checkcast* is that *null* is never an *instanceof* anything, whereas *null* will *checkcast* to any reference type. This means that it is unnecessary to do a null check along with the *instanceof* in the expression of an *if* statement.

The compiler recognizes similar byte code sequences and transforms them into an equivalent alternate sequence that only does the type checking once. This is a good thing.

Unfortunately, the compiler needs the javac generated byte code to be close to what it expects and not to contain other constructs that would invalidate the correctness of the transformation. In order to ensure that Java source will be translated into byte code that is recognizable by the compiler, it is necessary to follow a few simple rules.

1. The expressions used to load 'x' first for the *instanceof* and then again for the *checkcast* must be identical. They may only consist of data read operations. They may not write to a variable, or invoke a method.
2. The *instanceof* and *checkcast* must be separated by only the conditional branch and the second load of 'x'. The *if* expression shouldn't check anything else after the *instanceof* and the first statement after the inside the *if* block should be the cast.

These won't work:

```
if( fetch() instanceof A ) {
  A y = (A)fetch();
  ...
}

if( x instanceof A && otherResult ) {
  A y = (A)x;
  ...
}

if( !(x instanceof A)) {
  ...
} else {
  A y = (A)x;
  ...
}
```

Passing the result of the cast to a method also won't work because of the code generated to set up the method invocation. It is usually acceptable to use the result of the cast as the reference for a method invocation.

```
if( x instanceof A ) {
  foo( (A)x );      // no good
  ...
}

if( x instanceof A ) {
  ((A)x).bar();     // ok
  ...
}
```

The best source construct is this:

```
if( x instanceof A ) {
  A y = (A)x;
  ...
}
```

Then y can be used wherever it is needed.

To wrap together all the above cases into a single example that works:

```
x = fetch();

if( otherResult && x instanceof A ) {
    A y = (A)x;
    foo( y );
    y.bar();
    ...
} else {
    ...
}
```

Making smaller .cod files

- Declare members as *private* whenever possible. This allows the compiler to omit the name from the .cod file.
- Don't declare members protected unless you really think somebody is going to want to override them. Protected member names end up in the .cod file.
- Prefer *package* access over *public* access (*package* access is used when you omit the *private*, *public* and *protected* keywords).
- A class that will never be extended should be marked as *final*. The presence of the *final* keyword allows the compiler to generate code that invokes methods directly, without using the virtual method table.

  This has two benefits:

  1. Direct invocation is faster. There is no indirect lookup required to find the new code offset.
  2. Direct invocation, within a .cod file, is smaller. It requires no fix up information to be generated for the invocation. This makes for smaller .cod files. The fix up information needs 10 bytes for the first invocation and 2 more bytes for each additional invocation.

Avoid Object.getClass()

Avoid this method. It is only included for TCK compatibility and is **not** efficient.

Avoid Vector.ensureCapacity()

Avoid this method. It is only included for TCK compatibility. If called, that vector will stop using the *Array.resize()* function.

Loop induction variables

Do not rely on javac to do loop optimizations. Always attempt to do your own induction variable optimizations.

Avoid this:

```
// calculates 2*I+1 every loop iteration
for( int i = 0; i < n; ++i )
    x[i] = x[2*i+1];
```

Instead, use:

```
// introduce a new variable j==2*I+1
for( int i=0, j=1; i < n; ++i, j += 2 )
    x[i] = x[j]
```

Common sub-expressions

If you ever use the same expression twice, don't rely on javac to optimize it for you. Use a local.

For example, avoid this:

```
foo( i+1 ); bar( i+1 );
```

Instead use:

```
int tmp = i+1; foo(tmp); bar(tmp);
```

Use int not long

A Java *long* is a 64-bit integer (unlike a C++ *long* which is 32-bits). Since Java-based BlackBerry handhelds use 32-bit processors, things will go 2 to 4x faster if you use an *int* instead of a *long* wherever possible.

Avoid using Strings as keys or GUIDs

While classes like *ResourceBundle* and *Hashtable* may encourage using *String* keys in Big Java, this practice is entirely inappropriate in a wireless environment.

Unlike C++, where a literal string is just another constant, a *String* in Java is a first class object, which must be instantiated. Each time a “literal” or a *static final String* constant is referenced, it creates a garbage object. Non-final *static Strings* must be instantiated on a per-process basis.

You would not use *malloc()* to create constant values in C++, and you should not do it in Java. Integer constants should always be used in place of *Strings* when possible. With a little bit of discipline, integer keys can be just as flexible and upwardly compatible as *String* keys.

C++ Bits & Pieces

Efficiently Timing Out

In a complex protocol where many events occur in rapid sequence (such as bytes coming in through the serial port), there may be a need to time out when there is a gap of duration M milliseconds between bytes. The typical approach is to reset the countdown timer for every byte to M milliseconds into the future. Of course, this approach can be very inefficient.

A better way is to set the countdown timer M milliseconds into the future, and simply let it run to expiration. When it expires, check if the maximum of M milliseconds has elasped since the last byte. If not, reset the timer for only the remaining allowable period. The trick is to keep track of when the last byte was received. So do something like this:

At start:

```
// Set the initial timeout timer.
SetTimer(M)
```

Received character:

```
// Make note of when the character was received.
LastRxByteTime = ReadTime()
```

On timer expiry:

```
if (LastRxByteTime - ReadTime() > M){
  // It has been interval M. Timeout applies.
  do timeout handling
}else{
  // If no further bytes are received, set timer
  // to expire when the timeout would occur.
  SetTimer(M - (LastRxByteTime - ReadTime())
}
```

Random Numbers

Random number can be generated by using the srand() and rand() functions with the BlackBerry TIME structure. The code fragment below details how to create random numbers:

```
#include "pager.h"
#include "ribbon.h"

// Initialization
int AppStackSize = 2048;
char VersionPtr[] = "RandomNumbers";
// Main entry point
void PagerMain(void) {
  MESSAGE msg;
// The TIME structure stores the local
// time of the device.
  TIME t;

  // Register Application with Ribbon
  RibbonRegisterApplication
    (VersionPtr, NULL, 0, 20);

  // Grab the time every time the application
  // is loaded on the device.
  RimGetDateTime(&t);

  // Create a unique seed every time the
  // application is loaded on the device.
  srand(t.minute+t.hour*60+t.date*60*24);

  //Message loop
  for (;;) {
    // Create a new secret code for a game.
    // 'a_SecretCode' is an array of 4 structs
    // 'code' is an int variable in a struct that
    // can only take on the values from 0 to 5.
    for (int i = 0; i < 4; ++i)
      a_SecretCode[i].code = rand()%6;

    RimGetMessage(&msg);
    //your code;
  }
}
```

The following code fragment uses srand() to seed rand() with TIME then shuffles a deck of cards:

```
//Randomize the seed
TIME t;
int counter     = 0;
int MAX_SHUFFLE = 10;

RimGetDateTime(&t);
srand(t.minute+t.hour*60+t.date*60*24);

while ( counter < MAX_SHUFFLE ) {
  //Local Variables used in shuffling the deck
  unsigned short int randNum1=rand()%52;
  unsigned short int randNum2=rand()%52;

  //Swap two random cards in the deck
  //MAX_SHUFFLE times to shuffle the deck
  temp_storage   = deck[randNum1];
  deck[randNum1] = deck[randNum2];
  deck[randNum2] = temp_storage;

  //Increment counter
  counter++;
}
```

Storing Data Persistently

Richard Evers, Editor

Persistently trying to resolve issues, persisting to an old age, and the persistent odor of a light perfume. These can all be good things. Committing facts to memory, writing a shopping list, taking notes in class, sending email, saving documents to disk, and storing information within a database are all good examples of persistence.

In the wireless world of BlackBerry development, persistence can be found when sending and receiving email; making browser requests; pushing and pulling data; syncing and adjusting calendar, task list, contacts, and notepad data; installing applications over the air; and storing data programmatically.

In this article, we will address the programmatic storage of data on the BlackBerry handheld.

Considering the facts

Creating an application for the BlackBerry handheld requires a developer to consider many factors not present in a conventional system, such as:

- Persistent store objects live in a finite amount of flash memory that is shared with the operating system, BlackBerry applications, third-party applications, and user data.
- The data transfer rates to and from flash memory is slower than conventional memory (RAM).
- Data transmission speeds across wireless networks are far lower than across wireline networks.
- Reading and writing to flash memory forces the CPU to work harder, which in turn compromises battery life.
- Physical representations of persistent data are restricted to byte arrays for MIDlets (and Java objects when writing specifically for the BlackBerry API).

Two ways to do it

There are two ways to programmatically store data on a Java-based BlackBerry handheld:

1. MIDP record stores
2. The BlackBerry Persistence API set

MIDP record stores should be used when writing applications that will work on all J2ME-enabled devices. MIDP record stores have a lot of methods available to manipulate them, and are relatively easy to work with.

The most useful feature is the ability to read, insert, delete, and update individual records within a store of records without having to load the entire store into memory first. The down side is that you can only store byte arrays, cannot retain more than 64 KB of data within a record store, and automatically provide data visibility to MIDlets within the same MIDlet suite.

The signed BlackBerry persistence model can be used to develop applications that take full advantage of the BlackBerry architecture. Size restrictions are largely removed, data visibility is set by the application, and objects (including custom objects) can be saved to persistent store. Persistently stored data can also be backed up and restored through the use of the signed synchronization API in the *net.rim.device.api*.synchronization package.

Note that *PersistentStore* is a lightweight database solution. You can serialize an object of any type to persistent store, but cannot selectively update or locate elements. You must load the entire object into memory, alter as required, and then commit the entire object back to persistent store.

Code signing

The BlackBerry persistence model requires the use of signed APIs. For more information, please review Jonathan Nobels's article, **Give Me A Sign**, which has been published in this issue.

BlackBerry PersistentStore & PersistentObject

Serialization of data is straightforward with the BlackBerry API.

Create/Open

Call *PersistentStore.getPersistentObject(long_key)* to retrieve a reference to an existing *PersistentObject*, or to create a new one if it does not exist. Use a static constructor so that only one *PersistentObject* is created the first time that an object of this class is created. Each time a process starts, the static block will be run again.

```
public class MyStore implements KeyListener,
TrackwheelListener {
  private static PersistentObject store;

  static {
    // key hash of com.rim.bbdj.mystore
    store = PersistentStore.
      getPersistentObject(0xa406067aeb8ca6ebL);
  }
```

Note that “long_ key” must be a unique long value. The easiest way to create a unique value is to create a hash of your fully qualified package name. If you use more than one persistent store object within your application, append a descriptive table identifier to your package name before deriving the hash. The steps required to create a hash value from within the BlackBerry IDE are:

1. Type a string value, such as com.rim.bbdj.mystore
2. Highlight the entire string
3. Right-click then click “Convert 'com.rim.bbdj.mystore' to long”

The long value appears (0xa406067aeb8ca6ebL). Make sure to include a comment in your code to indicate the string used to generate the long key.

Store

Use *PersistentObject.setContents(Object obj)* then *PersistentObject.commit()* within a synchronized block to store data.

```
private MenuItem saveItem = new MenuItem
  ("Save", 110, 10) {
  public void run() {
    String username = new String("Tiberius");
    String password = new String("IGrokSpock");
    String[] userinfo = {username, password};

    synchronized(store) {
      store.setContents(userinfo);
      store.commit();
    }
  }
};
```

Retrieve

Use *PersistentObject.getContents()* within a synchronized block to retrieve data.

```
private MenuItem getItem = new MenuItem
  ("GetItem", 110, 11) {
  public void run() {
    synchronized(store) {
      if(store.getContents() == null)
        System.out.println("Error");
      else
        String[] currentinfo =
          (String[])store.getContents();
    }
  }
};
```

Delete

Use *PersistentStore.destroyPersistentObject(long id)* to delete the database.

```
PersistentStore.destroyPersistentObject
  (0xa406067aeb8ca6ebL);
```

Custom Persistent Objects

Custom objects can be stored persistently. The process is basically the same when creating/opening, storing, retrieving and deleting a persistent store object.

There are two main differences. First, you need to work with anything that extends *Object*. Second, the class of the object to be saved must implement the *Persistable* interface.

The main differences in sequence:

1. Create a Vector object to store multiple objects

```
private static Vector data;
```

2. Create a PersistentObject database

```
private static PersistentObject store;
```

3. Initialize the database to store a Vector

```
static {
  store = PersistentStore.
    getPersistentObject(0xa406067aeb8ca6ebL);

  synchronized (store) {
    if (store.getContents() == null) {
      store.setContents(new Vector());
      store.commit();
    }
  }
  data = new Vector();
  data = (Vector)store.getContents();
}
```

4. Create a Persistable class to handle the Vector data

```
private static final class RestaurantInfo
implements Persistable {
  //data
  private Vector elements;

  //fields
  public static final int NAME = 0;
  public static final int ADDRESS = 1;
  public static final int PHONE  = 2;
  public static final int SPECIALTY = 3;

  public RestaurantInfo() {
    elements = new Vector(4);

    for ( int i=0; i<elements.capacity(); ++i)
      elements.addElement(new String(""));
  }

  public String getElement(int id) {
    return (String)elements.elementAt(id);
  }

  public void setElement(int id, String value) {
    elements.setElementAt(value, id);
  }
}
```

MIDP Record Stores

MIDP Record Management System (RMS) provides developers with the ability to create persistent store databases, then selectively add, update and delete individual rows of byte array data within the database.

Multiple MIDlets within a MIDlet suite, or threads within a MIDlet, can open databases, and read/write records within a persistent store database. This can be performed without clash because all record store operations are atomic, synchronous and serialized. To ensure data integrity, locking of the entire *RecordStore* is employed when reading/writing individual records within the store. While table locking is unacceptable with conventional relational database systems, it was deemed acceptable on wireless devices as the overhead for memory/storage and processing to maintain row-level locks would be far too excessive.

MIDP relies on the *javax.microedition.rms.RecordStore* class and four interfaces for most operations:

1. RecordComparator
2. RecordEnumerator
3. RecordFilter
4. RecordListener

Each MIDlet suite has its own separate name space for record stores. This means that MIDlets can access any record store within their MIDlet suite.

Create/Open

Create a new *RecordStore* database by calling *RecordStore.openRecordStore()*, passing the name of the database, and a Boolean flag set to 'true' to create the database if it doesn't exist.

```
String name = "myDb";
RecordStore rs = RecordStore.openRecordStore
  ( name, true );
```

Best practice is to iterate through all *RecordStores* in the MIDlet suite to determine if a naming clash exists before creating or opening a database. This can be accomplished as follows:

```
String [] stores = RecordStore.
  listRecordStores();

if ( stores!= null ) {
  for ( int element = 0;
    element < stores.length;
    element++ ) {
    if ( stores[element].equals(name)) {
      // name is already in use
    }
  }
}
```

It is best to check before creating a new database for the following reasons:

- Any MIDlet within a MIDlet suite can create databases that are shared within the suite
- *RecordStore* names are limited to 32 Unicode case-sensitive characters
- No guidelines are in place to construct filenames

Store

There are two ways to store record data.

The first method adds a new record to a *RecordStore* database:

```
// addRecord(byte[], byte[]);
int recordId = rs.addRecord
  (data, 0, data.length);
```

Pass a *byte []* buffer populated with the data to write, the offset into the buffer where the data starts, and the number of bytes to write. The example uses *data.length*, which assumes that the full buffer is being used. This method returns the sequential record identifier assigned to the new record.

The second method updates an existing record within a *RecordStore* database:

```
// setRecord( int, byte[], int, int );
rs.setRecord(recordId, data, 0, data.length);
```

Pass the record identifier, a *byte []* buffer populated with the data to write, the offset into the buffer where the data starts, and the number of bytes to write.

Retrieve

There are two ways to retrieve a *RecordStore* record.

The easiest is to simply pass the record identifier you want:

```
// getRecord( int );
byte [] data = rs.getRecord(recordId);
```

The second method is a little more powerful because it allows you to selectively append data to the *byte []* buffer:

```
// int getRecord( int, byte[], int );
int bytes_copied = getRecord
  (recordId, data, offset);
```

Pass the record identifier, a buffer to hold the *byte []* data, and the starting offset into the buffer to write data. The number of bytes retrieved into the passed *byte []* buffer (data) is returned.

Delete

Delete an individual record via:

```
// deleteRecord( int );
rs.deleteRecord(recordId);
```

Delete the entire *RecordStore* database via:

```
// deleteRecordStore( String );
rs.deleteRecordStore(recordStoreName);
```

Close

```
if ( rs != null )
  rs.closeRecordStore();
```

Row iteration

RecordStore.enumerateRecords() is used to step through the rows within an active *RecordStore* database. The returned *RecordEnumeration* object can be used to work forward or backward through a record set as defined when first calling *enumerateRecords*. This process can be made to simply walk through all rows without regard to sequence, or can be enhanced to track row changes, filter results or custom-sort underlying rows.

Methods available within the *RecordEnumeration* interface include:

- public byte[] nextRecord();
- public byte[] previousRecord();
- public int nextRecordId();
- public int previousRecordId();
- public boolean hasNextElement();
- public boolean hasPreviousElement();
- public int numRecords();
- public void keepUpdated(boolean keepUpdated);
- public boolean isKeptUpdated();
- public void reset();
- public void rebuild();
- public void destroy();

The following code can be used to access all rows within a *RecordStore* database in sequence. Note that the parameters passed to *enumerateRecords()* disable *RecordFilter* and *RecordComparator*, and also disable automatic updates to the underlying rows if any external changes occur during processing:

```
// open, but not create, database
RecordStore rs = RecordStore.openRecordStore
  ( "myStoreDB", false );

if (rs != null ) {
  RecordEnumeration dataset = null;

  try {
    // parm 1: RecordFilter [ disabled ]
    // parm 2: RecordComparator [ disabled ]
    // parm 3: track changes [ disabled ]
    // RecordEnumeration enumerateRecords
    // (RecordFilter, RecordComparator,
    //  boolean);
    dataset = rs.enumerateRecords
      ( null, null, false );

    while(dataset.hasMoreElements()) {
      int recordId =dataset.getNextRecordId();

      // row identifier is valid
      if ( recordId )
        byte [] data = rs.getRecord(recordId);
    }
  }
  catch( RecordStoreException except ) {
  }
  finally {
    dataset.destroy();
  }
  rs.closeRecordStore();
}
```

***Note:** RecordEnumeration sets can be traversed forward and backward.*

For forward navigation, use *RecordEnumeration. getNextRecordId()* in combination with *RecordStore. getNextRecord()* to get the next row, and *RecordEnumeration.hasNextElement()* to determine if additional rows are available.

For backward navigation, use *RecordEnumeration. getPreviousRecordId* and *RecordStore.getPreviousRecord()* to get the previous row, and *RecordEnumeration. hasPreviousElement ()* to determine if additional rows are available.

Call *RecordEnumeration.reset()* at any time to change the direction, and reset access at the start.

RecordStore.enumerateRecords Parameters

Parameter 1: RecordFilter

Passing a *RecordFilter* object to *enumerateRecords()* will pass the *byte []* data within all processed rows to the *matches()* method of the *RecordFilter* object before allowing processing to occur.

The *RecordFilter* interface contains a single method as shown below:

```
public interface RecordFilter {
  public boolean matches( byte[] recordData );
}
```

A sample implementation could work as follows:

```
dataset = rs.enumerateRecords
  ( new DSFilter(), null, false );

...

public class DSFilter implements RecordFilter{
  public boolean matches( byte[] recordData ){
    boolean ret = false;
```

```
    if (recordData[0] != 0
    && recordData.length > 0)
      ret = true;

    return ret;
  }
}
```

Parameter 2: RecordComparator

Pass a *RecordComparator* object to *enumerateRecords()* to sort rows. It contains a single method as shown below:

```
public interface RecordComparator {
  public int compare(byte[] rec1,byte[] rec2);

  public int EQUIVALENT =  0;
  public int FOLLOWS    =  1;
  public int PRECEDES   = -1;
}
```

A sample implementation could work as follows:

```
dataset = rs.enumerateRecords
  ( null, new DSCompare(),false );

...

public class DSCompare implements
RecordComparator {
  public int compare(byte[] rec1, byte[] rec2){
    String srec1 = new String(rec1);
    String srec2 = new String(rec2);

    return (srec1.compareTo(srec2));
  }
}
```

Note that there will be a speed penalty when using *RecordComparator*, especially when the third parameter, keepUpdate, is set to true. In this situation, the entire data set will automatically be resorted after the *RecordEnumeration* index has been rebuilt.

Parameter 3: keepUpdated

The third parameter to *enumerateRecords()* should be set to ‘true’ only if there is a risk that some rows will be externally changed, added or deleted during navigation. In this situation, the *RecordEnumeration* will become a listener of the *RecordStore* and react to record additions and deletions by recreating its internal index. Do not use this feature unless absolutely warranted, as there will be a performance hit every time the index is rebuilt due to a change to the *RecordStore*.

Summary

If you do not require the cross-platform capability offered by MIDP record stores, use the BlackBerry persistence API set to take full advantage of the BlackBerry handheld architecture.

FoNts

Michael Clewley, Editor

Version 3.7 of the Java Development Environment (JDE), for color BlackBerry handhelds, introduces many enhancements to the user interface (UI).

Among the enhancements is the ability to specify fonts for your applications. The current release of the JDE supports three types of fonts:

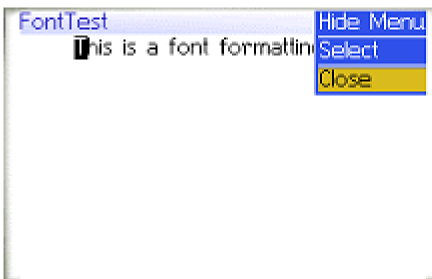
1. Millbank
2. Millbank Tall
3. System

As a developer, you can programmatically specify a default font to be used by your application. To accomplish this, we use a combination of methods from the *FontFamily* class and the *Font* class, both of which are included in the *net.rim.device.api.ui* package.

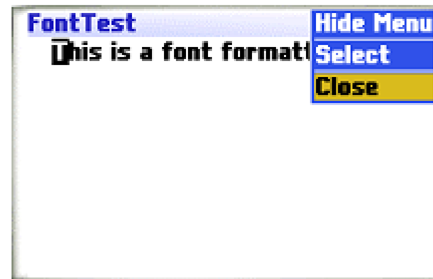
```
// Get a list of the system font families
// in the simulator this will return Millbank,
// Millbank Tall, and System.
FontFamily fontFamily[] =
  FontFamily.getFontFamilies();

// Create a font instance using the first
// element from the font family listing.
// In this case it will be Millbank
//
// With the getFont() method the style and
// font height must alsobe specified.
Font font = fontFamily[0].getFont
  (FontFamily.SCALABLE_FONT,16);

// Set the default font to be used in the
// application. Now all text will be display
// using this specific font, including
// fields and menus.
Font.setDefaultFont(font);
```



Before the change in default font



After the change in default font

You can now create an application with a specific font type and you will not have to be concerned about the formatting of your text and fields due to user-specific font settings.

As an alternative to working with fonts, the RichTextField offers the ability to format text to a specific font. There are two constructors for RichTextField that will accept a Font parameter.

```
RichTextField( String text, int[] offsets,
  byte[] attributes, Font[] fonts, long style)
RichTextField( String text, int[] offsets,
  byte[] attributes, Font[] fonts,
  Object[] cookies, long style)
```

Using similar code from the previous example, we will create a simple rich text field and modify the text in that field to be formatted for our specific font.

```
//The RichTextField constructor will only accept
//an array of fonts. For this example, it will
//be kept simple and just create a font array
//with one element.
Font fonts[] = new Font[1];

//Assign the font (Millbank) to the first
//element of the font list.
fonts[0] = font;

String text = "This is a font formatting test";

//A formatted rich text field must contain a
//list of offsets which define the boundaries or
//the formatting regions. The first offset marks
//the beginning of the field's text (normally
//0), and the last offset in the list marks the
//end of the field's text.
int offset[] = new int[2];
offset[0] = 0;
offset[1] = text.length();
```

```
//An attribute list must also be passed into the
//constructor. This is related to the regions
//described in the offset list. The attribute
//list will have one less element than the
//offset list. Each element's value corresponds
//to an index in the fonts list.
byte attribute[] = new byte[1];
attribute[0] = 0;

RichTextField richTextField =
  new RichTextField(text, offset, attribute,
  fonts, RichTextField.TEXT_ALIGN_HCENTER);
```

Note: *It is also acceptable to pass a null value into the offset and attribute values. This will format the entire rich text field using the font in the first element of the font array.*

As seen in the image above, only the font in the rich text field has been modified and the title and menu remain as the current system default font.

Note: *Please remember that these font classes are only included with the 3.7 APIs, and will cause errors when the same application is run on a BlackBerry running version 3.6 or earlier. Be sure to code around the font classes if running your application on version 3.6.*

C++ Edit Field in Over-ride

Bill Foust

Nearly every developer has faced a situation where the standard field types are simply not enough because they do not allow you to develop the application in the way the user needs. Even simple User Interface compromises can contribute to an application which some users will find confusing or non-intuitive. When it comes to application development, I subscribe to the theory of least surprises - do what the user will find the least surprising.

I know I'm not the only developer to ask “Is there any way that I can make the edit field do this in a slightly different way?” The *Edit* field is a tremendously powerful tool supplied as part of the base SDK. The WIN32 API doesn't have anything as powerful and it's for a desktop! Still, it can't be used to fit every need. Fortunately we can implement our own custom fields based on the *Edit* field through sub-classing and overriding, an important undocumented function.

The *Edit* field has many undocumented functions that we won't need to discuss at this point. The important one for this technique is the HandleInput function. As its name implies, this function is responsible for handling all user input. Its one parameter is a *MESSAGE* structure, so its pretty clear that the *UIEngine* object somehow passes input events such as key down events to this control using this *HandleInput* function. Therefore, if we override this function and intercept these incoming events, it is possible to exert considerable control over how this field behaves.

Let's use a realistic example to start with. Let's say you want to input a serial number for some production facility. Assume that this serial number follows a pattern of three upper case letters followed by some numbers. The numeric portion of the serial number may be variable length, but there are always three letters in the beginning and then only numbers.

Obviously, the best you can do with a standard *Edit* field is to apply the *ALPHANUMERIC* flag to restrict input to only letters and numbers. However, it is still very easy for someone to enter an invalid serial number. Only by checking the value after input can you inform the user they have made a mistake.

If you think about how the *Edit* field *HandleInput* function is likely to be implemented, it probably looks something like this:

```
if(CharacterIsAllowedBasedOnProperties())
{
    AddCharacterToField();
}
```

Since we don't have any control over the properties, we will simply have to do the same basic logic in our special *HandleInput* function. If you want to allow the character, then pass the message onto the standard *Edit* field *HandleInput* function. If not, then do nothing!

```
if (CharacterIsAllowed())
{
  Edit::HandleInput();
}
```

So now, lets go back to the serial number example we started with and create a new field called *SerialNumber.*

```
class SerialNumber : public Edit
{
  SerialNumber(){} // Constructor
  RESULT HandleInput(MESSAGE const &);
};

RESULT SerialNumber::HandleInput
  (MESSAGE const & msg)
{
  return Edit::HandleInput(msg);
}
```

This code implements a custom control that behaved exactly like the standard *Edit* field, after all, it simply calls the default *HandleInput* function. Now, however, we can begin to implement the custom logic needed for this field. Let's begin with forcing any letters to be upper case.

We can't simply modify the parameter that is passed in. It is a const, which means that it will not be changed by this function. Instead, we have to create a local copy of the *MESSAGE* struct, and then modify and use it when calling the base class HandleInput function.

```
RESULT SerialNumber::HandleInput
  (MESSAGE const & msg)
{
  // Make a local copy first
  MESSAGE NewMsg = msg;
  // We are only interested in KEYPAD events
  if(InputMessage.Device == DEVICE_KEYPAD)
  {
    // Particularly, key down events
    if ( InputMessage.Event == KEY_DOWN ||
      InputMessage.Event == KEY_REPEAT)
    {
      // If the key pressed was lower case
      if (NewMsg.SubMsg >= 'a' &&
        NewMsg.SubMsg <= 'z')
      {
        // Change it to upper.
        NewMsg.SubMsg -= ('a'-'A');
      }
             // Now call HandleInput with the
```

```
            // new value!
            return Edit::HandleInput(NewMsg);
        }
    }
    return Edit::HandleInput(msg);
}
```

So now we have a custom field that turns all of the input into upper case letters. This by itself may be useful on its own, but we aren't done with it yet. The important part though is demonstrating how you can intercept an incoming event and alter it before letting the *Edit* field handle it.

In order to fully implement the SerialNumber field, we need to make a decision about whether the incoming event can be allowed or not by examining the data that is already in the control.

```
RESULT SerialNumber::HandleInput
    (MESSAGE const & msg)
{
    // Make a local copy first
    MESSAGE NewMsg = msg;
    // We are only interested in KEYPAD events
    if(InputMessage.Device == DEVICE_KEYPAD)
    {
        // Particularly, key down events
        if ( InputMessage.Event == KEY_DOWN ||
             InputMessage.Event == KEY_REPEAT)
        {
            // If less then 3 characters
            // in the buffer, this new character
            // must be a letter
            if (GetBufferLength() < 3)
            {
                // If the key pressed was
                // lower case
                if (NewMsg.SubMsg >= 'a' &&
                    NewMsg.SubMsg <= 'z')
                {
                    // Change it to upper.
                    NewMsg.SubMsg -= ('a'-'A');
                }

                if (NewMsg.SubMsg >= 'A' &&
                    NewMsg.SubMsg <= 'Z')
                {
                    // Now call HandleInput with
                    // the new value!
                    Edit::HandleInput(NewMsg);
                }
            }
            else
            {
                if (NewMsg.SubMsg >= '0' &&
                    NewMsg.SubMsg <= '9')
                {
                    Edit::HandleInput(NewMsg);
                }
            }

            return CONTINUE;
        }
    }
    return Edit::HandleInput(msg);
}
```

By checking the length of the data in the buffer we can determine whether the field should accept letters or numbers. If the field is only accepting letters, then only events with characters between the range of 'A' and 'Z' are passed to the base class *HandleInput* function. Any other characters are simply ignored. The same is also if the field is only accepting numbers. Only events with characters between the range of '0' and '9' are passed to the base class *HandleInput* function.

***Note:** Calls to HandleInput no longer check the return value. Instead the function always returns the constant CONTINUE. CONTINUE is a value defined in result.h and indicates that the event was processed successfully and that the field is ready for the next event.*

That's it! The *SerialNumber* class is complete. As you can see, the logic in the *HandleInput* function can be as simple or complex as necessary. Of course, this cannot be used in every situation, but I've found that any situation where an *Edit* field is just OK can be made better by defining a custom field.

Want to make the *SerialNumber* class even better? Alter the function so that it should only be accepting numbers, but when a letter is pressed that shares a key with a number, the corresponding number is displayed. For instance, pressing 'q' will automatically be changed to a '1' thus making it even easier for the user by not having to press the ALT key repeatedly.

Bill Foust is an independent consultant specializing in BlackBerry development. He has written several BlackBerry applications including ExpenseMinder and EzToDo, two of the best-selling BlackBerry applications at http://www.handango.com.

Give Me A Sign

Jonathan Nobels, Research In Motion

Security: Who Needs It?

There is a mantra at Research In Motion (RIM) that goes as follows:

“Never ignore the top three concerns of your corporate customers. In order, these are security, security and finally... security.”

The design of the BlackBerry platform, indeed the architecture of the Java Virtual Machine upon which the BlackBerry Platform is built, owes much of its existence to this mantra.

Security is a broad term affecting many aspects of any wireless solution. To an enterprise customer, security could mean limiting access to confidential information and systems. To a network operator, security often means guaranteeing the integrity of their networks. To an end user, security typically means that personal information cannot be compromised, applications do what they are expected to do, and the user is only billed for things they have explicitly agreed to pay for.

In designing the BlackBerry platform, RIM recognized that in many instances, these visions of security conflicted. Many applications, by their very nature, require unfettered access to personal data or network resources. At the same time, access to these resources creates significant security concerns with the various stakeholders. Power users may not want to be hassled for permission every time an application wishes to open a network connection, and enterprise developers may not wish to submit their applications to scrutiny by a third party.

To address their security concerns, some network operators are choosing to lock down their networks to only specific “network-certified” applications. MIDP 1.0 addresses security by severely limiting the APIs available to the developer, but at the same time severely limiting usefulness of the platform. Many other operating systems have chosen to use the PC model, forgoing most security controls and putting the responsibility for handling security on the user. This latter approach can create a support problem for wireless network operators.

RIM's Controlled API Java architecture was created as a compromise between these extremes. Recognizing that developers require flexibility, and that IT managers and users will demand to be masters of their own destiny, RIM implemented the notion of "controlled APIs" with BlackBerry 3.6.

BlackBerry Code Signing 101

The BlackBerry API set is subdivided into five distinct classifications.

The first of these classifications is the set of “Open” APIs. These APIs include all standard Java APIs from MIDP and CLDC, as well as many BlackBerry-specific APIs. The open API set is, as the name implies, open for all developers. Applications developed that use only open APIs require no signatures of any sort.

The remaining classifications are all “Controlled” APIs. These APIs fall into four categories pertaining to their specific use:

1. RIM Runtime APIs
2. BlackBerry APIs
3. RIM Cryptography
4. Certicom Cryptography

There is a security risk associated with every class and method in these APIs. For example, the BlackBerry API allows a developer to access the user's Personal Information Manager (PIM) data, which may be a security risk to the end user. The RIM Runtime API allows the developer to send SMS messages, a potential security risk to the network operator and the end user.

To help control these risks, any application that uses a Controlled API will not run on a BlackBerry handheld unless the application is properly signed.

How Do Developers Sign Applications?

Prior to granting access to any of the Controlled APIs, RIM requires the developer to go through a simple registration process during which the developer's identity is verified. The developer needs to download and fill out a registration form from:

```
http://na.blackberry.com/eng/developers/
  downloads/api.jsp
```

... then fax it to RIM. Once the registration form and associated processing fee has been received, RIM will send the developer a set of keys via email. These keys can be used in conjunction with the Signature Tool utility in the BlackBerry JDE to sign any application.

The signing process itself is very straightforward. After compilation, the developer manually starts the Signature Tool , which displays a list of necessary signatures. With a single button press, the Signature Tool submits a hash of the application to RIM's signing authority. The signing authority automatically returns the required signature, which is automatically appended to the application. The application can then be loaded onto a handheld.

Detailed instructions related to installing keys and running the signature tool can be found in the BlackBerry Java Developer Guide, Volume 2 - Advanced Topics.

A Few Things to Remember...

In order to run applications that use controlled APIs on a handheld, you must first get it signed by RIM. At no point in the signing process does RIM receive a copy of your application, only a hash of the file(s). The application itself is not reviewed by RIM and the signing is completely automated. Should an application later turn out to have malicious intent, RIM will be able to determine the author of the application by matching the hash of the application against records of the hash kept by RIM signing authorities.

As a developer, your key is your identity. Do not give your keys to others or let others use your keys to sign their applications.

When you register for your keys, you will get three emails from RIM, one from each of the various signing authorities (with the exception of Certicom Crypto). Follow the instructions in the BlackBerry Developer Guide carefully. Also, when you request your keys, you provide RIM with a PIN number. This PIN is used in conjunction with the keys when first contacting the signing authority to ensure that you, and only you, can use the keys that you have been assigned.

To receive a key for the Certicom Cryptography APIs, RIM requires confirmation from Certicom Corp that you have a license for Certicom's Security Builder Toolkit v3.0.

Keep in mind that you do not need to register with RIM to use the JDE or to simulate applications. Applications do not need to be signed to run on the BlackBerry simulator.

Why a $100 fee?

A $100 fee is charged to sign applications. This fee is only charged once to receive a set of keys. There is no fee for each subsequent use of thek eys.

There are two reasons for the fee. First, a legitimate credit card account and a successful transaction help us verify the identity of the registrants. Second, it helps us offset some of the costs of running the program. The expectation is that this small fee is not a burden to legitimate developers, but creates a moderate disincentive for any developer with malicious intent.

So Where is the Benefit for Developers?

RIM has made every attempt to ensure the code signing process presents as minimal a barrier as possible to legitimate developers. While the initial request for keys requires some paperwork and a small administration fee, from that point on the developer has full access to any of the APIs they have registered to use.

In building this system, we hope we have achieved a balance that gives developers the freedom they require in terms of development and deployment while imposing legitimate and useful checks and balances to ensure that all other stakeholders are comfortable with the BlackBerry platform.

Let me introduce you to ...

Peter Hantzakos

The Plazmic Content Developer's Kit 3.7 for BlackBerry (CDK) is a set of utilities that allow users to create cool, colorful, interactive media content for the BlackBerry 3.7 platform. The CDK is a content development platform.

Now, lets explain what the CDK is not:

The CDK is **not** a software development environment.

But wait! Don't stop reading just yet ...

CDK + JDE = Cool BlackBerry Fun!

You can combine the use of the CDK with the use of the Java Development Environment to create a rich more compelling user experience for your BlackBerry handheld clients, more compelling than simple text alone.

How?

The JDE includes Media Engine libraries that can view the content created in the CDK. So, create some content using the CDK then develop an application with the Media Engine libraries using the JDE. Combine the two to create a rich experience for BlackBerry handheld users. You can also use the CDK on its own to create fun, interactive sites that are viewable on the BlackBerry handheld browser.

The pieces of the content puzzle

The CDK includes various components. One component, Composer, allows creation of content within a GUI-based environment and features shape tools, palettes, an animation timeline, layers, interactivity set up, and content simulation.

Hand code your SVG if you are technically inclined and don't like using GUI-based tools. If you do, refer to the BlackBerry Wireless Handheld SVG Reference Documentation that is shipped with the CDK. It lists all SVG elements and attributes supported in the CDK. We are however, strict about compliance to our DTD, the Document Type Description that is used to validate the SVG in your document against what we support in SVG. If there are errors or omissions in the hand coded SVG, the Media Engine Simulator will surely find them, and let you know about it too, through its logging window.

Another piece of the content puzzle

The Media Engine Simulator is another useful component of the CDK. Launched on its own or from within Composer, it is a useful tool for debugging and previewing content before deploying it to a web server for viewing on the BlackBerry handheld browser. It features handheld profiles, logging window, scene properties information, and play, pause, and restart content playback actions.

The SVG transcoding utility is also included with the CDK. It transcodes supported SVG it into a format that the BlackBerry handheld browser understands, called '.pmb' or 'pme.' Why this extra step? Well, .pme/.pmb is more compact and transmits quickly over wireless networks.

Pme can be exported directly from Composer, or can be created using the command line SVG transcoding utility. For more information, refer to the **Plazmic Content Developer's Kit User Guide Documentation**, chapter 5 “**Transcoding SVG Content**”.

The final piece of the content puzzle

The CDK includes an SWF to SVG conversion utility. Note that the swf conversion utility does not support the entire swf format. The utility only provides a bridge to our supported set of SVG elements and attributes. Be sure to familiarize yourself with the SWF conversion utility before creating swf content for BlackBerry. Refer to chapter 3 of the **Plazmic Content Developer's Kit User Guide Documentation**, “**Converting Flash to Scalable Vector Graphics**,” for more information on the SWF Conversion Utility.

The CDK in action

The CDK can be used as a presentation layer. For example, some of our partners have used the CDK to spruce up their intranet sites viewable on the BlackBerry handheld browser. Combining the use of html, and the SVG subset supported in the CDK, they have created fun, interactive, and colorful intranet sites viewable on the BlackBerry handheld browser. The CDK can also be used to create colorful interactive product demonstrations or presentations complete with animation and events based on user input.

We recently used the CDK to create a demonstration WES site where users could view information about the show and sponsors, with an interactive floor section that displays a sponsor's booth when the user scrolled over a booth 'hotspot'.

A live example of the CDK in action is the browser demo. Use your BlackBerry browser to visit:

http://www.blackberry.com/browserdemo

Although the CDK is geared towards those who create, design and develop content, that shouldn't stop you from going in and having some fun and experimenting with the CDK. After all, whether you're a software programmer, a web developer, a graphic designer or content creator, we are all artists of our own craft!

We've mentioned some possible uses of the CDK above. Now, let your imagination run wireless, and see where the CDK will take you. For further information, and to download the CDK, visit the Plazmic website at:

http://www.plazmic.com

Gaming 101 - My First Experience

Mark Sohm, Research In Motion

I recently decided to create a game for the BlackBerry handheld. I figured this would be an opportunity to not only further my knowledge in some areas of the BlackBerry APIs, but also to have some fun. As I sat down and started to plan out what I wanted to create and how I was going to do so, I realized that the design issues for a game can be quite different than an application. In an application the look and feel, interaction and general layout is for the most part pre-determined by what information you are presenting to the user. In a gaming environment, screen design and the way a player can interact with the game can be highly customized. Controls and layout must be intuitive so a player does not get frustrated trying to figure out how to play, instead of enjoying actually playing the game. Taking these thoughts into consideration, it was time to make some design decisions.

The Screen

The first item on my design list was the screen. Since I wanted animation and custom graphics, using a screen manager and populating it with fields was not suitable. Allowing the game to scale across different BlackBerry handheld models with different resolutions and color or monochrome displays was also important. To accommodate this, I had to allow of all of the objects to correctly scale in size. The simplest method I found to designing my screen layout was to draw my screen by hand on a piece of graph paper. Since I wasn't dealing with any actual coordinates, all I had to do was to draw everything based on a percentage of the screen resolution. This would allow my game to be portable across multiple models.

After completing my layout design, I was ready to start coding the class that would draw the screen. First I had to retrieve the screen resolution, which is used as a base for all future calculations.

```
private int screenWidth =
  Graphics.getScreenWidth();
private int screenHeight =
  Graphics.getScreenHeight() - 20;
```

Removing 20 pixels off of the height of the screen was necessary because my game requires some blank space at the bottom of the screen. This shrinks my main height so that I can set all of the objects to take up 100% of the specified screen space. Now I am ready to extend the *MainScreen* class and override the paint method, which is called when the screen is drawn.

```
private class TumblerScreen extends MainScreen
{
  public void start()
  {
  }
  //draw objects to the screen here
  public void paint(Graphics graphics)
  {
    //Draw the 6 drop slots
    int fieldStart = screenWidth / 6;
    int fieldEnd = screenWidth / 18;

    for (count = 0; count < 6; ++count)
    {
      graphics.drawRect((fieldStart) * count, 0,
        fieldStart, fieldEnd);
    }
... snip ...
    }
}
```

The first part of my screen consists of six rectangles that I am setting to be one-sixth of the screen width wide and half of that in height. This allows them to properly scale to fit any size of BlackBerry handheld screen. For now, I can preview what it looks like by creating an instance of my TumblerScreen in the main class.

```
public class Tumbler
{
  private TumblerScreen tumblerScreen;

  public static void main(String[] args)
  {
    Tumbler theApp = new Tumbler();
    theApp.enterEventDispatcher();
  }

  public Tumbler()
  {
    pushScreen(tumblerScreen);
  }
}
```

Controls

Now that I had the screen drawn, it was time to implement a way for my game to receive input from the player. A ball will appear in one of the top drop slots that were drawn using the drawRect method in the paint method above. What I wanted was to have the ball move left or right when the user scrolls the trackwheel. I also needed input for the user to drop the ball from its current drop slot. To do this, trackwheel, Enter, and Space events are captured by implementing TrackwheelListener and KeyListener, overriding trackwheelClick, trackwheelRoll and keyChar.

When a player scrolls the trackwheel up, the ball will move a column to the left. When they scroll down, the ball will move a column to the right. The currTopBallColumn is used in the paint method determining the drop slot to draw the

ball. When the game is in an animation state the user controls are disabled by checking a boolean flag triggered before starting the animation and reset after it has finished.

```
// Invoked when the trackwheel is rolled.
public boolean trackwheelRoll
  (int amount, int status, int time)
{
  //User controls are disabled if the game
  //is animating.
  if (!currentlyAnimating)
  {
    if (amount < 0)
    {
      if (currTopBallColumn == 0)
        currTopBallColumn = 5;
      else
        --currTopBallColumn;
    }
    else
    {
      if (currTopBallColumn == 5)
        currTopBallColumn = 0;
      else
        ++currTopBallColumn;
    }
    tumblerScreen.invalidate();
  }
  return true;
}

// Invoked when the trackwheel is clicked
public boolean trackwheelClick
  (int status, int time)
{
  if (!currentlyAnimating)
    gamePlay();

  return true;
}
```

Pressing Enter, Space or clicking on the trackwheel will initiate the round in the game and drop the ball from its current drop slot. This will call the *gamePlay* method that will handle the game calculations. To capture these events, override the *keyChar* method as shown below. As with the *trackwheelRoll* and *trackwheelClick* user input is ignored if the game is currently animating.

```
public boolean keyChar
  (char key, int status, int time)
{
  boolean retval = false;
  //User controls are disabled if the
  //game is animating.
  if (!currentlyAnimating)
  {
    switch (key)
    {
      case Characters.ENTER:
        gamePlay();
        retval = true;
        break;
      case Characters.SPACE:
        gamePlay();
        retval = true;
        break;
    }
  }
  return retval;
}
```

Animation

The next step involved is to create a method that would handle the animation for my game. This method would perform the calculations involved for transitioning or transforming all objects on the screen based on events that occur during the current round. I decided to extend *Thread* for the animation method in case I required the *sleep* command to slow down the rate of animation. It turns out that the game didn't require the animation to be slowed down so the sleep command was not required. However, since my animation was originally set to move one pixel each frame a turn took a painful amount of time on the higher resolution BlackBerry handhelds. To combat this issue I allow the user to select an animation rate from 1 through to 5. These values represent the number of pixels each element will move in a given frame. This allows people to customize the game speed to suit their BlackBerry model.

```
//Handles all animation calculations and
collision detection
private class AnimationThread extends Thread
{
  public void run()
  {
    ...snip...

    //Repaint the screen
    UiApplication.getUiApplication().
      invokeAndWait(new Runnable()
    {
      public void run()
      {
        tumblerScreen.invalidate();
      }
    });

  ...snip...
}
```

The animation method is responsible for a lot of calculations. It calculates the screen coordinates for all moving objects, detects when objects collide (a ball landing on a platform, lever or another ball) and starts or stops an object's animation sequence based on what it may have collided with. After all calculations for the current frame are complete, the animation method invalidates the screen that causes it to be redrawn by calling the *paint* method.

High Scores - Persistent Storage

What would any game be without keeping track of high scores? It gives users some incentive to keep playing to increase their own high scores or knock others off the list. Storing high scores on the BlackBerry required the implementation of the *Persistable* class. The storage requirements are quite low, as I am only storing the top five scores, which mean allocating five *int* and *String* values.

```
public final class TumblerHighScores
  implements Persistable
{
  private String[] highNames = new String[5];
  private int highScores[] = new int[5];

  public TumblerHighScores(String[] names,
    int scores[])
  {
    highNames  = names;
    highScores = scores;
  }

  public String[] getNames()
  {
    return highNames;
  }

  public void setNames(String[] names)
  {
    highNames = names;
  }

  public int[] getScores()
  {
    return highScores;
  }

  public void setScores(int[] scores)
  {
    highScores = scores;
  }
}
```

Code Signing

The *Persistable* class is a secure API that requires signing of your application. For more information, please review Jonathan Nobels's article, **Give Me A Sign**, which has been published in this issue.

Finishing Up

At this point the main portions of my game were complete. I was capturing user input, displaying custom objects on the screen, performing animation, and storing high scores. There were a few other items that I had to create to put the finishing touches on the game, such as the creation of a main menu screen, an instruction screen, and a screen to display the high scores. These screens were more traditional in the sense that they didn't require any customization.

The complete source code for my Tumbler game can be found at the end of the article. Feel free to use any code from it that you find useful. I hope this can be a resource for those who might start developing for the BlackBerry platform. Creating a game is definitely a fun way to learn about the development environment and producing something that other people enjoy can be very rewarding; not to forget of course the fun you can have playing your own game. Just don't make it too addicting that it keeps you from starting your next project!

Complete Source Code for Tumbler

Tumbler.java

```
/**
  * Tumbler
  *
  * Author:  Mark Sohm
  *
  * Changes: 1.0 Initial release
  * 1.1 Bugfix - Prevent user from dropping multiple balls on one by clicking more than once
  * on the Trackwheel.
  * 1.2 Added Features - Added option to control game speed.
  * High scores now support backup and restore via Desktop Manager.
  * User is now prompted to start a new game when the game is over if they did not get a high score.
  * Added image on intro screen and added menu support.
  * Verifies game exit when leaving the game screen.
  *
  */

package com.msohm.tumbler;

import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
import java.util.*;
import net.rim.device.api.synchronization.*;


public class Tumbler extends UiApplication implements DrawStyle, TrackwheelListener, KeyListener
{
  //Member definitions

  private TumblerScreen tumblerScreen;
  private MainScreen introScreen = new MainScreen();
  private TumblerHighScoreScreen highScoreScreen;
  private static TumblerOptions tumblerOptions = new TumblerOptions();
  private static PersistentObject store;  //Permanently store the high scores here.
  private static PersistentObject optionsStore;  //Permanently store Tumbler options.
  private String[] highScoreNames = {"Tumbler", "Tumbler", "Tumbler", "Tumbler", "Tumbler"};
  private int highScoreValues[] = {500, 400, 300, 200, 100};
  private TumblerHighScores tumblerHighScores;
  private final int screenWidth = Graphics.getScreenWidth();
  //Leave 20 pixels free at the bottom of the screen for the drop off area.
  private final int screenHeight = Graphics.getScreenHeight() - 20;
  private int gameSpeed = 1;  //The default number of pixels each frame will change by.
  private int swingBarTop[][] = new int[8][5];  //See swingBar format below.
  private int swingBarShaft[][] = new int[8][5];  //See swingBar format below.
  //Holds the x coordinates for the balls in the top column.
  private int topBallColumn[] = new int[7];
  private final int ballDiameter = screenWidth / 18;  //The diameter of the ball.
//The column width a ball must travel left or right.
  private final int columnWidth = screenWidth / 6;
  //How far a given ball has moved left or right in the current animation stage.
  private int ballMovement[] = new int[9];
  private int currTopBallColumn = 0;  //The current column of the ball in the top drop slot.
  private int movingBall[][] = new int[9][7];  //The balls in the animation area.
  private boolean currentlyAnimating = false;  //Flag for if the animation state of the game.
  //The left position platform coordinates where the ball could land.
  private int leftStopLocation[][] = new int[8][2];
```

```
//The right position platform coordinates where the ball could land.
private int rightStopLocation[][] = new int[8][2];
//The left position shaft coordinates where the ball could trigger a swingbar change.
private int leftTriggerLocation[][] = new int[8][2];
//The right position shaft coordinates where the ball could trigger a swingbar change.
private int rightTriggerLocation[][] = new int[8][2];
//How many pixels the swingbars will move when animated
private final int animationDistance = screenWidth / 6;
private int ballsLeft;  //The number of balls left in the current game.
private int score;  //The score of the current game.
private static String newHighScoreName;  //New name to be entered into the high score list.
//Flag set after the user enteres their name for a high score.
private static boolean newHighScoreReady;

/*  swingBar array format
  0: x start coordinate
  1: y start coordinate
  2: x end coordinate
  3: y end coordinate
  4: state (0 = stable left, 1 = stable right, 2 = transitioning left to right
  3 = transitioning right to left)

  movingBall array format
  0: x coordinate
  1: y coordinate
  2: 0/1 is ball currently on the screen
  3: ball owner (0 = player 1, 1=player 2, 2=computer)
  4: ball motion direction (0 = down, 1 = left, 2 = right)
  5: 0/1 in motion, is the ball currently moving
  6: The current column the ball is in.
*/

static
{
  //Long value = com.msohm.tumbler.Tumbler
  store = PersistentStore.getPersistentObject(0x6086b8131c33cfc2L);
  //Long value = com.msohm.tumbler.TumblerOptions
  optionsStore = PersistentStore.getPersistentObject(0x48d43d190c69ef01L);
}

public static void main(String[] args)
{
  boolean startup = false;

  //Check parameters to see if the application was entered through
  //the alternate application entry point.
  for (int i=0; i<args.length; ++i)
  {
    if (args[i].startsWith("AutoStart"))
    {
      startup = true;
    }
  }

  if (startup)
  {
    //Entered through the alternate application entry point.
    //Enable application for synchronization on startup.
    SerialSyncManager.getInstance().enableSynchronization(new TumblerSyncItem());
  } else
  {
```

```
      Tumbler theApp = new Tumbler();
      theApp.enterEventDispatcher();
    }
  }

  /**
   * The Tumbler constructor.
   */
  public Tumbler()
  {
    //Retrieve high scores or set to default values.
    synchronized (store)
    {
      if (store.getContents() == null)
      {
        tumblerHighScores = new TumblerHighScores(highScoreNames, highScoreValues);
        store.setContents(tumblerHighScores);
        store.commit();
      }
      else
      {
        tumblerHighScores = (TumblerHighScores)store.getContents();
        highScoreNames = tumblerHighScores.getNames();
        highScoreValues = tumblerHighScores.getScores();
      }
    }

    //Retrieve options or set to defaults.

    synchronized (optionsStore)
    {
      if (optionsStore.getContents() == null)
      {
        tumblerOptions = new TumblerOptions(gameSpeed);
        optionsStore.setContents(tumblerOptions);
        optionsStore.commit();
      }
      else
      {
        tumblerOptions = (TumblerOptions)optionsStore.getContents();
      }
    }


    //Setup and draw the intro screen
    MenuItem startGame = new MenuItem("New Game", 10, 10)
    {
      public void run()
      {
        initialize();
        UiApplication.getUiApplication().pushScreen(tumblerScreen);
      }
    };

    //The High Scores menu item.
    MenuItem highScores = new MenuItem("High Scores", 20, 20)
    {
      public void run()
      {
        highScoreScreen = new TumblerHighScoreScreen(highScoreNames, highScoreValues);
        UiApplication.getUiApplication().pushScreen(highScoreScreen);
```

```
      }
    };

    //The Options menu item.
    MenuItem options = new MenuItem("Options", 30, 30)
    {
      public void run()
      {
        TumblerOptionsScreen optionsScreen = new TumblerOptionsScreen(tumblerOptions);
        UiApplication.getUiApplication().pushScreen(optionsScreen);
      }
    };

    //The Instructions menu item.
    MenuItem instructions = new MenuItem("Instructions", 40, 40)
    {
      public void run()
      {
        TumblerInstructionScreen insScreen = new TumblerInstructionScreen();
        UiApplication.getUiApplication().pushScreen(insScreen);
      }
    };

    //Add the custom menu items to the screen.
    introScreen.addMenuItem(startGame);
    introScreen.addMenuItem(highScores);
    introScreen.addMenuItem(options);
    introScreen.addMenuItem(instructions);

    //The main screen background image.
    BitmapField splashField =
      new BitmapField(Bitmap.getBitmapResource("TumblerSplash.png"),
      Field.NON_FOCUSABLE | Field.USE_ALL_WIDTH | Field.FIELD_HCENTER |
      Field.FIELD_VCENTER | Field.USE_ALL_HEIGHT);
    introScreen.add(splashField);

    pushScreen(introScreen);

    //Setup the game screen
    tumblerScreen = new TumblerScreen();
    tumblerScreen.addKeyListener(this);
    tumblerScreen.addTrackwheelListener(this);

  }


    //Initialize/reset the initial values for the gameplay screen
  private void initialize()
  {

    int count, fieldStart, fieldEnd, tempCoord, countb;
    Random leftOrRight = new Random();

    //Initialize the game speed.

    synchronized (optionsStore)
    {
      if (optionsStore.getContents() == null)
      {
        tumblerOptions = new TumblerOptions(gameSpeed);
        optionsStore.setContents(tumblerOptions);
```

```
      optionsStore.commit();
    }
    else
    {
      tumblerOptions = (TumblerOptions)optionsStore.getContents();
    }
  }

  //Get the animation speed.
  gameSpeed = tumblerOptions.getGameSpeed();

  //Initialize swing bar values

  fieldStart = (screenWidth / 18) + (screenHeight / 12);
  fieldEnd = (screenWidth / 18) + (screenHeight / 4);

  //First row of swing bars;

  swingBarShaft[0][0] = screenWidth / 12;
  swingBarShaft[0][1] = fieldStart;
  swingBarShaft[0][2] = screenWidth / 12 * 3;
  swingBarShaft[0][3] = fieldEnd;
  swingBarShaft[0][4] = 0;

  swingBarTop[0][0] = 0;
  swingBarTop[0][1] = fieldStart;
  swingBarTop[0][2] = screenWidth / 6;
  swingBarTop[0][3] = fieldStart;
  swingBarTop[0][4] = 0;

  swingBarShaft[1][0] = screenWidth / 12 * 5;
  swingBarShaft[1][1] = fieldStart;
  swingBarShaft[1][2] = screenWidth / 12 * 7;
  swingBarShaft[1][3] = fieldEnd;
  swingBarShaft[1][4] = 0;

  swingBarTop[1][0] = screenWidth / 3;
  swingBarTop[1][1] = fieldStart;
  swingBarTop[1][2] = screenWidth / 6 * 3;
  swingBarTop[1][3] = fieldStart;
  swingBarTop[1][4] = 0;

  swingBarShaft[2][0] = screenWidth / 12 * 9;
  swingBarShaft[2][1] = fieldStart;
  swingBarShaft[2][2] = screenWidth / 12 * 11;
  swingBarShaft[2][3] = fieldEnd;
  swingBarShaft[2][4] = 0;

  swingBarTop[2][0] = screenWidth / 12 * 8;
  swingBarTop[2][1] = fieldStart;
  swingBarTop[2][2] = screenWidth / 12 * 10;
  swingBarTop[2][3] = fieldStart;
  swingBarTop[2][4] = 0;

  fieldStart = (screenWidth / 18 * 3) + (screenHeight / 12 * 3);
  fieldEnd = (screenWidth / 18 * 3) + (screenHeight / 12 * 5);

  //Second row of swing bars.

  swingBarShaft[3][0] = screenWidth / 12 * 3;
  swingBarShaft[3][1] = fieldStart;
```

```
swingBarShaft[3][2] = screenWidth /12 * 5;
swingBarShaft[3][3] = fieldEnd;
swingBarShaft[3][4] = 0;

swingBarTop[3][0] = screenWidth / 6;
swingBarTop[3][1] = fieldStart;
swingBarTop[3][2] = screenWidth / 3;
swingBarTop[3][3] = fieldStart;
swingBarTop[3][4] = 0;

swingBarShaft[4][0] = screenWidth / 12 * 7;
swingBarShaft[4][1] = fieldStart;
swingBarShaft[4][2] = screenWidth / 12 * 9;
swingBarShaft[4][3] = fieldEnd;
swingBarShaft[4][4] = 0;

swingBarTop[4][0] = screenWidth / 2;
swingBarTop[4][1] = fieldStart;
swingBarTop[4][2] = screenWidth / 12 * 8;
swingBarTop[4][3] = fieldStart;
swingBarTop[4][4] = 0;

fieldStart = (screenWidth / 18 * 5) + (screenHeight / 12 * 5);
fieldEnd = (screenWidth / 18 * 5) + (screenHeight / 12 * 7);

//Third row of swing bars.

swingBarShaft[5][0] = screenWidth / 12;
swingBarShaft[5][1] = fieldStart;
swingBarShaft[5][2] = screenWidth / 12 * 3;
swingBarShaft[5][3] = fieldEnd;
swingBarShaft[5][4] = 0;

swingBarTop[5][0] = 0;
swingBarTop[5][1] = fieldStart;
swingBarTop[5][2] = screenWidth / 6;
swingBarTop[5][3] = fieldStart;
swingBarTop[5][4] = 0;

swingBarShaft[6][0] = screenWidth / 12 * 5;
swingBarShaft[6][1] = fieldStart;
swingBarShaft[6][2] = screenWidth / 12 * 7;
swingBarShaft[6][3] = fieldEnd;
swingBarShaft[6][4] = 0;

swingBarTop[6][0] = screenWidth / 3;
swingBarTop[6][1] = fieldStart;
swingBarTop[6][2] = screenWidth / 6 * 3;
swingBarTop[6][3] = fieldStart;
swingBarTop[6][4] = 0;

swingBarShaft[7][0] = screenWidth / 12 * 9;
swingBarShaft[7][1] = fieldStart;
swingBarShaft[7][2] = screenWidth / 12 * 11;
swingBarShaft[7][3] = fieldEnd;
swingBarShaft[7][4] = 0;

swingBarTop[7][0] = screenWidth / 12 * 8;
swingBarTop[7][1] = fieldStart;
swingBarTop[7][2] = screenWidth / 12 * 10;
swingBarTop[7][3] = fieldStart;
```

```
    swingBarTop[7][4] = 0;

    //End initialize initial swing bar values

    //Randomly set swing bar direction
    for (count = 0; count < 8; ++count)
    {
        if ((Math.abs(leftOrRight.nextInt()) % 2) == 1)
        {
            //switch the x values for the shaft and update state to right stable
            tempCoord = swingBarShaft[count][0];
            swingBarShaft[count][0] = swingBarShaft[count][2];
            swingBarShaft[count][2] = tempCoord;
            swingBarShaft[count][4] = 1;

            //move the top bar right and update state to right stable
            swingBarTop[count][0] = swingBarTop[count][2];
            swingBarTop[count][2] += (screenWidth / 6);
            swingBarTop[count][4] = 1;

        }
    }


    //Initialize top bar ball coordinates
    for (count = 0; count < 6; ++count)
    {
      topBallColumn[count] = (screenWidth / 20) + ((count) * (screenWidth / 6));
    }

    //Initialize all moving ball values to 0
    for (count = 0; count < 9; ++count)
    {
      for(countb = 0; countb < 6; ++countb)
      {
        movingBall[count][countb] = 0;
      }
    }

    //Initialize stop locations
    //First swingbar row stop locations
    fieldStart = screenHeight / 12;

    leftStopLocation[0][0] = topBallColumn[0];
    leftStopLocation[0][1] = fieldStart;
    rightStopLocation[0][0] = topBallColumn[1];
    rightStopLocation[0][1] = fieldStart;
    leftStopLocation[1][0] = topBallColumn[2];
    leftStopLocation[1][1] = fieldStart;
    rightStopLocation[1][0] = topBallColumn[3];
    rightStopLocation[1][1] = fieldStart;
    leftStopLocation[2][0] = topBallColumn[4];
    leftStopLocation[2][1] = fieldStart;
    rightStopLocation[2][0] = topBallColumn[5];
    rightStopLocation[2][1] = fieldStart;

    //Second swingbar row stop locations
    fieldStart = (screenWidth / 18 * 2) + (screenHeight / 12 * 3);

    leftStopLocation[3][0] = topBallColumn[1];
    leftStopLocation[3][1] = fieldStart;
```

```
rightStopLocation[3][0] = topBallColumn[2];
rightStopLocation[3][1] = fieldStart;
leftStopLocation[4][0] = topBallColumn[3];
leftStopLocation[4][1] = fieldStart;
rightStopLocation[4][0] = topBallColumn[4];
rightStopLocation[4][1] = fieldStart;

//Third swingbar row stop locations
fieldStart = (screenWidth / 18 * 4) + (screenHeight / 12 * 5);

leftStopLocation[5][0] = topBallColumn[0];
leftStopLocation[5][1] = fieldStart;
rightStopLocation[5][0] = topBallColumn[1];
rightStopLocation[5][1] = fieldStart;
leftStopLocation[6][0] = topBallColumn[2];
leftStopLocation[6][1] = fieldStart;
rightStopLocation[6][0] = topBallColumn[3];
rightStopLocation[6][1] = fieldStart;
leftStopLocation[7][0] = topBallColumn[4];
leftStopLocation[7][1] = fieldStart;
rightStopLocation[7][0] = topBallColumn[5];
rightStopLocation[7][1] = fieldStart;

//Initialize trigger locations

//First swingbar row trigger locations
fieldStart = screenHeight / 4;

leftTriggerLocation[0][0] = topBallColumn[0];
leftTriggerLocation[0][1] = fieldStart;
rightTriggerLocation[0][0] = topBallColumn[1];
rightTriggerLocation[0][1] = fieldStart;
leftTriggerLocation[1][0] = topBallColumn[2];
leftTriggerLocation[1][1] = fieldStart;
rightTriggerLocation[1][0] = topBallColumn[3];
rightTriggerLocation[1][1] = fieldStart;
leftTriggerLocation[2][0] = topBallColumn[4];
leftTriggerLocation[2][1] = fieldStart;
rightTriggerLocation[2][0] = topBallColumn[5];
rightTriggerLocation[2][1] = fieldStart;

//Second swingbar row trigger locations
fieldStart = (screenWidth / 18 * 2) + (screenHeight / 12 * 5);

leftTriggerLocation[3][0] = topBallColumn[1];
leftTriggerLocation[3][1] = fieldStart;
rightTriggerLocation[3][0] = topBallColumn[2];
rightTriggerLocation[3][1] = fieldStart;
leftTriggerLocation[4][0] = topBallColumn[3];
leftTriggerLocation[4][1] = fieldStart;
rightTriggerLocation[4][0] = topBallColumn[4];
rightTriggerLocation[4][1] = fieldStart;

//Third swingbar row trigger locations
fieldStart = (screenWidth / 18 * 4) + (screenHeight / 12 * 7);

leftTriggerLocation[5][0] = topBallColumn[0];
leftTriggerLocation[5][1] = fieldStart;
rightTriggerLocation[5][0] = topBallColumn[1];
rightTriggerLocation[5][1] = fieldStart;
leftTriggerLocation[6][0] = topBallColumn[2];
```

```
    leftTriggerLocation[6][1] = fieldStart;
    rightTriggerLocation[6][0] = topBallColumn[3];
    rightTriggerLocation[6][1] = fieldStart;
    leftTriggerLocation[7][0] = topBallColumn[4];
    leftTriggerLocation[7][1] = fieldStart;
    rightTriggerLocation[7][0] = topBallColumn[5];
    rightTriggerLocation[7][1] = fieldStart;

    //Initialize score and balls left
    score = 0;
    ballsLeft = 10;

  }

  //Gameplay calculations
  private void gamePlay()
  {
    AnimationThread animationThread = new AnimationThread();

    int count, countb;
    int freeBall = 0;
    boolean found = false;

    //Reset the distance all balls have travelled in the current turn to 0.
    for (count = 0; count < 9; ++count)
    {
      ballMovement[count] = 0;
    }

    if (ballsLeft > 0)
    {
      //Find the first ball that is currently unused.
      while (!found)
      {
        if (movingBall[freeBall][2] == 0)
          found = true;
        else
          ++freeBall;
      }

      //Initialize the new ball values
      movingBall[freeBall][0] = topBallColumn[currTopBallColumn];
      movingBall[freeBall][1] = 0 - ballDiameter;
      movingBall[freeBall][2] = 1;
      movingBall[freeBall][3] = 1;
      movingBall[freeBall][4] = 0;
      movingBall[freeBall][5] = 1;
      movingBall[freeBall][6] = currTopBallColumn;

      //Remove the top column ball
      currTopBallColumn = 6;

      --ballsLeft;

      //Lock user controls
      currentlyAnimating = true;
      animationThread.start();
    }
    else
    {
        //Game is over, ask for new game.
```

```
      if (Dialog.ask(Dialog.D_YES_NO, "Game over. Play again?") == Dialog.YES)
      {
        //User wants to play again, reset game and screen.
        UiApplication.getUiApplication().invokeAndWait(new Runnable() {
          public void run()
          {

            initialize();
            tumblerScreen.invalidate();
          }
        });
      }
      else
      {
        //User does not want to play again.  Pop the game screen to return to the main
        //into screen.
        popScreen(tumblerScreen);
      }

  }
}


//Handles all animation calculations and collision detection
private class AnimationThread extends Thread
{
  public void run()
  {
  int count, countb, countc, countd;  //For loop counters
  boolean swingBarChange = false;
  boolean checkForHighScore, keepGoing;
  int highScorePosition;
  int ballsFell = 0;  //The number of balls that have left the screen this turn

    //Loop animation while any ball is currently in motion
    while (movingBall[0][5] == 1 || movingBall[1][5] == 1
        || movingBall[2][5] == 1 || movingBall[3][5] == 1
        || movingBall[4][5] == 1 || movingBall[5][5] == 1
        || movingBall[6][5] == 1 || movingBall[7][5] == 1
        || movingBall[8][5] == 1)
    {
      //For each ball
      for (countb = 0; countb < 9; ++countb)
      {
        //Check if the ball is currently moving
        if (movingBall[countb][5] == 1)
        {
          //Which direction is the ball moving?
          if (movingBall[countb][4] == 0)
          //Moving down
          {
            movingBall[countb][1] += gameSpeed;
          }
          else if (movingBall[countb][4] == 1)
          //Moving left
          {
            movingBall[countb][0] -= gameSpeed;
            ballMovement[countb] += gameSpeed;

            //Check to see if we are in the next column
            if (ballMovement[countb] >= columnWidth)
```

```
  {
    //Decrement the column the ball is in and set the
    //x coordinate to the previous column value in case
    //we passed it since we can move 1 to 5 pixels each frame.

    --movingBall[countb][6];
    movingBall[countb][0] = topBallColumn[movingBall[countb][6]];
    ballMovement[countb] = 0;

    //Set the direction to down.
    movingBall[countb][4] = 0;
  }

} // End if (movingBall[countb][5] == 1)
else if (movingBall[countb][4] == 2)
//Moving right
{
  movingBall[countb][0] += gameSpeed;
  ballMovement[countb] += gameSpeed;

  //Check to see if we are in the next column

  if (ballMovement[countb] >= columnWidth)
  {
    //Decrement the column the ball is in and set the
    //x coordinate to the previous column value in case
    //we passed it since we can move 1 to 5 pixels each frame.

    ++movingBall[countb][6];
    movingBall[countb][0] = topBallColumn[movingBall[countb][6]];
    ballMovement[countb] = 0;

    //Set the direction to down.
    movingBall[countb][4] = 0;
  }
}    // End else if (movingBall[countb][4] == 2)

//Check to see if the ball has left the screen
if (movingBall[countb][1] >= screenHeight)
{
  //Reset ball to off screen and not moving.
  movingBall[countb][2] = 0;
  movingBall[countb][5] = 0;

  ++ballsFell;
  score += 10 * ballsFell;
}

//For each swingbar
for (countc = 0; countc < 8; ++countc)
{
  //Is swingbar in the stable left position?
  if (swingBarTop[countc][4] == 0)
  {
    //Check to see if the ball landed on a platform
    if (movingBall[countb][0] == leftStopLocation[countc][0] &&
        movingBall[countb][1] < leftStopLocation[countc][1] &&
        movingBall[countb][1] >= (leftStopLocation[countc][1] - gameSpeed))
    {
      //Stop moving the ball and set the y value to the platform level.
      movingBall[countb][5] = 0;
```

```
  movingBall[countb][1] = leftStopLocation[countc][1];
}
//Check to see if the ball hit a shaft
else if (movingBall[countb][0] == rightTriggerLocation[countc][0] &&
    movingBall[countb][1] < rightTriggerLocation[countc][1] &&
    movingBall[countb][1] >= (rightTriggerLocation[countc][1] - gameSpeed))
{
  //Set the swingbar to a transitional state
  swingBarShaft[countc][4] = 2;
  swingBarTop[countc][4] = 2;
  swingBarChange = true;

}

//Check to see if the ball landed on another ball
//Check all other 8 balls
for (countd = 0; countd < 9; ++countd)
{
  //Ensure the ball we colided with was not moving
  if (movingBall[countd][5] == 0)
  {
    if (movingBall[countd][0] == movingBall[countb][0] &&
      movingBall[countb][1]  >= (movingBall[countd][1] - ballDiameter + 1) &&
      movingBall[countb][1] <=
      (movingBall[countd][1]  - ballDiameter + 1 + gameSpeed))
    {
      //There was a ball collision, determine which way to roll
      if ((movingBall[countd][0] == topBallColumn[5]) ||
        (movingBall[countd][0] == topBallColumn[1] &&
        movingBall[countd][1] == rightStopLocation[0][1]) ||
        (movingBall[countd][0] == topBallColumn[3] &&
        movingBall[countd][1] == rightStopLocation[1][1]) ||
        (movingBall[countd][0] == topBallColumn[2] &&
        movingBall[countd][1] == rightStopLocation[3][1]) ||
        (movingBall[countd][0] == topBallColumn[4] &&
        movingBall[countd][1] == rightStopLocation[4][1]) ||
        (movingBall[countd][0] == topBallColumn[1] &&
        movingBall[countd][1] == rightStopLocation[5][1]) ||
        (movingBall[countd][0] == topBallColumn[3] &&
        movingBall[countd][1] == rightStopLocation[6][1]))
      {
        //Roll the ball we landed on to the left
        movingBall[countd][4] = 1;
        movingBall[countd][5] = 1;
      }
      else if ((movingBall[countd][0] == topBallColumn[0]) ||
          (movingBall[countd][0] == topBallColumn[2] &&
          movingBall[countd][1] == leftStopLocation[1][1]) ||
          (movingBall[countd][0] == topBallColumn[4] &&
          movingBall[countd][1] == leftStopLocation[2][1]) ||
          (movingBall[countd][0] == topBallColumn[1] &&
          movingBall[countd][1] == leftStopLocation[3][1]) ||
          (movingBall[countd][0] == topBallColumn[3] &&
          movingBall[countd][1] == leftStopLocation[4][1]) ||
          (movingBall[countd][0] == topBallColumn[2] &&
          movingBall[countd][1] == leftStopLocation[6][1]) ||
          (movingBall[countd][0] == topBallColumn[4] &&
          movingBall[countd][1] == leftStopLocation[7][1]))
      {
        //Roll the ball we landed on to the right
        movingBall[countd][4] = 2;
```

```
                movingBall[countd][5] = 1;
              }
            }
          }
        }
      }
      //Is swingbar in the stable right position?
      else if (swingBarTop[countc][4] == 1)
      {
        //Check to see if the ball landed on a platform
        if (movingBall[countb][0] == rightStopLocation[countc][0] &&
            movingBall[countb][1] < rightStopLocation[countc][1] &&
            movingBall[countb][1] >= (rightStopLocation[countc][1] - gameSpeed))
        {
          //Stop moving the ball and set the y value to the platform level.
          movingBall[countb][5] = 0;
          movingBall[countb][1] = rightStopLocation[countc][1];
        }
        //Check to see if the ball hit a shaft
        else if (movingBall[countb][0] == leftTriggerLocation[countc][0] &&
            movingBall[countb][1] < leftTriggerLocation[countc][1] &&
            movingBall[countb][1] >= (leftTriggerLocation[countc][1] - gameSpeed))
        {
          //Set the swingbar to a transitional state
          swingBarShaft[countc][4] = 3;
          swingBarTop[countc][4] = 3;
          swingBarChange = true;
        }
      }

    }
  }
}

//Repaint the screen.
UiApplication.getUiApplication().invokeAndWait(new Runnable() {
  public void run()
  {
    tumblerScreen.invalidate();
  }
});

//Do any swingbars need to be animated?
if (swingBarChange)
{
  for (countb = 0; countb < animationDistance; countb += gameSpeed)
  {

    //Check to see which swingbars need to be animated.
    //If they do, animate them before the next ball animation frame.

    for (countc = 0; countc < 8; ++countc)
    {
        if (swingBarShaft[countc][4] == 2)
        {
          //Swingbar is transitioning from left to right.

          if ((countb + gameSpeed) < animationDistance)
          {
            //Not in the last frame, just add gameSpeed.
            swingBarShaft[countc][0] += gameSpeed;
```

```
            swingBarShaft[countc][2] -= gameSpeed;
            swingBarTop[countc][0] += gameSpeed;
            swingBarTop[countc][2] += gameSpeed;
          }
          else
          {
            //Don't move the full gameSpeed, just enough
            //to move to the next position so we don't go past
            //the stop location since we can move 1 to 5 pixels
            //each frame.
            int animationRemainder = gameSpeed % animationDistance;

            swingBarShaft[countc][0] += animationRemainder;
            swingBarShaft[countc][2] -= animationRemainder;
            swingBarTop[countc][0] += animationRemainder;
            swingBarTop[countc][2] += animationRemainder;
          }

        }
        else if (swingBarShaft[countc][4] == 3)
        {
          //Swingbar is transitioning from right to left

          if ((countb + gameSpeed) < animationDistance)
          {
            //Not in the last frame, just add gameSpeed.
            swingBarShaft[countc][0] -= gameSpeed;
            swingBarShaft[countc][2] += gameSpeed;
            swingBarTop[countc][0] -= gameSpeed;
            swingBarTop[countc][2] -= gameSpeed;
          }
          else
          {
            //Don't move the full gameSpeed, just enough
            //to move to the next position so we don't go past
            //the stop location since we can move 1 to 5 pixels
            //each frame.
            int animationRemainder = gameSpeed % animationDistance;

            swingBarShaft[countc][0] -= animationRemainder;
            swingBarShaft[countc][2] += animationRemainder;
            swingBarTop[countc][0] -= animationRemainder;
            swingBarTop[countc][2] -= animationRemainder;
          }
        }
    }

    //Repaint the screen.
    UiApplication.getUiApplication().invokeAndWait(new Runnable()
    {
      public void run()
      {
        tumblerScreen.invalidate();
      }
    });

  }

  swingBarChange = false;

  //Update swingbar states to their new position.
```

```
    for (countb = 0; countb < 8; ++countb)
    {
        //Change state from transitional to stable.
        if (swingBarShaft[countb][4] == 2)
        {
          swingBarShaft[countb][4] = 1;
          swingBarTop[countb][4] = 1;

          //Set ball state to in motion because it was resting
          //on a swingbar that has moved.
          for (countc = 0; countc < 9; ++countc)
          {
            //Ball was resting on a left swingbar.
            if (movingBall[countc][2] == 1 &&
              movingBall[countc][0] == leftStopLocation[countb][0] &&
              movingBall[countc][1] == leftStopLocation[countb][1])
            {
              movingBall[countc][5] = 1;
            }
          }
        }
        //Change state from transitional to stable.
        else if (swingBarShaft[countb][4] == 3)
        {
            swingBarShaft[countb][4] = 0;
            swingBarTop[countb][4] = 0;

          //Set ball state to in motion because it was resting
          //on a swingbar that has moved.
          for (countc = 0; countc < 9; ++countc)
          {
            //Ball was resting on a left swingbar.
            if (movingBall[countc][2] == 1 &&
              movingBall[countc][0] == rightStopLocation[countb][0] &&
              movingBall[countc][1] == rightStopLocation[countb][1])
            {
              movingBall[countc][5] = 1;
            }
          }
        }
    }
  }
}

//Unlock user controls.
currentlyAnimating = false;

//Reset the top column ball to the first column.
currTopBallColumn = 0;

//Add more balls if more than 3 dropped off the screen.

if (ballsFell > 2)
  ballsLeft += ballsFell - 2;

//Update the screen.
UiApplication.getUiApplication().invokeAndWait(new Runnable()
{
  public void run()
  {
    tumblerScreen.invalidate();
```

```
      }
    });

    //If game over, check for high score.
    if (ballsLeft == 0)
    {
      highScorePosition = 5;
      checkForHighScore = true;

      //See if the current score is higher then an existing high score.
      while (checkForHighScore && highScorePosition != 0)
      {
        if (score > highScoreValues[highScorePosition - 1])
          --highScorePosition;
        else
          checkForHighScore = false;
      }

      if (highScorePosition < 5)
      //User has a new high score!
      {

        newHighScoreName = "";
        newHighScoreReady = false;

        //Alert the user of the new high score and prompt for their name.
        UiApplication.getUiApplication().invokeAndWait(new Runnable()
        {
          public void run()
          {
            TumblerNewHighScoreDialog highScoreDialog = new TumblerNewHighScoreDialog();
            highScoreDialog.show();
          }
        });

        //Wait for the user to enter their name.
        while (!newHighScoreReady)
        {
          try
          {
            sleep(500);
          } catch (InterruptedException e)
          {
            System.err.println(e.toString());
          }
        }

        keepGoing = true;
        count = 4;

        //Drop the lowest score off the bottom of the list, shift other
        //high scores down and then insert the new high score.
        while (keepGoing)
        {
          if (highScorePosition == count)
          {
            highScoreValues[count] = score;
            highScoreNames[highScorePosition] = newHighScoreName;
            keepGoing = false;
          }
          else
```

```
          {
            highScoreValues[count] = highScoreValues[count - 1];
            highScoreNames[count] = highScoreNames[count - 1];
            --count;
          }
        }

        //Save the new high scores.
        tumblerHighScores.setNames(highScoreNames);
        tumblerHighScores.setScores(highScoreValues);

        synchronized (store)
        {
          store.setContents(tumblerHighScores);
          store.commit();
        }

        //Display the high scores screen.
        UiApplication.getUiApplication().invokeAndWait(new Runnable()
        {
          public void run()
          {
            highScoreScreen = new TumblerHighScoreScreen(highScoreNames, highScoreValues);
            pushScreen(highScoreScreen);
          }
        });

      }
      else
      {
        //No high score, display the new game prompt.
        UiApplication.getUiApplication().invokeAndWait(new Runnable()
        {
          public void run()
          {
            if (Dialog.ask(Dialog.D_YES_NO, "Game over. Play again?") == Dialog.YES)
            {
              //Game over, prompte for a new game.
              UiApplication.getUiApplication().invokeAndWait(new Runnable()
              {
                public void run()
                {
                  initialize();
                  tumblerScreen.invalidate();
                }
              });
            }
          }
        });

      }
    }
  }
}

//Draw all graphics
private class TumblerScreen extends MainScreen
{

  public void start()
  {
```

```
    }

    //draw objects to the screen here
    public void paint(Graphics graphics)
    {
      int count, fieldStart, fieldEnd;
      String statusLine = new String("Balls: " + ballsLeft + " Score: " + score);

      //Draw the top status text.
      graphics.drawText(statusLine, 0, Graphics.getScreenHeight() - 15);

      //Draw the 6 drop slots.

      fieldStart = screenWidth / 6;
      fieldEnd = screenWidth / 18;

      for (count = 0; count < 6; ++count)
      {
        graphics.drawRect((fieldStart) * count, 0, fieldStart, fieldEnd);
      }

      fieldStart = screenWidth / 18;
      fieldEnd = screenHeight / 6;

      //Draw the separator lines.

      //First row of speparators, from left to right.
      graphics.drawLine(0, (fieldStart), 0, ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(1, (fieldStart), 1, ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 3 - 1, (fieldStart), screenWidth / 3 - 1, ((fieldEnd) +
(screenWidth / 18)));
      graphics.drawLine(screenWidth / 3, (fieldStart), screenWidth / 3,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 3 * 2 - 1, (fieldStart), screenWidth / 3 * 2 - 1,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 3 * 2, (fieldStart), screenWidth / 3 * 2,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth - 2, (fieldStart), screenWidth - 2,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth - 1, (fieldStart), screenWidth - 1,
        ((fieldEnd) + (screenWidth / 18)));

      fieldStart = (screenWidth / 18 * 3) + (screenHeight / 6);
      fieldEnd = (screenWidth / 9) + (screenHeight / 3);

      //Second row of separators, from left to right.
      graphics.drawLine(0, (fieldStart), 0, ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(1, (fieldStart), 1, ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 6 - 1, (fieldStart), screenWidth / 6 - 1,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 6, (fieldStart), screenWidth / 6,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 2 - 1, (fieldStart), screenWidth / 2 - 1,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 2, (fieldStart), screenWidth / 2,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 6 * 5 - 1, (fieldStart), screenWidth / 6 * 5 - 1,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth / 6 * 5, (fieldStart), screenWidth / 6 * 5,
        ((fieldEnd) + (screenWidth / 18)));
      graphics.drawLine(screenWidth - 2, (fieldStart), screenWidth - 2,
```

```
      ((fieldEnd) + (screenWidth / 18)));
    graphics.drawLine(screenWidth - 1, (fieldStart), screenWidth - 1,
      ((fieldEnd) + (screenWidth / 18)));

    fieldStart = (screenWidth / 18 * 5) + (screenHeight / 3);
    fieldEnd = (screenWidth / 18 * 4) + (screenHeight / 2);

    //Third row of separators, from left to right.
    graphics.drawLine(0, (fieldStart), 0, ((fieldEnd) + (screenWidth / 18)));
    graphics.drawLine(1, (fieldStart), 1, ((fieldEnd) + (screenWidth / 18)));
    graphics.drawLine(screenWidth / 3 - 1, (fieldStart), screenWidth / 3 - 1,
      ((fieldEnd) + (screenWidth / 18)));
    graphics.drawLine(screenWidth / 3, (fieldStart), screenWidth / 3,
      ((fieldEnd) + (screenWidth / 18)));
    graphics.drawLine(screenWidth / 3 * 2 - 1, (fieldStart), screenWidth / 3 * 2 - 1,
      ((fieldEnd) + (screenWidth / 18)));
    graphics.drawLine(screenWidth / 3 * 2, (fieldStart), screenWidth / 3 * 2,
      ((fieldEnd) + (screenWidth / 18)));
    graphics.drawLine(screenWidth - 2, (fieldStart), screenWidth - 2,
      ((fieldEnd) + (screenWidth / 18)));
    graphics.drawLine(screenWidth - 1, (fieldStart), screenWidth - 1,
      ((fieldEnd) + (screenWidth / 18)));

    //Draw all the swing bars
    for (count = 0; count < 8; ++count)
    {
        graphics.drawLine(swingBarShaft[count][0], swingBarShaft[count][1],
            swingBarShaft[count][2], swingBarShaft[count][3]);

        graphics.drawLine(swingBarTop[count][0], swingBarTop[count][1],
            swingBarTop[count][2], swingBarTop[count][3]);
    }

    //Draw the ball in the top bar in its current location,
    //unless it is currently being dropped (currTopBall = 6).

    if (currTopBallColumn != 6)
    {
      graphics.fillArc(topBallColumn[currTopBallColumn], 0,
            ballDiameter, ballDiameter, 15, 360);
    }

    //Draw all the moving balls that are on screen.

    for (count = 0; count < 9; ++count)
    {
        if (movingBall[count][2] == 1)
        {
          graphics.fillArc(movingBall[count][0], movingBall[count][1],
              ballDiameter, ballDiameter, 0, 360);
        }
    }
  }
}

//Invoked when the trackwheel is clicked.
public boolean trackwheelClick(int status, int time)
{
  if (!currentlyAnimating)
  {
    gamePlay();
```

```
    }

    return true;
  }

  //Invoked when the trackwheel is released.
  public boolean trackwheelUnclick(int status, int time)
  {
    return false;
  }

  //Invoked when the trackwheel is rolled.
  public boolean trackwheelRoll(int amount, int status, int time)
  {
    //User controls are disabled if the game is animating.
    if (!currentlyAnimating)
    {
      if (amount < 0)
      {
        if (currTopBallColumn == 0)
          currTopBallColumn = 5;
        else
          --currTopBallColumn;
      }
      else
      {
        if (currTopBallColumn == 5)
          currTopBallColumn = 0;
        else
          ++currTopBallColumn;
      }
      tumblerScreen.invalidate();
    }
    return true;
  }

  public boolean keyChar(char key, int status, int time)
  {
    boolean retval = false;
    //User controls are disabled if the game is animating.
    if (!currentlyAnimating)
    {
      switch (key)
      {
        case Characters.ENTER:
          gamePlay();
          retval = true;
          break;

        case Characters.SPACE:
          gamePlay();
          retval = true;
          break;

        case Characters.ESCAPE:
          if (Dialog.ask(Dialog.D_YES_NO, "Quit game?") == Dialog.YES)
            popScreen(tumblerScreen);
          retval = true;
          break;
      }
    }
```

```
    else if (key == Characters.ESCAPE)
    {
      if (Dialog.ask(Dialog.D_YES_NO, "Quit game?") == Dialog.YES)
        popScreen(tumblerScreen);
        retval = true;
    }

    return retval;
  }

  //Implementation of KeyListener.keyStatus.
  public boolean keyStatus(int keycode, int time)
  {
     return false;
  }

  //Implementation of KeyListener.keyDown.
  public boolean keyDown(int keycode, int time)
  {
     return false;
  }

  //Implementation of KeyListener.keyRepeat.
  public boolean keyRepeat(int keycode, int time)
  {
     return false;
  }

  //Implementation of KeyListener.keyUp.
  public boolean keyUp(int keycode, int time)
  {
     return false;
  }

  //Set a new high score name.
  public static void setNewHighScoreName(String name)
  {
      newHighScoreName = name;
  }

  //Set the new high score flag.
  public static void setNewHighScoreReady(boolean ready)
  {
      newHighScoreReady = ready;
  }

}
```

TumblerIntroScreen.java

```
/**
 * Tumbler Intro Screen
 *
 * Author:  Mark Sohm
 */

package com.msohm.tumbler;

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

public final class TumblerIntroScreen extends MainScreen
{
  public TumblerIntroScreen()
  {
    super(MainScreen.FIELD_HCENTER | MainScreen.FIELD_VCENTER);
  }

  public boolean onClose()
  {
    System.exit(0);
    return true;
  }
}
```

TumblerHighScores.java

```
/**
 * Tumbler High Scores
 *
 * Author:  Mark Sohm
 *
 * The persistant class used to store the high score list.
 */

package com.msohm.tumbler;

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


public final class TumblerHighScores implements Persistable
{
  //Array of string and ints to hold the high score names and score values.
  private String[] highNames = new String[5];
  private int highScores[] = new int[5];

  public TumblerHighScores(String[] names, int scores[])
  {
    highNames = names;
    highScores = scores;
  }
  public String[] getNames()
  {
    return highNames;
  }

  public void setNames(String[] names)
  {
    highNames = names;
  }

  public int[] getScores()
  {
    return highScores;
  }

  public void setScores(int[] scores)
  {
    highScores = scores;
  }

}
```

TumblerHighScoreScreen.java

```
/**
 * Tumbler High Score Screen
 *
 * Author:  Mark Sohm
 *
 * Screen that displays the Tumbler high scores.
 */

package com.msohm.tumbler;

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

public final class TumblerHighScoreScreen extends MainScreen
{
  public TumblerHighScoreScreen(String[] names, int[] scores)
  {
    super();

    //Screen is made up of two VerticalFieldManagers in a HorizontalFieldManager.
    //Gives the effect of having two columns for the fields.
    HorizontalFieldManager backGround = new HorizontalFieldManager();
    VerticalFieldManager leftColumn = new VerticalFieldManager();
    VerticalFieldManager rightColumn = new VerticalFieldManager();

    LabelField displayNames[] = new LabelField[5];
    LabelField displayScores[] = new LabelField[5];
    int count;

    LabelField title = new LabelField("Tumbler High Scores", LabelField.ELLIPSIS |
LabelField.USE_ALL_WIDTH);
    setTitle(title);

    //Column headings.
    LabelField leftTitle = new LabelField("Player  ");
    leftColumn.add(leftTitle);
    LabelField rightTitle = new LabelField("Score");
    rightColumn.add(rightTitle);

    //Blank label fields to add a blank line.
    leftColumn.add(new LabelField(""));
    rightColumn.add(new LabelField(""));

    //Add fields that list the high score names and values.
    for (count = 0; count < 5; count++)
    {

      displayNames[count] = new LabelField(names[count]);
      displayScores[count] = new LabelField(String.valueOf(scores[count]));

      leftColumn.add(displayNames[count]);
      rightColumn.add(displayScores[count]);

    }

    backGround.add(leftColumn);
```

```
    backGround.add(rightColumn);
    add(backGround);
  }

  public boolean onClose()
  {
    System.exit(0);
    return true;
  }

  //Close the high score screen whenever a key is pressed.
  public boolean keyChar(char key, int status, int time)
  {
    this.close();
    return true;
  }

  //Close the high score screen if the trackwheel is clicked.
  public boolean trackwheelClick(int status, int time)
  {
    this.close();
    return true;
  }
}
```

TumblerNewHighScoreDialog.java

```
/**
 * Tumbler New High Score Dialog
 *
 * Author:  Mark Sohm
 *
 * Custom dialog with a text entry field that prompts the user for their
 * name that will be added to the high scores list.
 */

package com.msohm.tumbler;

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

public final class TumblerNewHighScoreDialog extends Dialog
{
  private EditField name;

  public TumblerNewHighScoreDialog()
  {
    super(Dialog.D_OK, "New High Score", 1, Bitmap.getPredefinedBitmap
      (Bitmap.EXCLAMATION), Manager.FOCUSABLE);

    //Add an EditField to the standard OK dialog.
    name = new EditField("Enter your name: ", "", 10, EditField.EDITABLE);
    add(name);

  }

  public boolean keyChar(char key, int status, int time)
  {
    //Override key commands

    switch (key)
    {
      case Characters.ENTER:
        //Update main class with high score name and alert it that the new name is ready.
        Tumbler.setNewHighScoreName(name.getText());
        Tumbler.setNewHighScoreReady(true);
        this.close();
        break;

      case Characters.BACKSPACE:
        if (name.getTextLength() > 0)
        {
          name.setText(name.getText().substring(0, name.getTextLength() - 1));
        }
        break;

      case Characters.ESCAPE:
        if (name.getTextLength() > 0)
        {
          name.setText(name.getText().substring(0, name.getTextLength() - 1));
        }
        break;
```

```
      default:
        name.setText(name.getText() + key);
        break;
    }

    return true;
  }

  //Update main class with high score name and alert it that the new name is ready.
  public boolean trackwheelClick(int status, int time)
  {
    Tumbler.setNewHighScoreName(name.getText());
    Tumbler.setNewHighScoreReady(true);
    this.close();
    return true;
  }
}
```

TumblerInstructionScreen.java

```
/**
 * Tumbler Instruction Screen
 *
 * Author:  Mark Sohm
 *
 * Screen that displays instructions for Tumbler.
 */

package com.msohm.tumbler;

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

public final class TumblerInstructionScreen extends MainScreen
{
  public TumblerInstructionScreen()
  {
    super();

    String instructions = "Gameplay\n~~~~~~\nThe goal of Tumbler is to score points by dropping
balls off of the bottom of the screen.  You start with 10 balls but are able to earn more.  If more
than 2 balls fall off the screen in one turn you will receive an additional ball for every ball that
fell off the screen beyond the first two.\n\nKeys\n~~~\nUse the trackwheel to select the column you
wish to use and then click the trackwheel, spacebar or enter key to drop a ball.  Exit the game by
pressing the escape key.\n\nScoring\n~~~~~\nPoints are granted based upon how many balls fall off
the screen in one turn.  For example if 3 balls fell off the screen in one turn you would receive 10
points for the first ball, 20 for the second and 30 for the third.";

    LabelField title = new LabelField("How To Play Tumbler",
      LabelField.ELLIPSIS | LabelField.USE_ALL_WIDTH);
    setTitle(title);

    RichTextField instructionsLabel = new RichTextField(instructions);
    add(instructionsLabel);
  }

  public boolean onClose()
  {
    System.exit(0);
    return true;
  }

  //Close the instruction screen whenever a key is pressed.
  public boolean keyChar(char key, int status, int time)
  {
    this.close();
    return true;
  }

  //Close the instruction screen if the trackwheel is clicked.
  public boolean trackwheelClick(int status, int time)
  {
    this.close();
    return true;
  }
}
```

TumblerSyncItem.java

```
/**
 * TumblerSyncItem.java
 *
 * Handles the backup and restore of the high scores to/from the users Desktop.
 *
 * Author:  Mark Sohm
 */

package com.msohm.tumbler;

import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
import net.rim.device.api.synchronization.*;
import java.util.*;
import net.rim.device.api.i18n.Locale;


public final class TumblerSyncItem extends SyncItem
{
  private String[] permNames = new String[5];
  private int permScores[] = new int[5];
  private static TumblerHighScores permHighScores;
  private static final int FIELDTAG_NAME = 1;
  private static final int FIELDTAG_SCORE = 2;

  private static PersistentObject store;

  //This store is static so that all instances of this class use the same
  //Persistent Store.
  static
  {
    //Unique identifier for the persistent store object.
    //Long value = com.msohm.tumbler.Tumbler
    store = PersistentStore.getPersistentObject(0x6086b8131c33cfc2L);
  }

  public TumblerSyncItem()
  {
  }

  //The name that will be used to reference this SyncItem.
  //It is shown in the Desktop Manager under backup/restore advanced.
  public String getSyncName()
  {
    return "Tumbler High Scores";
  }

  //Localization is not supported in this example.
  public String getSyncName(Locale locale)
  {
    return null;
  }

  //The version of this SyncItem.
  public int getSyncVersion()
  {
    return 1;
  }
```

```
//Formats the data to the specification required by the Desktop Manager.
public boolean getSyncData(DataBuffer db, int version)
{

  int count;
  boolean retVal = true;
  String tempString;

  synchronized (store)
  {
    //Check to see if the persistent store object exists on the BlackBerry.
    if (store.getContents() != null)
    {
      //Store exists, retrieve data from store.
      permHighScores = (TumblerHighScores)store.getContents();
      permNames = permHighScores.getNames();
      permScores = permHighScores.getScores();
    }
  }

  //Format data that will be interpreted by the Desktop Manager.
  //Format is Length Type Data
  //Length is a short, type is a byte and data is the length specified by Length.
  //Data must be structured in this format to be understood by the Desktop Manager.
  try
  {
    for (count = 0; count < 5; ++count)
    {
      //Write the name.
      db.writeShort(permNames[count].length() + 1);
      db.writeByte(FIELDTAG_NAME);
      db.write(permNames[count].getBytes());
      db.writeByte(0);
      //Write the age.
      tempString = permScores[count] + "";
      db.writeShort(tempString.length() + 1);
      db.writeByte(FIELDTAG_SCORE);
      db.write(tempString.getBytes());
      db.writeByte(0);
  }
  } catch (Exception e)
  {
    retVal = false;
  }

  //Return true if all data was processed.
  return retVal;
}

//Interprets and stores the data sent from the Desktop Manager.
public boolean setSyncData(DataBuffer db, int version)
{

  int length, nameCount = 0, scoreCount = 0;
  boolean retVal = true;

  try
  {
    //Read until the end of the Databuffer.
    while (db.available() > 0)
    {
```

```
      //Read the length of the data.
      length = db.readShort();

      //Set the byte array to the length of the data.
      byte[] bytes = new byte[length];

      //Determine the type of data to be read (name or age).
      switch (db.readByte())
      {
        case FIELDTAG_NAME:
          //Read the name from the Databuffer.
          //Convert and store it in the String array.
          db.readFully(bytes);
          permNames[nameCount] = new String(bytes).trim();
          ++nameCount;
          break;

        case FIELDTAG_SCORE:
          //Read the age from the Databuffer.
          //Convert and store it in the int array.
          db.readFully(bytes);
          permScores[scoreCount] = (int)(Integer.parseInt(new String(bytes).trim()));
          ++scoreCount;
          break;
      }
    }

  } catch (Exception e)
  {
    retVal = false;
  }

  try
  {
    //Store the new data in the persistent store object.
    permHighScores = new TumblerHighScores(permNames, permScores);
    store.setContents(permHighScores);
    store.commit();
  } catch (Exception e)
  {
    retVal = false;
  }

  //Return true if the data was successfully restored, or false if it was not.

  return retVal;
}
```

TumblerOptions.java

```
/**
 * Tumbler Options
 *
 * Author:  Mark Sohm
 *
 * The persistant class used to store Tumbler options (game speed).
 */

package com.msohm.tumbler;

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


public final class TumblerOptions implements Persistable
{
  private int speed = 1;

  public TumblerOptions()
  {
    speed = 1;
  }

  public TumblerOptions(int gameSpeed)
  {
    speed = gameSpeed;
  }

  public int getGameSpeed()
  {
    return speed;
  }

  public void setGameSpeed(int gameSpeed)
  {
    speed = gameSpeed;
  }

}
```

TumblerOptionsScreen.java

```
/**
 * Tumbler Options Screen
 *
 * Author:  Mark Sohm
 *
 * Screen that displays options for Tumbler.
 */

package com.msohm.tumbler;

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

public final class TumblerOptionsScreen extends MainScreen
{
  private static PersistentObject store;  //Permanently store the game speed here.
  private static TumblerOptions tumblerOptions;
  private NumericChoiceField speed;
  int gameSpeed;

  static
  {
    //Long value = com.msohm.tumbler.TumblerOptions
    store = PersistentStore.getPersistentObject(0x48d43d190c69ef01L);
  }

  public TumblerOptionsScreen(TumblerOptions options)
  {
    super();

    tumblerOptions = options;
    gameSpeed = tumblerOptions.getGameSpeed();

    LabelField title = new LabelField("Tumbler Options", LabelField.ELLIPSIS |
LabelField.USE_ALL_WIDTH);
    setTitle(title);

    speed = new NumericChoiceField("Game Speed", 1, 5, 1, (gameSpeed - 1));
    add(speed);

    MenuItem saveClose = new MenuItem("Save", 30, 30)
    {
      public void run()
      {
        saveOptions();
      }
    };

    addMenuItem(saveClose);
  }

  //Save the tumbler options.
  private boolean saveOptions()
  {
    tumblerOptions.setGameSpeed(speed.getSelectedValue());
```

```
    synchronized (store)
    {
      store.setContents(tumblerOptions);
      store.commit();
    }

    //Inform the screen that the data has been saved.
    setDirty(false);

    return true;
  }

  //Is called on the exiting Save/Discard/Cancel prompt if the user selects save.
  public boolean onSave()
  {
    return saveOptions();
  }

}
```

© 1997-2004 Research In Motion Limited. All rights reserved. The BlackBerry and RIM families of related marks, images and symbols are the exclusive properties of Research In Motion Limited. RIM, Research In Motion, 'Always On, Always Connected', the "envelope in motion" symbol and the BlackBerry logo are trademarks registered with the U.S. Patent and Trademark Office and may be pending or registered in other countries. All other brands, product names, company names, trademarks and service marks are the properties of their respective owners. The BlackBerry handheld and/or associated software are protected by copyright, international treaties and various patents, including one or more of the following U.S. patents: 6,278,442; 6,271,605; 6,219,694; 6,075,470; 6,073,318; D,445,428; D,433,460; D,416,256. Other patents are registered or pending in various countries around the world. Please visit www.rim.net/patents.shtml for a current listing of applicable patents.

NOTE

This document is provided for informational and non-commercial or personal use only and must not be copied, disclosed, or posted on any network computer or broadcast in any media or otherwise distributed, in whole or in part. Any such copying, distribution, disclosure, posting, or broadcast is a violation of copyright laws. This document must not be modified. Use for any other purpose is expressly prohibited by law, and may result in severe civil and criminal penalties. Violators will be prosecuted to the maximum extent possible.

No Research In Motion Limited or BlackBerry logo, graphic, sound or image may be copied or retransmitted unless expressly permitted in writing by Research In Motion Limited.

RESEARCH IN MOTION LIMITED AND ITS SUBSIDIARIES AND AFFILIATES ("RIM") MAKE NO WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THE CONTENT OF THIS DOCUMENT, AND ALL INFORMATION PROVIDED HEREIN IS PROVIDED "AS IS". RIM HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS INFORMATION, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. TO THE EXTENT PERMITTED BY LAW, IN NO EVENT SHALL RIM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES FOR ANY USE OF THIS DOCUMENT, INCLUDING WITHOUT LIMITATION, RELIANCE ON THE INFORMATION PRESENTED, LOST PROFITS OR BUSINESS INTERRUPTION, EVEN IF RIM WAS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE EXCLUSIONS AND LIMITATIONS SET OUT HEREIN SHALL APPLY REGARDLESS OF WHETHER A CLAIM AGAINST RIM ARISES FROM A BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), BREACH OF WARRANTY OR CONDITION, OR ANY OTHER TYPE OF CIVIL LIABILITY.

RIM ASSUMES NO RESPONSIBILITY FOR ANY TYPOGRAPHICAL ERRORS, TECHNICAL OR OTHER INACCURACIES IN THIS DOCUMENT. RIM RESERVES THE RIGHT TO PERIODICALLY CHANGE INFORMATION THAT IS CONTAINED IN THIS DOCUMENT; HOWEVER, RIM MAKES NO COMMITMENT TO PROVIDE ANY SUCH CHANGES, UPDATES, ENHANCEMENTS OR OTHER ADDITIONS TO THIS DOCUMENT TO YOU IN A TIMELY MANNER OR AT ALL.

Prior to subscribing to or implementing any third party products or services, it is your responsibility to ensure that the airtime service provider you are working with has agreed to support all of the features of the third party products and services. Installation and use of third party products and services with RIM's products and services may require one or more patent, trademark or copyright licenses in order to avoid infringement of the intellectual property rights of others. You are solely responsible for determining whether such third party licenses are required and are responsible for acquiring any such licenses. To the extent that such intellectual property licenses may be required, RIM expressly recommends that you do not install or use these products and services until all such applicable licenses have been acquired by you or on your behalf. Your use of third party software shall be governed by and subject to you agreeing to the terms of separate software licenses, if any, for those products or services. Any third party products or services that are provided with RIM's products and services are provided "as is". RIM makes no representation, warranty or guarantee whatsoever in relation to the third party products and services and RIM assumes no liability whatsoever in relation to the third party products and services even if RIM has been advised of the possibility of such damages or can anticipate such damages.

RIM does not claim ownership of the materials you provide to RIM in any way. However, by posting, uploading, inputting, providing or submitting any such materials, you are granting RIM and any necessary sublicensees permission to use your submitted materials in connection with the operation of its business, including, without limitation, the rights to: copy, distribute, transmit, publicly display, publicly perform, reproduce, edit, translate and reformat your submitted material; and to publish your name and any contact information you provide in connection with your submitted material. No compensation will be paid with respect to the use by RIM or any necessary licensee of your submitted material, as provided herein. RIM is under no obligation to post or use any material you may provide and RIM may remove any such material at any time in its sole discretion. By posting, uploading, inputting, providing or submitting your material you warrant and represent that you own or otherwise control all of the rights to your material as described in this section including, without limitation, all the rights necessary for you to provide, post, upload, input or submit the material. Notwithstanding the foregoing, any suggestions, improvements or modifications to RIM products and services ("Enhancements") made by you or anyone acting on your behalf, including your employees, will be the property of RIM without any further consideration to you, whether or not such Enhancements are incorporated into RIM products and services.

By posting, uploading, inputting, providing or submitting any information to RIM, you consent to RIM's collection of such information, and you grant RIM, its affiliated companies and necessary sublicensees permission to use such submitted information in connection with the operation of this journal and its business, including, without limitation, the worldwide, royalty-free rights to: copy, distribute, transmit, publicly display, publicly perform, reproduce, edit, translate and reformat your submitted information; and to publish your name and any contact information you provide of in connection with your submitted information.

If you submit personal information to RIM, you consent to the collection, use, and disclosure of such information by RIM in accordance with RIM's privacy policy which can be found at http://www.blackberry.com/legal/privacy.shtml.