Advanced Java Application Development for the BlackBerry Smartphone

BlackBerry Academic Program
Prior to subscribing for, installing or using any Third Party Products and Services it is your responsibility to ensure that your airtime service provider has agreed to support all of their features. Some airtime service providers may not offer Internet browsing functionality with a subscription to BlackBerry® Internet Service. Check with your service provider for availability, roaming arrangements, service plans and features. Installation or use of Third Party Products and Services with RIM's products and services may require one or more patent, trademark, copyright or other licenses in order to avoid infringement or violation of third party rights. You are solely responsible for determining whether to use, Third Party Products and Services and if any third party licenses are required to do so. If required you are responsible for acquiring them. You should not install or use Third Party Products and Services until all necessary licenses have been acquired. Any Third Party Products and Services that are provided with RIM's products and services are provided as a convenience to you and are provided 'AS IS' with no express or implied conditions, endorsements, guarantees, representations or warranties of any kind by RIM and RIM assumes no liability whatsoever, in relation thereto. Your use of Third Party Products and Services shall be governed by and subject to you agreeing to the terms of separate licenses and other agreements applicable thereto with third parties, except to the extent expressly covered by a license or other agreement with RIM.

Certain features outlined in this documentation require a minimum version of BlackBerry® Enterprise Server, BlackBerry® Desktop Software, BlackBerry® Device Software and/or BlackBerry Handheld Software and may require additional development or Third Party Products and Services for access to corporate applications.

Certain products mentioned in this documentation include software developed by the Apache Software Foundation (http://www.apache.org/) and/or licensed pursuant to Apache License, Version 2.0 (http://www.apache.org/licenses/). For more information, see the NOTICE.txt file included with the software. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

The terms of use of any RIM product or service are set out in a separate license or other agreement with RIM applicable thereto. NOTHING IN THIS DOCUMENTATION IS INTENDED TO SUPERSEDE ANY EXPRESS WRITTEN AGREEMENTS OR WARRANTIES PROVIDED BY RIM FOR PORTIONS OF ANY RIM PRODUCT OR SERVICE OTHER THAN THIS DOCUMENTATION.
Chapter 3
Optimization for mobile application development

Objectives

- Describe optimization techniques for BlackBerry® applications
- Create BlackBerry applications that demonstrate the use of optimization techniques
- Demonstrate how to optimize BlackBerry applications using the *BlackBerry IDE profiler* tool
- Identify a memory leak using the Object Viewer

This chapter outlines the optimal coding techniques for designing applications for the BlackBerry smartphone. This chapter lists a number of coding techniques that you can use to write efficient and effective applications to meet the needs of BlackBerry smartphone users.
Optimal BlackBerry coding techniques

Inefficient and ineffective code can result in slow and costly applications. When you develop applications using the best techniques for optimizing your code, the result can be faster, more efficient applications. Use efficient coding techniques to create efficient code. Reuse or reduce code where possible to leverage existing code and to make your application more efficient.

The following general guidelines for effective and efficient BlackBerry application coding can be useful in day-to-day BlackBerry application development.

Writing efficient code

Consider applying the following guidelines when writing a BlackBerry application to create a BlackBerry Java Application that uses resources efficiently.

Using classes effectively

Use or extend a class from the Java® ME platform and BlackBerry APIs where possible, and use inheritance whenever possible.

Some API classes are so full-featured that the speed of your application can become slow and the size of your application can increase. You can prevent this, especially when you only need to use a subset of the complete API class functionality. Profile your code using the API class, substitute your own simplified method, and then profile the code again. The difference between the original code and the optimized code can be substantial in high-traffic areas of code.

Making classes final

When creating code libraries, ensure that you mark a class as final if you know it will never be extended. The compiler can generate faster code when a final keyword is present.

Using methods

Synchronized methods are more expensive than non-synchronized methods. Only declare a method as synchronized if necessary. If you unnecessarily declare a method synchronized, you can degrade the operation. If you are considering synchronizing a method, determine whether you must declare the entire body of the method synchronized, or if you can declare only an isolated section of the method synchronized. The latter option can be more efficient than the former.

The Java® compiler can only inline methods that are either final, private, or static and do not have local variables. If possible, rewrite high traffic methods to be final without local variables for inlining.
If you encounter performance issues with a method within an API class, override the method with a simplified, optimized version of your own.

**Using local variables**

Use local variables whenever possible. Access to local variables is more efficient than access to class members.

**Writing efficient loops**

Always factor loop invariant code out of a loop, as in the following example:

```java
for(int i = 0; i < vector.size(); i++) {
    ...
}
```

In the following sample, `vector.size()` is called each time through the loop, which is inefficient. If your container is likely to have more than one element, it is faster to assign the size to a local variable. In addition, the following sample using preincrement (`++i`) results in smaller code than postincrement (`i++`). The following sample shows the optimized code:

```java
int size = vector.size();
for(int i = 0; i < size; ++i) {
    ...
}
```

Alternatively, if the order in which you iterate over items is not important, you can iterate backward. By iterating backward, you can avoid the extra local on the stack, and the comparison is also faster, as the following code sample illustrates:

```java
for(int i = vector.size() - 1; i >= 0; --i) {
    ...
}
```

**Optimizing subexpressions**

If you use the same expression twice, do not rely on the compiler to optimize the expression for you. Use a local variable, as in the following code sample:
one(i + 1); two(i + 1); // Avoid
int tmp = i + 1;
one(tmp);
two(tmp); // Prefer

Optimizing division operations

Division operations are slow on the BlackBerry smartphone because the BlackBerry smartphone processor does not have a hardware divide instruction. When you write code that divides a positive integer by $2^n$, where $n$ is a positive integer, use shift right by $n$ ($\gg n$) instead of a division operation. The following code sample illustrates this:

```cpp
int = width n; >> n // prefer this
--> midpoint = width >> 1; // prefer this
```

Only use shift right ($\gg$) when you know you are dealing with a positive value.

Using shorthand for evaluating Boolean conditions

Use this efficient shorthand for evaluating Boolean conditions. Instead of writing the code as in example 1, write the code as in example 2:

```cpp
// Example 1 - Avoid this
if(something_that_evaluates_to_true) {
    return true;
} else {
    return false;
}

// Example 2 - Do this
return(something_that_evaulates_to_true);
```

The resulting compiled code is shorter.
Using int instead of long

In Java, a long is a 64-bit integer. Because the device uses a 32-bit processor, operations run two to four times faster if you use an int instead of a long. However, in the case of date objects, avoid using a lot of date objects to save space. Convert date values into longs.

Managing garbage collection

Avoid calling System.gc() to perform garbage collection. On a full device, this can take two seconds. Let the BlackBerry® Java® Virtual Machine collect garbage automatically.

BlackBerry is a purely Java-based platform that implements a system to periodically free its memory and resources, called garbage collection. All the objects that do not have a reference remain in memory until a garbage collection runs and frees that memory.

Create a String object String test = “some text”; it occupies a small amount of space in memory. You can use the command test = null to clear the object, but memory is still occupied with the string some text. That space in memory is freed only when a garbage collection runs.

Caution:
Do not run garbage collection manually because that ties up resources unnecessarily, for as much as 30 seconds. The BlackBerry smartphone heuristics monitor checks for available free memory and runs garbage collection as needed.

Using static variables for Strings

When you define static fields (also called class fields) of type String, you can increase program speed by using static variables (not final) instead of constants (final). The opposite is true for primitive data types, such as int.

The following code sample shows a String with a static constant:

private static final String x = "example";

For this static constant (denoted by the final keyword), a temporary String instance is created every time you use the constant. The compiler eliminates x and replaces it with the String example in the bytecode, so that the BlackBerry Java Virtual Machine has to perform a Hashtable lookup each time you reference x.

In contrast, for a static variable (no final keyword), the String is created once. The BlackBerry Java Virtual Machine performs the Hashtable lookup only when it initializes x, which makes access faster.

The following code sample shows a String with no final keyword:
private static String x = "example";

Note: The compiler (rapc) automatically marks classes as final that are not extended in an application COD file.

Note: Make variables private to allow the compiler to optimize the generated code.

Avoiding the String(String) constructor

In a BlackBerry Java Application, each quoted string is an instance of the java.lang.String class. Create a String without using the java.lang.String(String) constructor.

The following code sample illustrates this concept:

String str = "abc";

String str = "found " + n + " items"
Avoiding `java.util.Enumeration`

Avoid using `java.util.Enumeration` objects unless you want to hide data (in other words, to return an enumeration of the data instead of the data itself). Asking a Vector or Hashtable for an Enumeration object is slow and creates unnecessary garbage. If another thread can modify the Vector, synchronize the iteration.

```java
for( int i = v.size() - 1; i > = 0; --i ) {
    o = v.elementAt( i );
    ...
}
synchronized( v ) {
    for( int i = v.size() - 1; i > = 0; --i ) {
        o = v.elementAt( i );
        ...
    }
}
```

Using try/catch exception handling and using the `instanceof` operator

The BlackBerry Java Virtual Machine efficiently handles the try/catch exception handling and does not introduce expensive overhead like other Java applications.

Use `instanceof` instead of catching a `ClassCastException` to more efficiently evaluate whether a cast succeeds. The following is an example of using a try/catch block to catch the `ClassCastException`:

```java
try {
    (String)x.whatever();
} catch(ClassCastException e) {
    // something else
}
```

Alternatively, you can use the following instance of operator:

```java
if(x instanceof String) {
```
(String)x.whatever();

} else {
    // something else
}

Use instanceof for greater optimization. Only use the try/catch block when the failure of the cast is an exceptional circumstance. The BlackBerry compiler and the BlackBerry Java Virtual Machine are optimized to perform only one class check in the first block of code. Only one class check is performed on all code in which the cast is run immediately following a branch determined by an instanceof check. Always perform the cast immediately after the branch so that you can then perform an optimization.

**Evaluating conditions using instanceof**

To produce smaller and faster code, if you evaluate a condition using instanceof, do not evaluate explicitly whether the variable is null.

The following code sample illustrates this concept:

```java
if( e instanceof ExampleClass ) { ... }
if( ! ( e instanceof ExampleClass ) ) { ... }
```

**Using longs for unique identifiers**

Use longs instead of Strings for unique constants, such as **GUIDs**, Hashtable keys, and state/context identifiers. To ensure that unique identifiers remain unique across all third-party application developers, use randomly generated keys based on a hash of some String. In the input String, include enough information to make the long unique.

The convert to long function is available in the BlackBerry® Java® Plug-in for Eclipse™. Use the context menu to convert strings in Java code to long versions.

**Increasing speed**

To increase speed, inline the code in critical sections.

**Avoiding recursive calls**

Avoid recursive calls in critical sections of code to increase speed. Use recursion in less critical sections to reduce code size.
Avoiding multiple transactions

The number of requests that your application sends over the network can affect the speed of your application. In general, when performing many individual operations, it is beneficial to open one connection, send the data over the network, and then close the connection instead of opening and closing the connection to send each piece of the data individually.

For example, if an application must send a list of 50 names to a remote server, the “quick” approach may be to open a connection to the server, send one name, and then close the connection, repeating these steps for each of the 50 names. However, it is much faster to open the connection once, send the 50 names, then close the connection. By sending the complete data once rather than the individual pieces of data individually, you reduce the non-trivial amount of overhead associated with establishing a network connection.

This sort of optimization requires consideration both in the design of the BlackBerry smartphone application and the protocol with which to communicate with the server. In general, performing collections of operations in a transaction instead of each individually leads to performance gains.

Using threads

Create a new thread for any lengthy operations, such as network connections. Use background threads, such as java.util.Timer, for listeners or other processes that you want to run in the background when the application starts.

Reusing objects

Reuse objects wherever possible. New objects are costly to create and they generate additional garbage. In comparison, Object obj = new Object() is 43 times more expensive to use than declaring int i=0. Use int i=0 instead of creating an object within a conventional Applet for greater cost efficiency.

Avoid using immutable objects to save on garbage collection from discarded objects left behind after the creation of new, altered immutable objects.

```java
public static String argsToString1( String[] args ) {  
    String s = new String();  
    for ( int i = 0; i < args.length; ++i ) {  
        s = s + args[i];  
        s = s + " ";  
    }  
} 
```
return s;
}

public static String argsToString2(String[] args) {
    StringBuffer s = new StringBuffer();
    int len = args.length;
    for (int i = 0; i < len; ++i) {
        s.append(args[i]).append(' ');
    }
    return s.toString();
}

Using images effectively

If your application relies on images that are in PNG format, ensure you export them in the most compressed form possible in the same format. Some imaging packages save PNG images in a format that you can considerably reduce through supported compression techniques. As an example, you can create a 32 x 32 bit, 24-bit color image using such a product. The size is 29,582 bytes in original form, but 1050 bytes in compressed form.

Exiting applications correctly

Before you invoke System.exit(int status), perform any necessary cleanup, such as removing objects from the runtime store that applications no longer require.

Printing the stack trace

When you debug your application, to view the stack trace, catch a Throwable instance.

The following code sample illustrates this concept:

catch (Throwable t) {
    t.printStackTrace();
}
Using class and 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 you must explicitly initialize.

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):

<table>
<thead>
<tr>
<th>Type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>(byte)0</td>
</tr>
<tr>
<td>short</td>
<td>(short)0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
<tr>
<td>float</td>
<td>Positive 0.0f</td>
</tr>
<tr>
<td>double</td>
<td>Positive 0.0d</td>
</tr>
<tr>
<td>char</td>
<td>null ''</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>class, interface and array types</td>
<td>null</td>
</tr>
</tbody>
</table>

Therefore, writing the following code is redundant and has a negative impact on code size and performance:

```java
private boolean _foo = false;...
```

Instead, use the following code:

```java
private boolean _foo;
```

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

```java
private boolean _foo; // = false
```
Using loop induction variables

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

Avoid the following code:

// calculates 2*i+1 every loop iteration

for( int i = 0; i < n; ++i )
    x[i] = x[2*i+1];

Instead, use the following code:

// introduce a new variable j==2*i+1

for( int i=0, j=1; i < n; ++i, j += 2 )
    x[i] = x[j];

Using objects judiciously

To allow a BlackBerry Java Application to efficiently use memory resources, consider the following questions:

- Given the size of an application, are all of the objects necessary?
- Can your application store any objects that represent primitives, such as long, integer, and boolean, as primitives instead of as objects?
- Are all of the persisted objects necessary?
- Do any instances of Vector and Hashtable exist? Are these instances necessary? If so, how many object handles are not used in the Vector or Hashtable because the initial size is greater than the needed size?
- How many objects does your application create, and then throw away? How many scope-specific objects does your application create?

Reducing the size of compiled code

To reduce the size of compiled code, consider the following guidelines:

- Set appropriate access.
- Avoid creating interfaces.
- Use static strings.
- Import individual classes.
Setting appropriate access

When you create code libraries, you can reduce the size of your compiled code significantly if you use the appropriate access modifiers for fields and methods. The following steps help reduce the size of compiled code:

- Declare fields private whenever possible to enable the compiler to optimize the COD file.
- When possible, use the default (package) access instead of public access (that is, omit the public and protected keywords).

Avoiding creating interfaces

When you create API libraries, avoid creating interfaces unless you foresee multiple implementations of the API. Interfaces produce larger, slower code.

Using static Inner classes

When you use an inner class to hide one class inside another, but you do not need the inner class to reference the outer class object, declare the Inner class static. This suppresses the creation of the reference to the outer class.

Only use a nonstatic Inner class when you need access to data in the outer class from within methods of the inner class. If you are using an inner class only for name scoping, make the Inner class static.

In the following code sample, the first class illustrates when the inner class cannot be static, because it references a non-static member of the outer class: getName(). The second class shown in the following code sample illustrates when the inner class can (and should) be declared static, because it accesses only static members of the outer class.

```java
public class Outer {

    private String _name;

    public String getName() {
        return _name;
    }

    public static void report() {
        System.out.println("report() called");
    }
}
```
private class Inner {
    public void go() {
        System.out.println(Outer.this.getName());
        report();
    }
}

public class Outer {
    private String _name;

    public String getName() {
        return _name;
    }

    public static void report() {
        System.out.println("report() called");
    }

    private static class Inner {
        public void go() {
            System.out.println("I cannot access my name");
            report();
        }
    }
}
Using static strings

If the amount of memory that a BlackBerry smartphone application uses is more important than the speed of the application, consider using a final static String. A final static String uses less memory than a static String that is not final. A static String that is final requires a lookup operation, which can slow down the BlackBerry smartphone application.

If the speed of a BlackBerry smartphone application is more important than the amount of memory the application uses, consider using a static String that is not final. A static String that is not final does not require a lookup operation, but does use more object references and, therefore, uses more memory than a final static String.
1. What limitations do you need to consider when designing an application for a BlackBerry smartphone? Select all that apply.
   A. BlackBerry smartphones cannot use wireless networks.
   B. BlackBerry smartphones can only display one screen at a time.
   C. BlackBerry smartphones have limited battery life.
   D. BlackBerry smartphones do not have touch screens.

2. Name four ways to use classes effectively.

3. Why would you declare an inner class static?

4. When would you use a nonstatic inner class?

5. What is the benefit of marking a class as final?

6. Under what two circumstances must you verify that a public method returning an object does not return null?

7. Under what circumstance should you pass null into a method?

8. What is the correct way to pass null into a constructor to avoid ambiguity?

9. What are four ways to increase speed when writing an application?

10. Why is accessing a static variable faster than a static constant?
Answers

1. B and C

2. To use classes effectively
   A. Extend a class from the java platform
   B. Use native methods
   C. Use profile API classes, substitute your own simplified method, and then profile again
   D. Reduce the number of custom classes

3. To suppress the creation of the reference to the outer class.

4. Only when you need access to data in the outer class from within methods of the inner class.

5. To enable the compiler to generate faster code.

6. You must verify that a public method returning an object does not return null when
   • a null is expected during normal program operation
   • the javadoc @return parameter states that null is a possible return value

7. Only if the API reference documentation explicitly states that the method supports it.

8. Use `new someObject ((Object)null);`

9. To increase application speed
   A. inline code in critical sections
   B. reduce the code size by placing duplicate code in its own method
   C. use int instead of long
   D. avoid using a lot of date objects
Chapter 3

E. convert date objects into longs
F. avoid recursive calls in critical sections of code
G. use direct lookup instead of a table search

10. A static constant is a temporary string instance created each time the static constant is called and the BlackBerry Java Virtual machine has to perform a Hashable lookup each time. A static variable is created once and a Hashable look up is only done once, which makes access faster.
Using the BlackBerry IDE profiler tool

The BlackBerry Java Plug-in for Eclipse comprises a series of development tools that you can use to test and create applications for the BlackBerry smartphone.

The full environment consists of the following development tools:
- Java ME compiler
- Editor
- Debugger
- BlackBerry IDE profiler tool
- BlackBerry® MDS simulator

Optimizing code using the BlackBerry IDE profiler tool

The BlackBerry IDE profiler tool displays the percentage of time spent in each code area to the current point of execution.

To optimize your code, use the BlackBerry IDE profiler tool of the BlackBerry Java Plug-in for Eclipse.

To optimize code using the BlackBerry IDE profiler tool, you must do the following:
- generate the profile data
- set the profile options
- view the profile data
- save the contents to a file

To improve the reliability of results when you run the BlackBerry IDE profiler tool, exit other Microsoft® Windows applications.

Generate BlackBerry IDE profiler tool data

Use the BlackBerry IDE profiler tool to isolate and address coding bottlenecks. The BlackBerry IDE profiler tool shows the percentage of time spent in each code area to the current point of execution. You can use the BlackBerry IDE profiler tool to profile the efficiency of code sections.

1. Specify a breakpoint at the beginning and end of the section you want to profile.
2. On the Run menu, click Debug As > BlackBerry Simulator.
3. In the BlackBerry Smartphone Simulator, run the application. The debugging tool pauses the application when it reaches the first breakpoint.

4. In Eclipse®, on the Window menu, click Show View > Other.

5. Double-click the BlackBerry folder.

6. Click BlackBerry Profiler View.

7. Click OK.

8. In the BlackBerry Profiler View, click Setup Options.

9. In the Profiler Options dialog box, click the type of method attribution and the type of information that you want to profile.

<table>
<thead>
<tr>
<th>Drop-down list</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method attribution</td>
<td>Cumulative</td>
<td>The BlackBerry IDE profiler tool calculates the time spent executing bytecode in a method and all methods invoked by that method.</td>
</tr>
<tr>
<td></td>
<td>In method only</td>
<td>The BlackBerry IDE profiler tool calculates the time spent executing bytecode in the specified method only. The timer stops when a call is made to another method.</td>
</tr>
</tbody>
</table>
10. Click **OK**.

11. Press **F8** to resume program execution. The debugging tool pauses the application when it reaches the second breakpoint.

12. In the BlackBerry Profiler View, click **Refresh**. The BlackBerry IDE profiler tool retrieves the data. BlackBerry IDE profiler tool data is not cleared, so running a program again adds to the data.

13. In the BlackBerry Profiler View, click **Save** to save the contents of the profiler pane to a CSV file.

14. Repeat steps 1 through 11, setting breakpoints closer together until they converge on the bottleneck.

## Drop-down list

<table>
<thead>
<tr>
<th>Drop-down list</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>What to profile</td>
<td>Time</td>
<td>The BlackBerry IDE profiler tool measures execution time (measured in clock ticks).</td>
</tr>
<tr>
<td></td>
<td>Number of objects created</td>
<td>The BlackBerry IDE profiler tool calculates the number of objects created.</td>
</tr>
<tr>
<td></td>
<td>Size of objects created</td>
<td>The BlackBerry IDE profiler tool measures the size of objects created.</td>
</tr>
<tr>
<td></td>
<td>Number of objects committed</td>
<td>The BlackBerry IDE profiler tool calculates the number of committed objects.</td>
</tr>
<tr>
<td></td>
<td>Size of objects committed</td>
<td>The BlackBerry IDE profiler tool measures the size of committed objects.</td>
</tr>
<tr>
<td></td>
<td>Number of objects moved to RAM</td>
<td>The BlackBerry IDE profiler tool calculates the number of objects moved into memory.</td>
</tr>
<tr>
<td></td>
<td>Size of objects moved to RAM</td>
<td>The BlackBerry IDE profiler tool measures the size of objects moved into memory.</td>
</tr>
<tr>
<td></td>
<td>User counting</td>
<td>The BlackBerry IDE profiler tool calculates user counting.</td>
</tr>
</tbody>
</table>
View BlackBerry IDE profiler tool data

1. On the Window menu, click Show View > Other.
2. Double-click the BlackBerry folder.
3. Click BlackBerry Profiler View.
4. Click OK.

There are three views you can use to display the profile efficiency information.

<table>
<thead>
<tr>
<th>View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>The Summary view displays general statistics about the system and the garbage collector. The Summary view displays the percentage of time that the BlackBerry Java Virtual Machine has spent idle, executing code, and performing quick and full garbage collection. The Percent column displays the percent of total BlackBerry Java Virtual Machine running time, including idle and collection time.</td>
</tr>
<tr>
<td>Methods</td>
<td>The Methods view displays a list of modules, sorted either by the information that you are profiling or by the number of times each item has been executed.</td>
</tr>
<tr>
<td></td>
<td>- Expand the All item to see a list of all methods.</td>
</tr>
<tr>
<td></td>
<td>- Expand a specific module to see only its methods.</td>
</tr>
<tr>
<td></td>
<td>- Right-click a method, and then click Profile Source to view source lines in the Source view.</td>
</tr>
<tr>
<td></td>
<td>- Right-click a method, and then click Show Source to view source code in the Edit window.</td>
</tr>
<tr>
<td></td>
<td>In this view, the Percent column displays the percentage of BlackBerry Java Virtual Machine execution time only, not including idle and garbage collection time.</td>
</tr>
</tbody>
</table>
The Summary view displays percentage spent in Idle, Code Execution and Garbage Collection stages. This screen shows the Summary view in the BlackBerry IDE profiler tool.

<table>
<thead>
<tr>
<th>View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>The Source view displays the source lines of a single method. You can navigate through the methods that call, and are called by, that method. The Source view displays the following items:</td>
</tr>
<tr>
<td></td>
<td>• A list of callers to the method, including the number of times that they make the call and the total time spent on these calls.</td>
</tr>
<tr>
<td></td>
<td>• A list of source lines for the method and the total time spent on these lines.</td>
</tr>
<tr>
<td></td>
<td>You can expand a source line to show individual bytecode.</td>
</tr>
<tr>
<td></td>
<td>You can further expand any bytecode that corresponds to a method invocation to show the target (or targets) of the method invocation.</td>
</tr>
<tr>
<td></td>
<td>To view the source code in the Edit window, right-click a line, and then select Show Source.</td>
</tr>
<tr>
<td></td>
<td>To follow the history of methods that you have visited in the Source view, click Back and Forward.</td>
</tr>
</tbody>
</table>

The Summary view displays percentage spent in Idle, Code Execution and Garbage Collection stages. This screen shows the Summary view in the BlackBerry IDE profiler tool.

**Figure 4.1** Summary view in the BlackBerry IDE profiler tool
Chapter 3

The Method view provides a detailed view of all running packages. Under options you can choose what to profile. Default settings are in ticks (time), but you can also view the size and number of objects. This screen shows the Method view in the BlackBerry IDE profiler tool.

![Method view in the BlackBerry IDE profiler tool](image)

Figure 4.2 Method view in the BlackBerry IDE profiler tool

The Source view displays the source lines of a single method. You can navigate through the methods that call, and are called by, that method. This screen shows the Source view in the BlackBerry IDE profiler tool.

![Source view in the BlackBerry IDE profiler tool](image)

Figure 4.3 Source view in the BlackBerry IDE profiler tool
Save the contents of the BlackBerry Profiler View to a CSV file

1. In the BlackBerry Profiler View, click Refresh.
2. Click Save data to a CSV file.
3. Select a location, and then type a name for the file.
4. Click Save.

The contents of the pane are saved to a CSV file. Contents are saved in the same format in which they appear.
1. What development tools are included in the BlackBerry Java Plug-in for Eclipse?

2. How does the BlackBerry IDE profiler tool address bottlenecks?

3. List the steps you must take to optimize code using the BlackBerry IDE profiler tool?

4. Name the three views that you can use to view the profile data after you have run the BlackBerry IDE profiler tool.

5. Which profile view can display a list of modules that have been sorted by the number of times each item has been executed?
Answers

1. The following tools are part of the BlackBerry Java Plug-in for Eclipse:
   A. Java ME compiler
   B. Editor
   C. Debugger
   D. BlackBerry IDE profiler tool
   E. BlackBerry® MDS simulator

2. The BlackBerry IDE profiler tool shows the percentage of time spent in each code area to the current point of execution.

3. To optimize code using the BlackBerry IDE profiler tool you must
   A. set the profile options
   B. generate the profile data
   C. view the profile data
   D. save the contents to a file

4. Summary view, Methods view, Source view

5. Methods view
Identifying memory leaks

The BlackBerry Java Virtual Machine relies on a garbage collection task that locates and deletes objects that are no longer needed by an application. At each pass, the garbage collector starts at the root nodes where classes exist that persist for the duration of application life, then traverses each referenced node. During traversal, all actively referenced nodes are tracked, and all nodes that are no longer referenced are deleted, with associated memory passed back to the BlackBerry Java Virtual Machine.

If the high persistence class fails to clear its reference to the limited persistence class after dismissal, memory is not reclaimed until the high persistence class goes out of scope. Memory can leak in situations where a high persistence class creates one, or more, limited persistence classes. For example, user-interface widgets (such as forms or controls) that are displayed then dismissed by users.

Common places to find the cause of memory leaks are in collection classes such as Vectors, list and Hashables, especially if the class is declared static and exists for the life of the application. Another common area occurs when a class is registered as an event listener, yet is never unregistered before the class goes out of scope.

The BlackBerry Java Plug-in for Eclipse provides a Memory Statistics tool and an Object tool that can display statistics on the number of objects and bytes that are used for object handles, RAM and flash memory. With memory statistics you set a few breakpoints, run your application, refresh and retain the memory statistics, run the application to the next breakpoint, and then refresh to compare memory statistics to the previous breakpoint.

Finding memory leaks

Memory leaks can cause the BlackBerry Java Virtual Machine to run out of flash memory, and the BlackBerry smartphone to reset. Memory leaks are created when an application maintains a reference to an object that is no longer needed. Even if you design your code to delete unneeded references, other references in the system can prevent an object from being deleted.

Memory leaks are most common in the following:

- data structure
- local variables
- runtime store
- listeners
Some possible symptoms of memory leaks include the following:

- Hourglass appears often (because the BlackBerry smartphone is trying to do garbage collection).
- Email messages are automatically deleted. When the BlackBerry smartphone is running low on available memory it asks applications that use the LMM to reduce memory usage, which they sometimes do by deleting unused files.
- A low Free File Number.

## Viewing statistics to locate memory leaks

Use the Memory Statistics tool with the Objects tool to find object leaks. The Memory Statistics tool retrieves information on application memory usage and identifies the number of objects in memory. The Objects tool displays detailed information for each object.

The Memory Statistics tool displays the statistics on the number of objects and bytes in use for object handles, RAM, and flash memory. This screen shows the Memory Statistics pane.

![Figure 4.4 Memory Statistics pane](image)

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># objects</td>
<td>displays the number of objects that are currently in memory</td>
</tr>
<tr>
<td>Bytes in use</td>
<td>displays the amount of memory that is used by Java objects</td>
</tr>
<tr>
<td>Allocated</td>
<td>displays the total memory that is allocated to the BlackBerry Java Virtual Machine</td>
</tr>
<tr>
<td>Free</td>
<td>displays the memory that is available</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object handles</td>
<td>displays the number of object handles in memory</td>
</tr>
</tbody>
</table>
View statistics to locate memory leaks

1. On the Window menu, click Show View > Other.
2. Double-click the BlackBerry folder.
3. Click BlackBerry Memory Statistics View.
4. Click OK.

Find a memory leak

1. Set two or more breakpoints in your code.
2. On the Run menu, click Debug As > BlackBerry Simulator Application. The application runs to the first breakpoint.
3. On the Window menu, click Show View > Other.
4. Double-click the BlackBerry folder.
5. Click BlackBerry Memory Statistics View.
6. Click OK.
7. In the main window, on the View menu, click Memory Statistics.
8. Click Refresh.
9. Click Snapshot.
10. Press F8 to resume running of the application. The application runs to the second breakpoint.
11. Click Refresh.
12. Click Compare.

<table>
<thead>
<tr>
<th>Row</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>displays the RAM usage</td>
</tr>
<tr>
<td>Transient objects (flash)</td>
<td>displays the number of transient objects in flash memory</td>
</tr>
<tr>
<td>Persistent objects (flash)</td>
<td>displays the number of persistent objects in flash memory</td>
</tr>
<tr>
<td>Code modules (flash)</td>
<td>displays the number of code modules in flash memory</td>
</tr>
<tr>
<td>Flash</td>
<td>displays the sum of the previous three rows</td>
</tr>
</tbody>
</table>
13. Repeat steps 1 through 10, setting breakpoints closer together until they converge on the object leak.

Displaying objects in memory to locate memory leaks

1. On the Run menu, click Debug As > BlackBerry Simulator Application. The application runs to the first breakpoint.

2. In the Debug view, click Suspend.

3. On the Window menu, click Show View > Other.

4. Double-click the BlackBerry folder.

5. Click BlackBerry Objects View.

6. Click OK.

7. Click Garbage collect.

8. Click Snapshot.

9. Press F8 to resume running of the program.

10. Perform the operations in the application that do not increase the number of reachable objects. For example, create a new contact, and then delete it.

11. In the Debug view, click Suspend.

12. Click Garbage collect.

13. Click Compare. The objects pane displays the number of objects that have been deleted and added since the previous snapshot. If the number of objects added is not the same as the number of objects deleted, this can indicate an object leak.

14. To identify new objects, use the Type, Process, and Location filter types available through the Filter drop-down list at the top of the objects pane.

15. To save the contents of the objects pane to a CSV file, click Save.
1. Which of the following can you examine to find the cause of a memory leak?
   A. Tables
   B. Longs
   C. Vectors
   D. Lists

2. One method of detecting the source of a memory leak is to make the leak worse. True or false?

3. List three symptoms of a memory leak?
Answers

1. C

2. True

3. The hourglass appears often, email messages are automatically deleted, and the low Free File Number appears.
Chapter 3

Summary

Develop applications with the BlackBerry smartphone user and the BlackBerry smartphone limitations in mind. The BlackBerry smartphone and the user both have different requirements than a computer or computer user.

To ensure that your BlackBerry smartphone applications run effectively for the user and efficiently for the device, use optimal coding techniques. Gain efficiency in coding practices and in the application by reusing existing code. By leveraging existing code, the application becomes less costly and ensures consistency with other existing applications. Remember these core development guidelines:

- Use or extend existing UI components where possible so that your application can inherit the default behavior of the component.
- Follow the standard navigation model as closely as possible so that BlackBerry smartphone users can make full use of the keyboard and trackball.
- Make all actions available from the menu. Verify that the actions available in the menu are relevant to users’ current context.

Inefficient code results in costly, ineffective applications that are not useful to the BlackBerry smartphone user. You can use tools such as the BlackBerry IDE profiler tool from the BlackBerry Java Plug-in for Eclipse to help you optimize your code.

Memory leaks are another key issue to resolve to increase the efficiency of your code and application. The BlackBerry smartphone has limited battery power, and it is important to the user that applications do not drain the battery, or consume memory. Watch for memory and object leaks. Use the Memory Statistics and Object tools to locate and identify the sources of memory leaks. These tools can help you ensure the quality of your final application.
1. Why is it important to optimize code, and name three benefits of code optimization?

2. What is the difference between a direct lookup and a table search?

3. Describe how garbage collection works and which optimal coding techniques can help reduce the need for more garbage collection.

4. Briefly describe five coding techniques that can help optimize application code.

5. Name four areas where memory leaks are most commonly found.

6. Describe how you can use the Memory Statistics tool to isolate memory leaks.