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 5

Introduction to threading and background processing

Objectives

- Describe multithreaded applications
- Describe the Main Event thread
- Describe background applications
- Describe how to create and use threads
- Describe the dangers of multithreaded applications

This chapter discusses multithreaded applications and outlines how to use multithreading for Java® development on the BlackBerry® smartphone.
What are multithreaded applications?

Traditionally, applications are typically run in a single thread of execution. This means that each task the application must perform during the course of its execution happens in a sequence, one task at a time. Each programmatic task is executed as it is encountered.

Multithreaded applications can create multiple threads of execution (known as spawning a thread) that perform many tasks concurrently. An application that performs many tasks at once rather than placing them in queue is more efficient and effective. Many programmatic tasks are processed and their results are used immediately or stored for later.

Consider the following diagram. On the left is an application performing three tasks in a single thread of execution: retrieving stock information, downloading traffic reports, and querying sales information. The application must perform each task in succession and cannot start the next task before finishing the previous one.

Figure 7.1 Single and multiple threads of execution

On the right is an application performing the same tasks with multiple threads of execution. The application places each task in its own thread of execution simultaneously and returns the data when the task is complete while the main thread of execution (the main application) continues to operate. The application can wait for data from one of these threads, perform another task while it waits, or store the information returned by the threads for later use.
Multithreaded applications provide the following benefits:

- improved application responsiveness
- efficient multiprocessor use
- efficient system resource usage

**Improved application responsiveness**

The main advantage of multithreaded applications is application responsiveness. An application that is able to separate tasks into individual threads of execution are much more responsive than applications that execute along one main thread. Multiple threads means you can perform processing on multiple tasks in parallel and use the outcomes as necessary. An application with a single thread of execution must wait for each task to be completed, and this can often result in a slow or unresponsive application.

**Efficient multiprocessor use**

Most processors used in modern computing devices are multicore processors. The core of a processor is the device that does the actual processing of an application. Traditionally, a processor contained a single core that did all of the processing work. A modern multicore processor has two or four cores to perform processing operations.

When additional processing cores are available, multithreaded applications see improved performance because the operating system schedules threads for the number of available processing cores. The performance of a multithreaded application will improve accordingly. Single threaded applications cannot use multicore processors efficiently.

**Efficient system resource usage**

Multithreaded applications can be more efficient users of system resources. Their ability to perform many concurrent operations means they can better utilize the resources available to them. Applications running in a single thread of execution must resolve processing tasks one at a time while multithreaded applications can resolve many operations concurrently.

**Multithreading and the BlackBerry smartphone**

The BlackBerry operating system supports multithreading. This means many applications and processes can run concurrently on the BlackBerry smartphone at the same time. Applications can use background threads to manage processor-intensive tasks or network communications so that they do not affect main threads of execution.
Multithreading is especially useful on the BlackBerry smartphone for blocking operations and during lengthy processing operations.

Blocking operations: A separate thread of execution should always be created for any operation that can block application processing. This ensures the UI of the application is not disrupted during the operation. Failure to create a separate thread of execution can block the application’s event thread while data is being sent or received, locking the application UI. One example of a common blocking operation is network communications.

Lengthy processing operations: Encapsulate lengthy operations in their own thread of execution. This ensures the UI of the application is not disrupted while the operation resolves itself.

Threads should also be created for listeners or other processes that run in the background of an application.
What is the Main Event thread?

Every Java® ME application developed for the BlackBerry smartphone contains one thread of execution called the Main Event thread. The Main Event thread is responsible for all updates to the currently displayed GUI of an application. It is also responsible for handling user input, such as scrolling the trackball or typing on the keyboard. This section describes the Main Event thread and its uses in your BlackBerry smartphone application.

The Main Event

All applications have one central line of execution from which the rest of the program processes flow. In a Java application this central point is referred to as the Main Event thread. The Main Event thread is created when the Application.enterEventDispatcher() method is called.

The thread that calls Application.enterEventDispatcher() becomes the application event-dispatching thread. The event-dispatching thread executes all drawing and event handling code in the application. Although this method can be called from the main() method when first setting up the application, these duties can be delegated to any thread.

The following example demonstrates a simple use of Application.eventEventDispatcher().

```java
public class SimpleApp extends UiApplication
{
    public static void main(String[] args)
    {
        MyApp app = new MyApp();
        app.enterEventDispatcher();
    }
}
```

This example demonstrates the basic framework for the execution of a simple GUI application. This sample demonstrates the following items:

- Class inherited from UiApplication—Any GUI application must be inherited from the UiApplication class which itself is inherited from the Application class. Part of this inheritance is the inclusion of the enterEventDispatcher() method.
• Use of `enterEventDispatcher()` method—The `enterEventDispatcher()` method is called in the `main()` method to assign event dispatching duties to this thread. The method is called on an instance of the class instantiated in the `main()` method.

Using the Main Event thread

The primary purpose of the Main Event thread is to control the overall execution of the application. When used in conjunction with multithreaded applications, the Main Event thread serves as the main launching and collapsing point of all threads of execution in the complete application. Consider the following diagram.

*Figure 7.2 Main Event thread and multiple threads of execution*

This diagram illustrates the Main Event thread and threads spawned from it. The following describes the threads viewed from left to right:

- A thread that is spawned at the beginning of the application and exists for the lifetime of the application. During application execution this thread performs a task ancilliary to the application itself and then exits at the end of the application. In a real-world application the ancillary task can be data retrieval or some other support task the application uses but does not rely on for the overall functionality of the program.

- A thread that is spawned and that itself spawns another thread over the lifetime of the first thread’s execution. Theoretically, threads can spin off like this forever, although the resource limits of the actual device will limit how many threads of execution can be tolerated before a fault condition exists. As illustrated in the diagram, most threads in a multithreaded application are both parents and children. A well-programmed application will gracefully exit all of these threads before exiting at the end of the Main Event thread.

- A thread that is spawned, performs some sort of execution, and then spawns a new thread as a condition of the original thread’s exit. This is an example of a typical data retrieval thread that retrieves information, exits, and passes the data set to a new thread that does something with it. Data retrieval through push notifications, however, is handled as shown in the first thread illustrated.
What are background applications?

One of the unique features of the BlackBerry operating system is the ability to run background applications. A background application is one that runs in a separate process that does not interfere with the other applications and processes running on the smartphone. Although a background application can theoretically perform any number of tasks, it is commonly use to listen for data being pushed to the device.

There are two primary means to move data to a BlackBerry smartphone: push and pull. In a pull situation, a device actively polls a server for data at predetermined intervals. If new data is available, it is downloaded to the device. In a push situation, a device registers with a service. If new data from the service is available for the device, it is pushed to the device without notification and integrated with any existing data.

To make a device ready to accept this information, small applications must run in the background to receive this information and make use of it. These background applications run as self-contained programs in separate threads of execution. When the background applications receive pushed data they will perform one of two tasks: store the data so it can be used the next time the supported application is run, or they will integrate it immediately into the user experience if the supported application is currently running. Consider the following two diagrams.

*Figure 7.3 Background application example*
In the diagram to the left is an illustration of a server pushing data to an application that is running in the background of a BlackBerry smartphone. Because the main application is not currently running, the background application will store the information until the user runs it. In the diagram to the right, the user has started the main application and the background application makes the new data available.

The important thing to remember about background applications is that because they are running in their own process, they do not interfere with the ongoing user experience on the device. None of this functionality would be possible in a single threaded environment.
Creating and using threads

This section focuses on how to create and use threads in Java ME applications developed for the BlackBerry smartphone. This section discusses how to create classes used in multithreaded applications and then use those classes in an application.

Creating runnable classes

The Java APIs provide a class and interface used to create runnable classes. The APIs provide two mechanisms because Java supports a single inheritance model. In a single inheritance model, any new class can only be inherited from a single parent class. But for developers who wish to implement multiple inheritance in their classes, the Java APIs provide either the Thread class or the Runnable interface.

Use of the Runnable interface is one method used to create a runnable class. By implementing this interface, a developer is free to inherit functionality from another fully formed class. An interface differs from a class in that it only provides a framework that any classes that implement it must follow. This gives the developer much more flexibility in the classes they create. The following example demonstrates a class that implements Runnable.

```java
import java.lang.*;

public class DataListener implements Runnable
{
    Thread dataThread;

    public void run()
    {
    }
}
```

This example comprises the following elements:

- **Import Statement**—Any class that implements the Runnable interface must import the java.lang package. This package contains the definition of the Runnable interface.
- **Class Statement**—The class definition statement tells the Java compiler the class implements the Runnable. This means the class definition must fully conform to the interface before it compiles successfully.
- **run() Method**—The run() method is where the threaded work of a class that implements the Runnable interface happens. A class that implements the Runnable interface must define this method or else it will not compile correctly.
Thread variable declaration—The class itself does not have any inherent properties of a thread, only a framework that defines an empty `run()` method. An object variable of type `Thread` is created to perform the actual threaded work of the class. The variable is declared to ensure visibility to other methods in the class. If the variable were declared inside of the `run()` method it would not be available for use outside of the method itself.

This is the basic framework of a runnable class that implements the `Runnable` interface. The executable code is placed in the `run()` method and supported by any other methods or subroutines in the class itself.

You can also use the `Thread` class to create a runnable class. Using the `Thread` class to create a new class may be a bit easier to do but you are limited by Java’s single inheritance model. When you use the `Thread` class, the class can only use interfaces to lend additional functionality. The following example demonstrates a class that extends the `Thread` class.

```java
import java.lang.*;

public class DataListener extends Thread
{

public void run()
{

}

}
```

This example comprises the following elements:

- **Import Statement**—Any class that extends the `Thread` class must import the `java.lang` package. This package contains the definition of the `Thread` class.
- **Class Statement**—The class definition statement tells the Java compiler it inherits from the `Thread` class. This means the class definition includes the variables, subroutines, and methods provided by the `Thread` class.
- **run() Method**—The `run()` method is where the threaded work of a class that extends the `Thread` class happens.

Unlike the class that implements the `Runnable` interface, this example does not need to use an instance of the `Thread` class as an object variable because the class is already inherently a `Thread` object by inheritance. Otherwise, this is the basic framework of a runnable class that extends the `Thread` class. The executable code is placed in the `run()` method and supported by any other methods or subroutines in the class itself.
Creating and starting threads

After you develop a runnable class, you can use it in your applications. To make use of the class you must first instantiate it as an object. To instantiate an object, complete the following steps:

1. Ensure the class to be used is within the scope of the current class—If both classes reside inside of the same package, no additional action is necessary. If the class to be used is not within the scope of the current class, use an import statement to bring the desired package within scope.

2. Declare an object variable—Declare an object variable of the appropriate type with the proper visibility. If the object requires visibility outside of a particular method, declare it as a Global variable. If it does not require that visibility, declare it locally in the method itself. Always declare variables in as short a scope as possible.

3. Instantiate the object—Assign the object variable to a new instance of the threaded class.

If the variable declaration and instantiation processes from steps 2 and 3 are combined, then the statement would look like this:

```java
DataListener weatherForecast = new DataListener();
```

Typically the declaration and instantiation processes take up multiple lines of code.

After a threaded class is properly instantiated into an object, it is ready for execution. To begin execution call the `start()` method. This method is defined for all runnable classes and requires no further programming work on the developer’s part. A call to start the thread instantiated in this example will look like this:

```java
weatherForecast.start();
```

Suspending threads

After a thread of execution is stopped, it cannot be restarted. Threads of execution can be suspended and resumed. There are multiple methods for suspending thread execution:

- Thread suspension—Threads can be manually suspended and resumed. This is a useful method for working with threads that depend on user interaction or other situations where suspension is reliant on a variable timeframe. To suspend the thread, call the `suspend()` method on the threaded object. This will stop execution on the thread until the `resume()` method is called.

- Thread sleeping—Threads can be suspended for a set amount of time using the `sleep()` method and passing the amount of time in milliseconds to suspend the thread. Putting a thread to sleep suspends operation for a set amount of time. This is the preferred method for suspending thread execution because it guarantees the thread will eventually exit and not put an indefinite lock on system resources.
• Wait and notify—The `wait()` and `notify()` methods work in conjunction to suspend and wake threads that require access to a group of objects. The `wait()` method tells a thread to release its lock on an object and suspend execution until another thread calls the `notify()` method. The `notify()` method wakes the waiting thread and processing continues.

**Prioritizing threads**

Threads can be prioritized to ensure they are executed before other threads spawned by the application. This prioritization applies only in the context of the current application. Regardless, this can be a useful technique for ensuring that some threads are executed before others.

Thread prioritization is accomplished using the `setPriority()` method of the threaded object and passing a numeric value that represents the new priority. Thread priority works on an ascending scale.

Threads inherit the priority of the thread they were spawned from, so you may not need to set the priority of the thread. Use the `getPriority()` method to retrieve the priority of the thread.

**Stopping threads**

Occasionally a thread will need to be stopped before it exits naturally. Although any threaded object provides a `stop()` method for ending execution, this method can sometimes cause problems in applications. A better solution is to fashion code that interrupts the execution of the thread.
Dangers of multithreaded applications

Although multithreading is an efficient and practical way to make the maximum use of system resources, there are some dangers inherent in its use. Be aware of the following items when you are developing multithreaded applications:

- A `TooManyThreadsError` exception will be returned by an application that has reached either the application or operating system maximums for thread creation. This exception should be handled during any thread creation code. Refer to the relevant J2ME and BlackBerry smartphone operating system documentation for current maximums.

- Long-running threads (such as those found in background applications) should occasionally be suspended so their resources can be freed up for other threads of execution.

- All multithreaded applications should be coded to properly handle all anticipated exceptions.

- All threads of execution spawned by applications that are not designed to run in the background should be ended by the time the application itself ends. Although Java provides a robust Garbage Collection service for freeing memory and system resources, it is good practice to manually perform these tasks and not rely on the Garbage Collection service to clean up after an application.

- To end a thread of execution use coded methods instead of the `stop()` method. The `stop()` method can cause damage to objects associated with the thread of execution, and even though the thread may exit, other parts of the application might still use these objects. Corrupted objects can result in arbitrary behaviors that lead to errors in the application.
Multithreaded applications have the ability to create multiple threads of execution that perform many tasks in a concurrent fashion. The BlackBerry smartphone operating system supports multithreading so developers can create responsive applications that best exploit the unique features of the operating environment.

Multithreaded applications provide the following key advantages:

- improved application responsiveness
- efficient multiprocessor use
- efficient system resource usage

A background application is one that runs in a separate process that does not interfere with the other applications and processes running on the smartphone. Although a background application could theoretically perform any number of tasks, a common use for them is to listen for data being pushed to the device.

A runnable class can be created from the `Thread` class or the `Runnable` interface. The `Runnable` interface is preferable because it allows a developer to further inherit from another class.

Runnable classes expose the `run()` method to perform the threaded work.

Thread execution can be frozen through either thread suspension or thread sleeping.

It is preferable to stop thread execution through a coded method other than explicitly calling the `stop()` method.

There are many risks inherent in using multithreaded applications. Be aware of these risks during the development process.
Review Questions

1. What is the disadvantage of applications that run in a single thread of execution?
   A. Tasks must be performed sequentially.
   B. Tasks must be performed concurrently.
   C. Tasks must be performed synchronously.
   D. Tasks must be performed asynchronously.

2. Which of the following are advantages of a multithreaded application?
   A. Improved device responsiveness.
   B. Improved application responsiveness.
   C. Efficient multiprocessor usage.
   D. Efficient system resource usage.

3. Which of the following represents an instance where multithreaded applications are recom-
   mended for BlackBerry smartphones?
   A. Lengthy processing operations.
   B. Network communications.
   C. Background applications.
   D. Pull notifications.

4. What is a possible outcome of creating an application with a single thread of execution instead
   of a multithreaded one?
   A. Improved application responsiveness as tasks resolve themselves.
   B. Efficient system resource usage.
   C. Improved code reuse.
   D. Lack of application responsiveness as tasks resolve themselves.
5. What does the Main Event thread represent?
   A. The secondary thread of application execution.
   B. The main thread of application execution.
   C. The primary event-handling thread.
   D. The secondary thread of event handling.

6. What method in all executable classes represents the Main Event thread in Java code?
   A. Application.mainEventStart()
   B. static()
   C. Application.enterEventDispatcher
   D. first()

7. What is a common use of a background application?
   A. To listen for events on the local device.
   B. To listen for data pushed from network servers.
   C. To listen for data change on the local device.
   D. To listen for keyboard events.

8. What does the use of background applications prevent?
   A. Tasks that interfere with the ongoing user experience on the device.
   B. Tasks that use pull notifications to retrieve network data.
   C. Tasks that use push notifications to retrieve network data.
   D. None of the above.

9. What type of inheritance does Java support?
   A. Dual
   B. Mutual
   C. Manual
   D. Single
10. Which of the following classes or interfaces are used to create runnable classes?
   A. Thread
   B. Running
   C. String
   D. Runnable

11. Which package contains the definitions of the classes or interfaces used to create runnable classes?
   A. java.io
   B. java.lang
   C. java.string
   D. java.thread

12. What method do runnable classes use to perform their work?
   A. go()
   B. start()
   C. run()
   D. running()

13. Which of the following are steps to using runnable classes in an application?
   A. Ensure the threaded class is in scope.
   B. Declare an object variable.
   C. Instantiate the object.
   D. All of the above.
14. What are two methods for temporarily interrupting thread execution?
   A. `sleep()`
   B. `suspend()`
   C. `stop()`
   D. `slow()`

15. What is the preferred method for ending thread execution?
   A. Using the `stop()` method.
   B. Using code to gracefully end thread execution.
   C. Using code to suspend a thread indefinitely.
   D. Using the `suspend()` method.