A Cup of Java Code

By Peg MacPhail

This article describes a sample Java application exploring Java features such as threads, Abstract Windowing Toolkit (AWT), and the new Event model. The entire code can be found in the Bank Loan Sample Program in this issue.

Click here to go to Sample Program Listings

This sample bank-loan application uses some Java's features, especially threads, that might be useful for a real application. The sample code in its entirety is available in this issue.

Application Background

This program began as a threads programming exercise. Many thread samples are complex number-crunching programs that are extremely difficult to understand. This sample represents a simplistic, rather than realistic, bank-loan application.

Since this program is a Java application, not an applet, it runs independently, outside a browser or an applet viewer. Although Java applications still run on the Java Virtual Machine, they do not have the same security restrictions as an applet. Applications have a main method—not an init method—in the first class that is instantiated to run the program.

The major classes developed for this application are as follows:

- `InputManager`: Handles the user interface
- `BankJob`: Contains the basic context of the loan application, such as how the accounts are named
- `Worker`: Runs on its own thread and pulls `WorkItems` from its own queue
- `WorkItem`: Handles all the actual work it represents since the sample is object-oriented
- `Approver`: Runs its simple algorithm to approve or refuse the loan after all of the transaction work is done
- `Account`: Represents the type of account, such as savings or loan
- `Transaction`: Represents actions associated with each account

How Threads Are Used

Since the `InputManager` runs in its own thread, the main application work does not block the user from doing other work. Although this sample does not take full advantage of this independent thread, your "real" code could. For example, a real program could input loan application request

```
class Worker implements Runnable {
  .
  .
  .
  public void run() {
    while (!done) {
      WorkItem work = getBottomWork();       // get WorkItem
      If ( ( work == null) && (!done) ) {    // if none, steal work
        work = stealWork();
      }
      if ( work != null) {
        work.setWorkMaster(this);
        work.run();                          // make work do itself
        job.ender.decreaseRC();              // decrease release count
                                             // for "final" WorkItem
      }
      else {
        job.threads (id).yield();
      }
    }
  }
  .
  .
  .

class Approver {
  .
  .
  .
  public boolean approveIt() {
    for (int i=0;i<job.NUMTHREADS;i++) {       // Create workers
      job.workers (i) = new Worker (i, job);
    }
    // Create ender (cleanup) WorkItem with release count = 1 so
    // that it will not post itself to the work queue before we have had a
    // chance to add the real WorkItems to the work queue.  Each
    // real WorkItem will add 1 to the release count for our ender
    // WorkItem.
    job.ender = new FinishWork(job.workers*_0*y, 1, job);
    for (int i=0;i<job.COUNT;i++) {
      new ReadAccount(job.workers (0, i, 0, job);
    }
    job.ender.decreaseRC();           // Set release count back
    for (int i=0;i<job.NUMTHREADS;i++) {    // Create threads
      job.threads (i)  = new Thread (job.workers (i);
      job.threads (i).setName("worker" + String.valueOf(i));
      job.threads (i).start();
    }
    for (int i=0;i<job.NUMTHREADS;i++) {   // Wait for threads to end
      try {
        job.threads (i).join();
      }
      catch (InterruptedException e) {
        job.IM.appendText("Thread interrupt" + e);
      }
    }
  .
  .
  .
  }
  .
  .
  .
}
```

Figure 1. Threads example

objects into a work queue for the `Approver` object(s) to handle.

This sample uses threads with the `Worker` objects. We used the Blumofe and Leiserson algorithm[1] to schedule the thread work. Each `Worker` has its own work list of `WorkItems`, such as reading an account from persistent storage and processing transactions not reflected in the current balance for the account.

To vary parameters with these threads, we built several tunable parameters into this application. Part of the work scheduling algorithm includes a *work number*—the number of transactions a `WorkItem` is willing to do at one time. If this number is less than the number of transactions to be processed, then the `WorkItem` creates two new `WorkItems`, each containing half of the work.

You also can set the number of dummy transactions and the number of threads for each account. The parameters are specified as static class variables in the `BankJob` object.

To vary the amount of time for reading and for processing a transaction, you can assign a burden value when the program is invoked. This burden value becomes the loop limit in a dummy math calculation that slows down the read or process work.

Two design choices were possible for the `Worker` object. It could be subclassed from the `Thread` object and run under a thread, or it could implement the `Runnable` interface. We chose to have the `Worker` implement the `Runnable` interface because it is more generic. Since Java does not support multiple inheritance but does support the implementation of multiple interfaces, we chose to implement the `Runnable` interface. This choice allows us to inherit from another class in the future, if appropriate. See the sample code in Figure 1.

Global Variables

C and C++ support global variables, but Java does not. However, with Java you can accomplish the same effect by using static class variables, which are either final (constant) or not. The Java Developer's Kit (JDK) uses this capability in several core classes, such as the `Color` class and its various colors like black, blue, and cyan; or the `Math` class and its E and PI final static variables. These class variables can be used without instantiating a new object, because the class itself is an instantiated object. The following shows an example of a static class variable:

```
public class BankJob implements
Observer {
  protected static final int
WORKNUMBER=50;
  protected static final int COUNT=5;
  protected static final int
NUMTHREADS=5;
  protected static InputManager IM;
  .
  .
  .
}
```

AWT and the New Event Model

Abstract Windowing Toolkit (AWT) is a set or package of classes used for the Graphical User Interface (GUI).

Figure 2 shows the user interface. The sample application handles the user interface

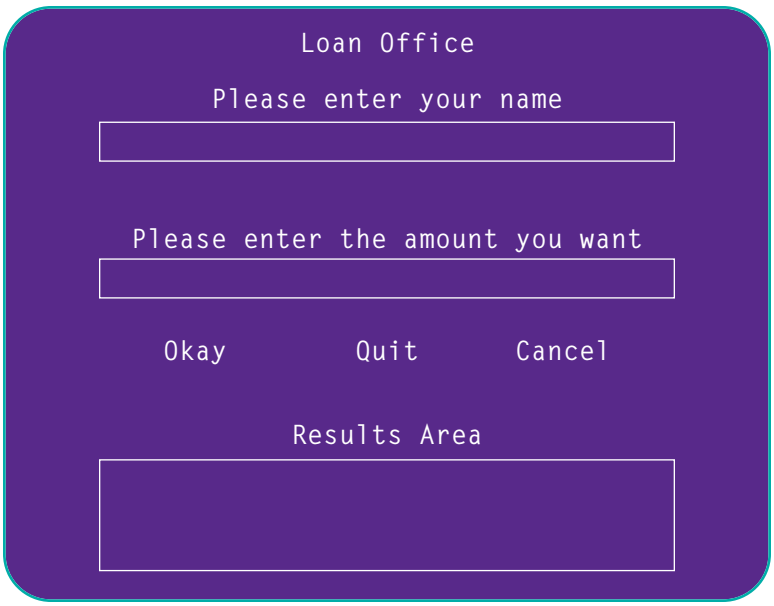

Figure 2. Application user interface

[1]Blumofe, Robert D. and Leiserson, Charles E. "Scheduling Multithreaded Computations by Work Stealing." *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*. Santa Fe, New Mexico. Nov. 20-22, 1994. Pages 356-368.

within the InputManager object. The InputManager uses two different kinds of events: user interface events such as pushing a button, and application events such as obtaining the required input data.

The event model for the JDK 1.1 is a subscription model: the object that is interested in the event registers to receive event notification. The object designer can freely design event handlers. For this sample code, the InputManager registers for the input events of interest, such as pushing the buttons and pressing Enter on the text fields. When the user pushes the "Okay" button or the Enter key, all of the input fields are processed. See Figure 3 for a simple user interface. A more complex interface design could require partitioning the event handling among different objects.

For the application-level events, the InputManager, a subclass of the Observable class, inherits the mechanisms from the Observable class to notify observers. The BankJob object inherits the Observer interface so that it can register for InputManager events, shown in Figure 4. Note that the

```
public class InputManager extends Observable implements Runnable,
                         ActionListener {
  .
  .
  .
  // Constructor
  InputManager () {
  .
  .
  .
    amountField.addActionListener(this);
  .
  .
  .
    loaneeField.addActionListener(this);
  .
  .
  .
    okayButton.addActionListener(this);
  .
  .
  .
    quitButton.addActionListener(this);
  .
  .
  .
    cancelButton.addActionListener(this);
  .
  .
  .
  }
  .
  .
  .
  public void actionPerformed (ActionEvent e) {
    Object source = e.getSource();
    if ( (source == amountField) || (source == loaneeField) ||
         (source ==okayButton) ) {
      processFields();
    }
    else if (source == quitButton) {
           iQuit();
         }
        else if (source == cancelButton) {
               iCancel();
             }
  }
  .
  .
  .
}
```

Figure 3. Action event handling example

```
public class InputManager extends Observable implements Runnable,
                          ActionListener {
  .
  .
  .
  public synchronized void iQuit() {
    setChanged();
    notifyObservers("Quit");
    System.exit(0);
  }
  .
  .
  .
  public synchronized void processFields() {
  .
  .
  .
    setChanged();
    notifyObservers("Ready");
  .
  .
  .
  }
  .
  .
  .
}

public class BankJob implements Observer {
  .
  .
  .
  // Constructor receiving artificial burdens for the worker threads
  public BankJob (int rb, int pb) {
    .
    .
    .
    IM = new InputManager();
    IM.addObserver(this);
    .
    .
    .
  }
  .
  .
  .
  public void update(Observable o, Object arg) {
    if (arg.equals("Ready")) {
      name = IM.getName();
      .
      .
      .
      amount = IM.getAmount();
      .
      .
      .
    }
  }
  .
  .
  .
}
```

Figure 4. Semantic event handling example

`setChanged` method of the `Observable` class must be invoked before the `notifyObservers` method can actually notify observers. Also, note that the `notifyObservers` method can pass an object along with the notification. In this case, we pass a `String` object with the status of the input—either `Quit` or `Ready`.

Conclusion

We hope this sample code will benefit your efforts in creating "real code." Be sure to check the Bank Loan Sample Program associated with this article.

Peg MacPhail, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. E-mail: macphail@austin.ibm.com. Ms. MacPhail is a technical consultant for RS/6000 solution providers. She has spent nearly 20 years with IBM in a variety of technical positions. She has a BA from the University of Connecticut at Storrs and a MCS from Texas A&M University in College Station.