Java Threads

By Chary Tamirisa

This article explores thread support provided in the Java language, the Java classes, and the Java Virtual Machine. It describes the Java execution environment with particular reference to the AIX 4 Java Development Kit (JDK) 1.0.2. We assume the reader is familiar with Java and has some knowledge of thread-related concepts.

One important feature of Java™ is its support for multithreaded programming. Part of this support is built into the Java language itself; the rest is provided through a few basic Java classes. The Java Virtual Machine also creates and manages threads to ease program execution. Programming languages such as C and C++ do not have thread support built into the language; in these languages, threads are typically supported through separate runtime libraries. For example, AIX 4 provides a POSIX.1c-based threads package called the *pthreads library* that is typically used by applications written in C and C++.

What is a Thread?

A *thread* is the basic unit of execution. You typically create a separate thread to execute an independent sequence of actions. There are two distinct parts of creating a thread that go hand in hand: the thread itself as an execution unit, and the action it executes (often referenced as the thread function or procedure, or thread body).

The distinction is often blurred between the execution unit and the thread body. For instance, when you create a thread using the pthreads library, you specify action as part of creating the thread. However, it is important to distinguish between these two.

A thread has certain attributes that govern its execution. For example, a thread has a priority that governs its scheduling. Other attributes, such as stack size, may be specified at thread creation time. Threads can be created and terminated as needed. In addition, threads can be temporarily suspended and later resume. When multiple threads concurrently operate on shared data, it becomes important to guard against simultaneous modifications of shared data by synchronizing the execution of multiple contending threads. Mutual exclusion locks among multiple threads typically guarantees this.

Understanding these basic concepts helps in writing well-behaved multithreaded programs.

Java Threads Overview

The following section provides a brief description of the Java Virtual Machine memory model that governs thread execution. It outlines the threads support in the Java language, then summarizes the built-in classes that assist in multithreading programs.

Memory Model

A *variable* is a memory storage location in a Java program. During their execution, threads share class variables, instance variables, as well as components of arrays. A thread may assign a value to a variable or use the value of a variable. When two or more threads act on the same variable concurrently, the variable may have timing-dependent values (this situation is also known as a race condition); therefore, the program can have indeterminate results. A well-behaved multithread program avoids a race condition.

The Java Virtual Machine provides a memory model that allows the fast and efficient implementation of concurrent programs. Briefly, this model provides a master copy of variables shared by all threads in *main memory*, with each thread keeping a local *working copy* of these variables in *working memory*. As threads operate on the

Chary Tamirisa

shared variables, data is transferred between the main memory and working memory.

The Java memory model specifies a set of rules when a thread is allowed or required to transfer data from the working memory to main memory, or vice versa. The main memory also associates each object with a lock. A thread typically obtains a lock before accessing a shared variable. The acquisition of a lock flushes a thread's working memory and guarantees that a thread's working memory is refreshed with the latest values from the main memory. Similarly, when a thread unlocks a lock, it guarantees that the variables in working memory will be written back to the main memory.

Language Features

Java supports thread synchronization by providing the keyword *synchronized*. This keyword can be used to create synchronized methods or to protect a block of code from a concurrent execution. A *block* is a sequence of statements with local variable declarations within braces. The syntax is as follows:

```
synchronized (Expression)Block
```

Java allows threads that access shared variables to keep private working copies of variables for the sake of efficiency.

The Expression must be a reference type (class, interface, or array). The executing thread acquires the lock associated with the Expression; then the Block is executed. If the Block executes normally, the lock is released and the synchronized statement completes normally. If the Block executes abnormally for any reason, the lock is unlocked and the synchronized statement completes abruptly.

Java allows threads that access shared variables to keep private working copies of variables for the sake of efficiency. These working copies are typically synchronized with the shared master copies only when objects are locked or unlocked. As a rule, threads should lock to enforce mutual exclusion before modifying shared variables.

Besides lock support, Java provides the *volatile* modifier for variables. This guarantees that a thread's copy of a variable is reconciled with its master copy every time it accesses the variable. The volatile modifier is useful in the following two cases:

- **Instance variables that are accessed in busy wait loops.** Here, by declaring them as volatile, we force the reloading of values that are typically changed by other threads during the iterations of the loop.
- **Instance variables used in native methods.** In this case, the best you can do is to mark the instance variable as volatile.

Built-in Classes

In addition to these two keywords—synchronized and volatile—in the language, a few basic Java classes support Java's thread creation and management. All classes that support multithreading are part of the `java.lang` package. By default, all Java programs (applications or applets) import this package. The thread-supporting classes are as follows:

- `java.lang.Thread`.This basic class relates to the actual creation and control of a Java thread. It provides basic thread management methods, such as start and stop a thread, suspend and resume, and get and set priorities of a thread.
- `java.lang.ThreadGroup`.This basic class is similar to the one above. It relates to the actual creation and control of a Java thread and provides basic thread management methods. It differs, however, by relating to the whole group of threads, not just to a single thread.
- `java.lang.Runnable`. The interface defines the `run()` method. Any class that implements this interface can provide a thread procedure.
- `java.lang.Object`. The thread-related methods in `java.lang.Object` are synchronization methods such as `wait()`, `notify()`, and `notifyall()`.

Using these Java language features and its basic classes, you can write efficient multithreaded programs. Moreover, the Java Virtual Machine creates threads to support the execution of a program, and these threads are additional to any threads created by a program. Since the actual threads created by the runtime may depend on a particular environment (such as AIX 4 or WIN32), you cannot depend on their specifics when writing Java programs. However, knowing what they do can help in understanding the execution and control flow of a program.

Java Threads Programming Model

Based on functionality, the methods provided in the Java `Thread` class can be categorized as

follows: thread creation, thread groups, thread properties, thread control, thread synchronization, thread scheduling, thread blocking, and asynchronous interruption of threads.

In the following sections, we discuss these and also the Java security model with respect to threads, followed by multithreading aspects of Java applets. Finally we examine thread safety in the Java built-in classes.

Thread Creation

A thread executes a specified action as a separate execution unit. Java provides the basic class `java.lang.Thread`, which you can extend (subclass) to create a thread of execution. You override the `Thread` class' `run()` method to specify the action to be executed by the thread.

Java allows only single class inheritance; it does not allow a class to subclass from multiple classes. So, if your class extends a thread, it cannot subclass from any other classes that you may need; this can be restrictive. For this reason, subclassing from the `Thread` class is not generally the best way to create classes that must be run in separate threads.

Implementing the `java.lang.Runnable` interface is another way to create classes that can be run as threads. A class can subclass from any user-defined class and still implement the `Runnable` interface to run as a separate thread. Implementing the `Runnable` interface in a class does not prevent the class from implementing other interfaces, because Java allows a class to implement multiple interfaces. This overcomes the restriction if you just subclass from the `Thread` class.

The following sections illustrate both ways of creating threads.

Extending the java.Thread class

In this method, your class simply extends the `java.lang.Thread` class and overrides its `run()` method to provide the thread action. Note that you need to invoke the `start()` method of the `Thread` class to run the thread.

In Figure 1, the `Sorter` class subclasses the `Thread` class and invokes the `start()` method in its constructor. The `run()` method specifies the desired sorting action.

An application program may use the sorter as shown in Figure 2.

In Figure 1, the sorter thread is started in the constructor. If you want to better control the starting of the sorter thread, you can do so from an application program that creates the sorter object. Figure 3 shows an example.

An application program can create the sorter object and start the thread as shown in Figure 4.

```
class sorter extends Thread
{

      sorter()
      {
            start(); // run the thread.
      }
      public void run()
      {
      }
}
```

Figure 1. A thread class invoking the start() method

```
class Sort
{
      // Create the sorter object. The  sorter class is a
      // thread that starts running right away.
      new sorter();

}
```

Figure 2. Creating a thread

```
class sorter extends Thread
{
      sorter()
      {
      }
      public void run()
      {
      }
}
```

Figure 3. A thread class without invoking the start() method

Implementing the Runnable Interface

Three steps are involved in implementing the `Runnable` interface. First, declare the class as implementing the `Runnable` interface. As part of this process, provide the implementation for the `run()` method—the real action to be executed in a separate thread. Then create an instance of the

```
class Sort
{

     // Create the sorter thread object.
     sorter s = new sorter();
     // start the thread.
     s.start();
}
```

Figure 4. Creating and starting the sorter thread

```
Class myThread extends myClass implements Runnable{

          // the action for the thread
          public void run()
          {
          }

}
```

Figure 5. Implementing the Runnable interface

Thread groups in Java are organized as a tree, with the system thread group at the root.

`Thread` class specifying the class to run as a thread, as an argument to the `Thread()` constructor. Finally, you can run the thread by invoking the `start()` method on the `Thread` object. Details of these steps follow.

Step 1: Specify the action to be executed in the thread. Figure 5 shows an example of a class implementing a `Runnable` interface. A `Runnable` object does the following:

- Declares that it implements `Runnable` interface
- Provides an implementation for the `run()` method; the `run()` method must be specified as a `public void`

Step 2: Create a thread object. A thread is typically created by invoking the constructor on the `Thread` class as follows:

```
Thread thread = new Thread(target);
```

where target is a `Runnable` object, an instance of `myThread` class.

Step 3: Run the thread. After a thread is created, it can be run by invoking the `start()` method on it as follows: `thread.start()`.

Each program decides how to start a thread. For more control on starting the thread, follow the steps outlined above; however, it is possible to invoke the `start()` method in the class constructor after creating the Thread.

The `Thread()` constructor has seven varieties (overloaded forms); the generic form is as follows:

```
public Thread(ThreadGroup group, Runnable
target, String name).
```

Typically, programs do not specify any thread group (or name) to be associated with the thread. If you do not specify a thread group (see below), the program assumes the current thread group. For a typical Java program (application or applet), the `ThreadGroup` is the main group. Remember the default `run()` method of the `Thread` class does not do anything; it simply returns (it is expected to be overridden). Therefore, it is important to override the `run()` method and provide some useful implementation for it. The `name` can be specified to identify threads created by a program, which can be useful when debugging programs.

Thread Groups

Sometimes it is desirable to create several threads and be able to control them collectively. If you need to `stop()`, `suspend()`, or `resume()` a collection of threads as a whole, the notion of thread groups becomes useful. With thread groups in Java, you can also impose security by controlling the ability of a thread to create another thread outside its own group.

Thread groups in Java are organized as a tree, with the `system` thread group at the root. A typical Java application (one with `main()`) will contain at least the `system` and `main` thread groups. A typical Java applet will contain at least these two groups and the `applet` group. A thread group can also contain other thread groups. That is, the `system` thread group contains the `main` group in the case of applications, and it contains the `main` group and the `applet` group in the case of applets.

Thread Properties

When you create a thread in Java, the newly created thread inherits the following properties from the parent thread: daemon state and priority.

After creating a thread but before starting it, you can change both properties. The new thread will run with the new values for these properties. However, once the thread is started, you cannot change the daemon state. Priority works differently; it can be set before starting a thread or while the thread is running.

Java runtime schedules the background (daemon) threads periodically. In the AIX 4 implementation of Java VM, the *Finalizer* thread and the *Async Garbage Collection* thread are examples of background threads; each thread has `Thread.MIN_PRIORITY`. Another important point to remember about daemon threads is that if only these background threads remain in an application, the Java runtime terminates the application.

Thread Control

Thread execution in Java can be controlled in several ways. First, note that Java distinguishes thread creation from running of the thread. You must explicitly invoke `Thread.start()` in order to run a thread; otherwise, the thread object is created, but the thread is not run.

You can suspend a thread during its execution by invoking the `suspend()` method on it. Invoking `suspend()` multiple times on a suspended thread does not affect the suspended state of the thread. A suspended thread can be restarted by invoking `resume()` on it. Invoking `resume()` on a non-suspended thread does not affect the execution of the thread. Similarly, even if `resume()` is invoked on a thread multiple times, only one call to `suspend()` is required to suspend it.

Lifetime of a Thread

A thread starts running when the `start()` method is invoked on it, but there are three ways to terminate it:

- It returns normally from the `run()` method (which constitutes the thread body)
- An uncaught exception is raised in it
- The `stop()` method is called on it

You can invoke `stop()` on a thread only one time. After it is stopped, you should not invoke `start()` on it. If you do, an `IllegalThreadState` exception will occur. When a thread is stopped, you can force the Java Virtual Machine to release the thread object (and its resources) by explicitly setting it to null; the thread object will be garbage collected later.

Thread Synchronization

Thread synchronization deals with mechanisms that allow multiple threads to operate on shared data concurrently without race conditions. In addition, Java allows a thread to wait for event notifications by another thread. The following sections discuss these synchronization mechanisms in detail.

Locks (Mutexes)

Every object in Java has a monitor or lock associated with it; that is, every class and every instance of a class has its own lock. This lock can synchronize concurrent access to the object by multiple threads. These locks are commonly known as *mutexes* or mutual exclusion locks. The synchronized modifier designates places where a thread must acquire the lock before proceeding. All static, synchronized class methods use the same class object lock. Similarly, all instance methods in a class use the same instance object lock. The following examples show static and non-static methods of locking.

Static method. In Figure 6, the static method `sync()` is locked using the class object's lock.

Every object in Java has a monitor or lock associated with it.

```
class StaticSynchExample {

     synchronized static void sync() {
     }
}
```

Figure 6. Locking a static method (class object locking)

Non-static method. The lock of the instance of the class, not the class object's lock, is used in Figure 7.

```
class SynchExample {

     synchronized void sync() {
     }

}
```

Figure 7. Locking a non-static method (instance object locking)

The synchronized modifier can synchronize or lock arbitrary blocks of code. Specify the object to be locked as follows:

```
synchronized(sharedobject) {
     // code to synchronize

}
```

The lock on shared object is acquired (locked) when the code in the block (within {}) is executed. A synchronized method is equivalent to the following:

```
void sync()
{

     synchronized(this) {
     }

}
```

Suppose a non-static method needs to synchronize with a static method. How does it acquire the lock used by the static method, that is, the class object lock? The class object lock can be locked as follows: `synchronized(getClass())` where `getClass()` is a method of the `java.lang.Object` class. Note that `getClass()` returns the runtime class of an object.

From the code in a synchronized block, you can invoke either the same method or another method in the same class without blocking. The same is true for calls on other objects for which the current thread holds a lock. Figure 8 shows an example.

```
class recursive {
     synchronized void method1()
     {
          method2(); //
     }
     synchronized void method2()
     {
     }
}
```

Figure 8. Recursive locking

In Figure 8, `method1()` holds the lock on the instance of the recursive object; it does not block when calling `method2()` on the same object.

Overriding A Synchronized Modifier

You can override the synchronized modifier in subclasses. Suppose you are subclassing from a class with synchronized methods and you want to override some of these synchronized methods. You must mark these methods "synchronized" in your subclass, otherwise these methods will not be synchronized in the subclass. The overridden method must be explicitly declared synchronized or the methods are treated as unsynchronized in the subclass.

Event Notification

Imagine a producer and consumer relationship where the consumer threads must wait for data from the producer thread. When the producer thread has data, it must inform the waiting consumer thread. These waits are best implemented with the consumer thread blocked and not polling some flag to check if the data is available.

To support this type of thread cooperation, Java provides a feature that is similar to the POSIX™ condition variable mechanism. Java allows a thread to release a lock and block atomically. The atomic unlock and block feature allows a thread to check if a condition to wait exists. If so, the thread waits for an event to occur to change the condition. This process involves locking a mutex, checking the condition, and if the condition indicates the thread should block, releasing the lock and then blocking.

Java allows this mechanism through the methods provided in the `java.lang.Object` class. Another thread notifies the blocked thread when the condition changes. This notification is done with the same mutex lock held by the other thread. When it is notified, the blocked thread is awakened and it returns from the `wait()` with the mutex locked again. When `wait()` returns, check the condition again to see if it is valid to proceed or if the wake up is spurious; hence, the thread should wait again. Figure 9 shows the idiom to follow when using the wait/notify mechanism.

```
Waiter:

    synchronized(obj) {
         while (  somecondition != true)
              obj.wait();
    }

Signaller:

     synchronized(obj) {
          somecondition = true;
          obj.notify();
     }
```

Figure 9. Waiting for an event

The waiter and signaler lock the same object: `obj`. The `wait()`, `notify()`, and `notifyAll()` methods must be invoked with the synchronization lock held on their targets. Failure to lock results in `IllegalMonitorStateException` at runtime.

When the wait is done on `obj`, the calling thread gets suspended. Similarly, when signaling, the `obj` is notified, the waiter thread is awakened as a result.

In addition, since `wait()` releases the lock of `obj` when it is blocked, it reacquires the lock when it is awakened. If it cannot get the lock immediately, it waits until the lock is acquired. Typically, the signaler thread releases the lock so that the waiter thread can reacquire the lock.

An important point to remember is that if the thread invoking `wait()` has acquired the same lock *n* times, the call to `wait()` will result in *n* unlock operations. This releases the lock so that another thread can acquire it. When the `wait()` returns, it reacquires the lock *n* times. Therefore, the invariant across a call to `wait()` is maintained; the lock on `obj` is in the same state before and after the call to `wait()`.

Although `wait()` releases the lock on the target object when blocking, other locks held by the thread are not released. This is important to note; otherwise threads may be blocked, holding locks that could potentially prevent progress of other threads that need these locks.

The following example shows two threads: a producer thread and a consumer thread that share a buffer (Vector). In this example, the `main()` creates a producer and a consumer object that share a common buffer object. The shared buffer supports `put()` and `get()` methods. The producer adds elements to the buffer and the consumer removes them. If the shared buffer is too full, the producer waits for the buffer to be emptied. If the shared buffer is empty, the consumer will wait. The producer notifies the consumer when an element is added to the shared buffer; the consumer notifies the producer when an item is consumed. The producer and consumer are synchronized.

Figure 10 shows an example of a producer and consumer sharing a buffer.

Deadlocks

Similar to the POSIX threads model, Java does not detect deadlocks; you must detect or prevent them.

Thread Scheduling

Java specifies a priority-based preemptive thread scheduling discipline; that is, an attempt is made to run the thread with the highest priority at any time. There is no guarantee that the highest priority thread is run always. If a thread with a lower priority is running when a higher priority thread becomes ready to run, the lower priority thread is preempted.

If two threads have equal priority, there is no guarantee that both threads receive equal CPU time; that is, Java does not mandate time-slicing. Some Java implementations may provide time-slicing; however, applications should not assume this. If it is necessary to share time across two threads of equal priority, use the `Thread.yield()` method to periodically yield CPU to another thread.

When a thread is created, it inherits the priority of the thread that created it. However, it is possible to change that priority by calling `Thread.setPriority()` with a priority argument between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`.

Java specifies a priority-based preemptive thread scheduling discipline.

Thread Blocking and Interrupting

A thread may block in Java when the following methods are executing:

- `java.Object.wait()`
- `java.Thread.sleep()`
- `java.Thread.join()`

A call to `java.Thread.interrupt()` may interrupt a thread from these waits. This interrupt causes the blocked thread to return with the exception `InterruptedException`. Note that a thread may not react immediately to an interrupt.

A call to `stop(Throwable thr)` can also interrupt a thread. The target thread is forced to complete abnormally and to throw the `Throwable` object `thr` as an exception. The exception is delivered asynchronously because it can occur any time during a thread's execution.

In addition, Java Virtual Machine can throw `InternalError` exceptions at any time because of errors that may occur in the Java Virtual Machine. These can arise because of a fault in the virtual machine software, in the underlying host software, or in the hardware. Such exceptions can occur at any point in a Java program.

The Java Development Kit (JDK) 1.1 provides the `Thread` class method `interrupt()` as a way to interrupt a thread asynchronously. This interrupt may not take immediate effect. If the thread

```
import java.util.*;
public class demo {

    public static void main(String[] argv) {
         sharedBuffer s = new sharedBuffer();
         new producer(s);
         new consumer(s);

    }

}
class producer implements Runnable {

    sharedBuffer s;
    producer(sharedBuffer s) {
         this.s = s;
         Thread thd = new Thread(this);
         thd.start();
    }
    public void run() {
        for(int i=0; I <100;i++) {
             s.put(new Integer(i));
             System.out.println("put: int=" + I);
        }
    }

}
class consumer implements Runnable {

    sharedBuffer s;
    consumer( sharedBuffer s)
    {
        this.s = s;
        Thread thd = new Thread(this);
        thd.start();
    }
    public void run() {
        Integer ret;
        for(;;) {
             ret= s.get();
             System.out.println("get: int=" + ret);
        }
    }
}
class sharedBuffer {

    Vector v;
    static final int MAX=10;
    sharedBuffer() {
        v = new Vector();
    }
    synchronized void put(Integer I ) {
        while(v.size() == MAX){
             System.out.println("MAX size is reached");
             try{
                 wait();
             }
             catch(InterruptedException e){}
             System.out.println("put(): try again—out of Wait()");
        }
```

(continued on next page)

Figure 10. Producer/consumer sharing a buffer

(continued from previous page)

```
        v.addElement(I);
        notifyAll();
        return ;
    }
    synchronized Integer get() {
         while(v.isEmpty() ) {
              try{
                  System.out.println("get(): buffer is EMPTY");
                  wait();
              }
              catch(InterruptedException e){}
              System.out.println("get(): out of Wait()");
         }
         Integer ret = (Integer)v.firstElement();
         v.removeElement(ret);
         if(v.size() < MAX) notify();
         return ret;
    }
}
```

Figure 10. Producer/consumer sharing a buffer

is waiting, it is awakened and an exception is generated. An interrupted exception is thrown.

Note: When a thread is blocked in `wait()`, `sleep()`, or `join()`, and an interrupt is sent to it (by `stop()` or `interrupt()`), the desired interrupt is not delivered to the thread in JDK 1.0.2 on AIX 4. It is not clear if this is just a bug. The interrupt is delivered if the thread is not blocked, but is executing some computation.

Security

Java allows programs to prevent certain actions on a thread (in addition to other system resources) for security reasons. Your program can check security before performing actions to stop, suspend, resume, change priority, set name, and set daemon on a thread. It can disallow such actions, if so desired. The way to provide this security is to subclass the `java.lang.SecurityManager` and override the `checkAccess()` method to do the desired checking for thread operations.

The methods of the `SecurityManager` provide the strictest security by throwing exceptions for all of its methods. You must override its methods to allow useful work to be done. Java runtime invokes the `checkAccess()` method before performing thread-related actions, and it is up to the `checkAccess()` to disallow the action as needed to enforce security.

The example in Figure 11 shows how security can be specified. We provide the `threadSecurityMgr` class, which inherits from the `SecurityManager` class and overrides `checkAccess()` to throw exceptions whenever a thread is stopped, suspended, resumed, or its priority, name or daemon state is set. Figure 12 shows the results when run on AIX 4, JDK 1.0.2. A set of checks can be imposed for most thread operations.

There are certain rules for installing a security manager for a program. You can install a security manager only once, and after it is installed, you cannot change it. This makes it important to override all the methods of the `SecurityManager` class as needed and install the new manager early in the program. Java-enabled browsers such as Netscape Navigator install their security manager class before they run Java applets.

Applets and Threads

A Java applet can run in a Java-enabled Web browser, such as Netscape Navigator 3.0. The applet class in Java provides methods such as `start()`, `stop()`, `init()`, and `destroy()`. Although the methods of the `Applet` class are similar to those of the `Thread` class, they are not identical. For example, the `start()` and `stop()` methods of an applet can be called several times by the Web browser as the user moves around scanning the Web pages. Typically, a Web browser starts the applet when the applet is displayed, and stops when the user moves to another page or scrolls the applet out of view. When an applet is stopped, the underlying thread is suspended; it is resumed when the

Specifying a Security Manager

```
import java.awt.*;

class threadSecurityMgr extends SecurityManager {
     public void checkAccess(Thread g) {
          throw new SecurityException("access to thread" + g);
     }
}
public class secure extends Thread {
     static public void main(String[] args) {
          System.setSecurityManager(new threadSecurityMgr());
          secure s = new secure();
          s.start();
     }
     public void run() {
          System.out.println("Secure Thread: now set priority");
          setPriority(Thread.MAX_PRIORITY);
          System.out.println("Secure Thread: now stop it ");
          stop();
     }
}
```

Results of running the Security Manager example

```
Secure Thread: now set priority
java.lang.SecurityException: access to threadThread[Thread-2,5,main]
at threadSecurityMgr.checkAccess(secure.java:8)
at java.lang.Thread.checkAccess(Thread.java)
at java.lang.Thread.setPriority(Thread.java)
at secure.run(secure.java:25)
```

Figure 11. Specifying a Security Manager and the results

applet is started again. When the browser invokes the applet's `destroy()` method, the applet's thread is stopped and garbage collected.

In the applet example in Figure 12, an applet implements `Runnable` interface. In addition to any threads the Java runtime creates, we create a thread to execute the `run()` method, which invokes `repaint()`. This in turn invokes the `paint()` method, which then draws the current date and time on the panel. We create this thread with the applet class as the thread target class and start it when the applet's `start()` method is called by the browser. Note the number of threads that exist in the applet and their function.

The AIX 4 JDK 1.0.2 creates several thread groups: `system`, `main`, and `applet-myapplet.class`. The thread in the `system` thread group is the `Finalizer` thread. Threads in the `main` thread group consist of the main thread, the `AWT-Motif` thread, and the `ScreenUpdater` thread. The threads in the `applet-myapplet` thread group are Thread-1 and Thread-2. All applets have the Thread-1 thread. Since `myapplet` class creates another thread, Thread-2 is created. Thread-1 invokes the `init()` and the `start()` methods in the `applet` thread group. The `AWT-Motif` thread invokes the `paint()` method. Thread-2 invokes the `run()` method.

Java Core Classes and Thread Safety

A Java program typically subclasses (inherits from) the base classes, such as those provided in the following packages: `java.lang`, `java.awt`, `java.io`, `java.util`, `java.net`, and `java.applet`. For a multithreaded program to be safe with respect to threads, it is important to know if the classes (methods) are thread-safe, or if the application developer must do external locking before invoking any of the methods.

Java classes have evolved from JDK 1.0 through JDK 1.1. Some classes have improved

thread-safety. All Java base classes are thread-safe and do not require external locking; that is, you do not need to synchronize calls to the base Java classes because they are already supposed to be thread-safe (unless the documentation says otherwise).

Another point regarding thread-safety is that you cannot override static or final methods in a class. For example, you cannot override the synchronization methods `wait()` and `notify()` in `java.lang.Object` class because these are declared final. If you are working with object references on the client side using Java's Remote Methods Invocation (RMI) Application Programming Interfaces (APIs), synchronizing on the local object references does not translate to synchronizing on the remote (real) objects.

AIX 4 Java Threads

To help you better understand the execution environment, we will describe some Java runtime threads found in AIX 4 JDK 1.0.2. Our intent is to help you understand the actual control flow or how applications/applets work in Java. You must not directly interact with these threads or manipulate them in another way. AIX 4 support for Java threads is based on the pthreads provided in the AIX 4 pthreads library. Pthreads maps application threads to the native kernel threads. Currently, there is a one-to-one mapping

```
import java.applet.*;
import java.awt.*;
import java.util.*;
public class myapplet extends Applet implements Runnable {

     private Thread updateThread;
     public void init() { }
     public void paint(Graphics g) {
          g.drawString(new Date().toString(),10,20);
     }
     public void run() {
          for(;;){
               try {
                   Thread.sleep(1000); // sleep for 1000 milliseconds
               }
               catch(InterruptedException e){

                   System.out.println("Sleep interrupted");
               }
               repaint();
          }
     }
     public void start() {
          if(updateThread == null){
               updateThread = new Thread(this);
               updateThread.start();
          }else{
               updateThread.resume();
          }
     }
     public void stop() {
          Thread mainthd = Thread.currentThread();
          updateThread.suspend();
     }
     public void destroy() {
          updateThread.stop();
          updateThread=null;
     }
}
```

Figure 12. Threads in applets

```
public class helloWorld
{

     static public void main(String[] args)
     {
          new helloWorld ();
     }
     public helloWorld()
     {
          Thread mainthd = Thread.currentThread();
          System.out.println("Current Thread=" + mainthd);
          System.out.println(" Listall Thread Groups and Threads:");
          PrintThreadsInfo.getAllThreads();
     }

}
public class PrintThreadsInfo {

     public static void getAllThreads() {
               ThreadGroup rootgrp;
               ThreadGroup parentgrp;
               ThreadGroup thdgrp=Thread.currentThread().getThreadGroup();
               rootgrp = thdgrp;
               parentgrp = thdgrp.getParent();
               while(parentgrp != null) {
               rootgrp = parentgrp;
               parentgrp = parentgrp.getParent(); }
               listgroup(rootgrp);
     }
     static void listgroup(ThreadGroup group)
     {
          if(group == null) return;
          int num_threads = group.activeCount();
          int num_groups = group.activeGroupCount();
          Thread[] threads = new Thread[num_threads];
          ThreadGroup[] groups = new ThreadGroup[num_groups];
          group.enumerate(threads,false);
          group.enumerate(groups,false);
          System.out.println( " Group Name = " + group.getName());
          System.out.println( " Group Max Priority="+ group.getMaxPriority());
          System.out.println((group.isDaemon()? " (Daemon Group?)Yes":";      (Daemon Group?)No") );
          for(int I=0;i<num_threads;i++) print_info(threads[i]);
          System.out.println(" ");
          for(int I=0;i<num_groups;i++) listgroup(groups[i]);
     }
     static void print_info(Thread t)
     {
          if(t==null) return;
          System.out.println("\tThread=" + t.getName() );
          System.out.println("\t\tPriority=" + t.getPriority());
          System.out.println("\t\t" + (t.isDaemon() ?       "Daemon(Yes)":"Daemon(NO)"));
          System.out.println("\t\t" + (t.isAlive() ? "Live":"not Live"));
     }

}
```

Figure 13. Extracting information on all the thread groups and threads in an application

of the pthreads to the kernel threads. The Java Virtual Machine creates threads that are hosted using the pthreads package.

Every thread in Java belongs to a thread group, which may contain other thread groups or the actual threads themselves. There is a notion of a parent of a thread group, but there is one thread group called the `system` group for which there is no parent. The `system` group is the root of all the thread groups. A thread group is governed by certain attributes, such as the maximum priority a thread belonging to this group can have, the name of this thread group, and whether this group has a daemon thread group.

In AIX 4, the Java runtime creates the following thread groups for a non-GUI application (a program invoked through a Java command and contains main): `system` and `main`. The `system` thread group is a non-daemon thread group with the maximum priority of any thread within it set to `Thread.MAX_PRIORITY`. The `system` group consists of two threads, the `Async Garbage Collector` and the `Finalizer` threads, both daemon threads with the priority set to the value `Thread.MIN_PRIORITY`.

The `main` thread group is a single thread when the main of the application is invoked. This main thread, a non-daemon thread, runs at `Thread.NORM_PRIORITY`. All application threads belong to one of the following:

- The `main` thread group
- Thread groups created by the application
- Thread groups created by the Java runtime

If the application creates a GUI, such as a frame, the AIX 4 Java runtime creates two additional threads: the `AWT-Input` and the `AWT-Motif` thread. These two threads also belong to the `main` thread group and run as non-daemon threads with priority set to `Thread.NORM_PRIORITY`. The `AWT-Motif` thread handles all GUI-based events. Thus, if your application has a `paint()` or `handleEvent()` method, these are invoked in the thread `AWT-Motif` thread.

In addition to the `system` and `main` thread groups, a new thread group is created within an applet, for that applet class. This applet thread group is a non-daemon thread group. A non-daemon thread with priority of six is created in this applet thread group. This applet thread executes the `init()` method of the applet.

```
Current Thread=Thread[main,5,main]
Listall Thread Groups and Threads:
Group Name = system
Group Max Priority=10
(Daemon Group?)No

     Thread=Async Garbage Collector

          Priority=1
          Daemon(Yes)
          Live
     Thread=Finalizer thread
          Priority=1
          Daemon(Yes)
          Live

Group Name = main
Group Max Priority=10
(Daemon Group?)No

     Thread=main
     Priority=5
     Daemon(NO)
     Live
```

Figure 14. Threads in an application

To illustrate the Java runtime environment, consider first a Java application, then a Java applet. The complete code for both is shown for illustration purposes. The results of running the two examples were summarized above.

Java Application Example

In this example, we create a simple `helloWorld` class to display all the thread groups and threads running in the process. The `helloWorld` class does not create any threads; all the threads and thread groups you see are created by the Java runtime. The static method `getAllThreads()` of `PrintThreadsInfo` class also illustrates how to navigate the thread group and get to the root thread group.

Figure 14 shows the results of running the program shown in Figure 13 on AIX 4 JDK 1.0.2.

Java Applet Example

Details of the actual threads created in an applet may vary based on how the applet is launched. For instance, you can run the applet using the `appletviewer` command. The applet also can be launched through a Java-enabled Web browser such as Netscape Navigator 3.0.

We have summarized the results of running the applet using the `appletviewer` command. Results from running in a Web browser do not vary much from using this command.

In an applet, Java runtime creates a new thread group named after the particular applet class, in addition to the `system` and `main` thread groups. The `system` thread group consists of the `Async Garbage Collector` and `Finalizer` threads; the main thread group consists of the `main`, `AWT-Input`, `AWT-Motif`, and `Screen Updater` threads. Two threads in the `system` group are daemon threads; the rest are not. For example, in the following `helloWorldApplet` class, Java runtime creates the `applet-helloWorldApplet` thread group plus the system and main thread groups. In addition, an applet thread is created in this applet group. The `paint()` method of the applet is invoked in the `AWT-Motif` thread of the main thread group. Figure 15 shows an example of extracting information on all the thread groups and threads in an applet.

To run this applet, create the HTML file found in Figure 16.

Type the following command to run the HTML file: `appletviewer run.html`. Figure 17 shows the results of running the applet on AIX 4 JDK 1.0.2.

Summary

Java provides a simple, but powerful, programming model for multithreading applications. This overview covered Java's memory model, its threads-related language features, and the built-in Java classes. It also detailed the Java threads programming model.

We explored some implementation details of the AIX 4 JDK 1.0.2 to understand the actual runtime threads created. This is helpful in debugging programs as well as in understanding the control flow in Java programs.

```
import java.applet.*;
import java.awt.*;
public class helloWorldApplet extends Applet {

     public void init() {

          Thread mainthd = Thread.currentThread();
          System.out.println("Current Thread=" + mainthd);
          PrintThreadsInfo.getAllThreads();
     }
     public void paint(Graphics g) {

          g.drawString("Hello World Applet", 10,100);
          System.out.println("paint()—Current Thread=" +
               Thread .currentThread());
     }

}
```

Figure 15. Extracting information on all thread groups and threads in an applet

```
run.html:

  <APPLET CODE=helloWorldApplet.class WIDTH=400 HEIGHT=400>
  </APPLET>
```

Figure 16. Simple HTML file to run the sample in Figure 15

```
     Current Thread=Thread[thread applet-helloWorldApplet.class,6,group app let-helloWorldApplet.class]

          Group Name = system
          Group Max Priority=10
          (Daemon Group?)No
               Thread=Async Garbage Collector
               Priority=1
               Daemon(Yes)
               Live
          Thread=Finalizer thread
               Priority=1
               Daemon(Yes)
               Live

     Group Name = main
     Group Max Priority=10
     (Daemon Group?)No

          Thread=main
               Priority=5
               Daemon(NO)
               Live
          Thread=AWT-Input
               Priority=5
               Daemon(NO)
               Live
          Thread=AWT-Motif
               Priority=5
               Daemon(NO)
               Live
          Thread=Screen Updater
               Priority=4
               Daemon(NO)
               Live

     Group Name = group applet-helloWorldApplet.class
     Group Max Priority=6
     (Daemon Group?)No

          Thread=thread applet-helloWorldApplet.class
               Priority=6
               Daemon(NO)

          Live

paint()—Current Thread=Thread[AWT-Motif,5,main]
```

Figure 17. Threads in an applet—results of running the sample in Figure 13

References

Gosling, James; Joy, Bill; Steele, Guy. *The Java Language Specification*. Addison-Wesley, 1996.

Flanagan, David. *Java in a Nutshell*. O'Reilly & Associates, Inc., 1996.

Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1996.

Chary Tamirisa, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com. Mr. Tamirisa was the team leader for the threads package on AIX and OS/2® DCE. He has also worked in the fields of communication protocols, system software, and National Language Support. Mr. Tamirisa has an MS in Computer Science from McGill University and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.*