CONTENTS

Chapter 16. Concurrent Programming with Java Threads

Topics in This Chapter

- Starting threads by using separate thread objects
- Starting threads within an existing object
- Solving common thread problems
- Synchronizing access to shared resources
- Methods in the Thread class
- Exploring alternative approaches to multithreaded graphics
- Implementing double buffering
- Animating images
- Controlling timers

The Java programming language has one of the most powerful and usable concurrent programming packages of any modern programming language: the `Thread` class. Threads are sometimes known as "lightweight processes": processes that share the heap but have their own stack. Threads provide three distinct advantages.

Efficiency Performing some tasks is quicker in a multithreaded manner. For instance, consider the task of downloading and analyzing five text files from various URLs. Suppose that it took 12 seconds to establish the network connection and 1 second to download the file once the connection was open. Doing this serially would take 5 x 13 = 65 seconds. But if done in separate threads, the waiting could be done in parallel, and the total time might take about 12 + 5 = 17 seconds. In fact, the efficiency benefits of this type of approach are so great that the Java platform *automatically* downloads images in separate threads. For more information, see Section 9.12 (Drawing Images).

Convenience Some tasks are simpler to visualize and implement when various pieces are considered separately. For instance, consider a clock display or an image animation; in each case, a separate thread is responsible for controlling the display updates.

New Capabilities Finally, some tasks simply cannot be done without multiprocessing. For instance, an HTTP server needs to listen on a given network port for a connection. When the server obtains one, the connection is passed to a different process that actually handles the request. If the server didn't do this, the original port would be tied up until the request was finished and only very light loads could be supported; the server would be unusable for

real-life applications. In fact, in Chapter 17 (Network Programming) we'll present code for a simple multithreaded HTTP server that is based on the socket techniques discussed in Chapter 17 and the threading techniques explained here.

However, a word of caution is in order. Threads in the Java programming language are more convenient than threads or "heavyweight" processes in other languages (e.g., `fork` in C). Threads add significant capability in some situations. Nevertheless, testing and debugging a program where multiple things are going on at once is much harder than debugging a program where only one thing is happening at a time. So, carefully weigh the pros and cons of using threads before considering them, and be prepared for extra development and debugging time when you do, especially until you gain experience.

16.1 Starting Threads

The Java implementation provides two convenient mechanisms for thread programming: (1) making a separate subclass of the `Thread` class that contains the code that will be run, and (2) using a `Thread` instance to call back to code in an ordinary object.

Mechanism 1: Put Behavior in a Separate Thread Object

The first way to run threads in the Java platform is to make a separate subclass of `Thread`, put the actions to be performed in the `run` method of the subclass, create an instance of the subclass, and call that instance's `start` method. This technique is illustrated in Listings 16.1 and 16.2.

Listing 16.1 `DriverClass.java`

```
public class DriverClass extends SomeClass {
  ...
  public void startAThread() {
    // Create a Thread object.
    ThreadClass thread = new ThreadClass();
    // Start it in a separate process.
    thread.start();
    ...
  }
}
```

Listing 16.2 `ThreadClass.java`

```
public class ThreadClass extends Thread {
  public void run() {
    // Thread behavior here.
  }
}
```

If a class extends `Thread`, then the class will inherit a `start` method that calls the `run` method in a separate thread of execution. The author of the class is responsible for implementing `run`, and the thread dies when `run` ends. So, even though you put your code in `run`, you call `start`, not `run`. If you call `run` directly, the code will be executed in the current thread, just like a normal method. Data that is to be local to that thread is normally kept in local variables of `run` or in private instance variables of that object. Outside data is only accessible to the thread if the data (or a reference to the data) is passed to the thread's constructor or if publicly available static variables or methods are used.

Core Warning

*Never call a thread's `run` method directly. Doing so does **not** start a separate*

thread of execution.

For example, Listing 16.3 gives a `Counter` class that counts from 0 to *N* with random pauses in between. The driver class (Listing 16.4) can create and start multiple instances of `Counter`, resulting in interleaved execution (Listing 16.5).

Listing 16.3 `Counter.java`

```
/** A subclass of Thread that counts up to a specified
 *  limit with random pauses in between each count.
 */

public class Counter extends Thread {
  private static int totalNum = 0;
  private int currentNum, loopLimit;

  public Counter(int loopLimit) {
    this.loopLimit = loopLimit;
    currentNum = totalNum++;
  }

  private void pause(double seconds) {
    try { Thread.sleep(Math.round(1000.0*seconds)); }
    catch(InterruptedException ie) {}
  }

  /** When run finishes, the thread exits. */

  public void run() {
    for(int i=0; i<loopLimit; i++) {
      System.out.println("Counter " + currentNum + ": " + i);
      pause(Math.random()); // Sleep for up to 1 second
    }
  }
}
```

Listing 16.4 `CounterTest.java`

```
/** Try out a few instances of the Counter class. */

public class CounterTest {
  public static void main(String[] args) {
    Counter c1 = new Counter(5);
    Counter c2 = new Counter(5);
    Counter c3 = new Counter(5);
    c1.start();
    c2.start();
    c3.start();
  }
}
```

Listing 16.5 `CounterTest` Output

```
Counter 0: 0
Counter 1: 0
Counter 2: 0
Counter 1: 1
Counter 2: 1
Counter 1: 2
Counter 0: 1
Counter 0: 2
Counter 1: 3
Counter 2: 2
Counter 0: 3
Counter 1: 4
Counter 0: 4
Counter 2: 3
Counter 2: 4
```

Mechanism 2: Put Behavior in the Driver Class, Which Must Implement Runnable

The second way to perform multithreaded computation is to implement the `Runnable` interface, construct an instance of `Thread` passing the current class (i.e., the `Runnable`) as an argument, and call that `Thread`'s `start` method. You put the actions you want executed in the `run` method of the main class; the thread's `run` method is ignored. This means that `run` has full access to all variables and methods of the class containing the `run` method. The actual `run` method executed (either the `run` method in the `Thread` class or the `run` method in the class implementing `Runnable`) is determined by the object passed to the thread constructor. If a `Runnable` is supplied, the thread's `start` method will use the `Runnable`'s `run` method instead of its own. Declaring that you implement `Runnable` serves as a guarantee to the thread that you have a public `run` method. Listing 16.6 shows an outline of this approach.

Listing 16.6 `ThreadedClass.java`

```
public class ThreadedClass extends AnyClass implements Runnable {
  public void run() {
    // Thread behavior here.
  }

  public void startThread() {
    Thread t = new Thread(this);
    t.start(); // Calls back to the run method in "this."
  }
  ...
}
```

You can invoke multiple threads of the class implementing the `Runnable` interface. In this case, each thread concurrently executes the *same* `run` method. Furthermore, each invocation of `run` owns a separate copy of the local variables, but you must carefully control access to any class variables, since they are shared among all thread instances of the class. Contention for common data is discussed in Section 16.2 (Race Conditions). If you want to access the thread instance to get private per-thread data, use `Thread.currentThread()`.

Listing 16.7 gives an example of this process. Note that the driver class (Listing 16.8) does not call `start` on the instantiated `Counter2`'s objects because `Counter2` is not a `Thread` and thus does not necessarily have a `start` method. The result (Listing 16.9) is substantially the same as with

the original `Counter`.

Listing 16.7 `Counter2.java`

```
/** A Runnable that counts up to a specified
 *  limit with random pauses in between each count.
 */

public class Counter2 implements Runnable {
  private static int totalNum = 0;
  private int currentNum, loopLimit;

  public Counter2(int loopLimit) {
    this.loopLimit = loopLimit;
    currentNum = totalNum++;
    Thread t = new Thread(this);
    t.start();
  }

  private void pause(double seconds) {
    try { Thread.sleep(Math.round(1000.0*seconds)); }
    catch(InterruptedException ie) {}
  }

  public void run() {
    for(int i=0; i<loopLimit; i++) {
      System.out.println("Counter " + currentNum + ": " + i);
      pause(Math.random()); // Sleep for up to 1 second.
    }
  }
}
```

Listing 16.8 `Counter2Test.java`

```
/** Try out a few instances of the Counter2 class. */

public class Counter2Test {
  public static void main(String[] args) {
    Counter2 c1 = new Counter2(5);
    Counter2 c2 = new Counter2(5);
    Counter2 c3 = new Counter2(5);
  }
}
```

Listing 16.9 `Counter2Test` Output

```
Counter 0: 0
Counter 1: 0
Counter 2: 0
Counter 1: 1
Counter 1: 2
Counter 0: 1
Counter 1: 3
Counter 2: 1
```

```
Counter 0: 2
Counter 0: 3
Counter 1: 4
Counter 2: 2
Counter 2: 3
Counter 0: 4
Counter 2: 4
```

In this particular instance, this approach probably seems more cumbersome than making a separate subclass of `Thread`. However, because the Java programming language does not have multiple inheritance, if your class already is a subclass of something else (say, `Applet`), then your class cannot also be a subclass of `Thread`. In such a case, if the thread needs access to the instance variables and methods of the main class, this approach works well, whereas the previous approach (a separate `Thread` subclass) requires some extra work to give the `Thread` subclass access to the applet's variables and methods (perhaps by passing along a reference to the `Applet`). When you are doing this from an `Applet`, the applet's `start` method is usually the place to create and start the threads.

16.2 Race Conditions

A class that implements `Runnable` could start more than one thread per instance. However, all of the threads started from that object will be looking at the *same* instance of that object. Per-thread data can be in local variables of `run`, but take care when accessing class instance variables or data in other classes because multiple threads can access the same variables concurrently. For instance, Listing 16.10 shows an *incorrect* counter applet. Before reading further, take a look at the `run` method and see if you can see what is wrong.

Listing 16.10 `BuggyCounterApplet.java`

```
import java.applet.Applet;
import java.awt.*;

/** Emulates the Counter and Counter2 classes, but this time
 *  from an applet that invokes multiple versions of its own run
 *  method. This version is likely to work correctly
 *  <B>except</B> when  an important customer is visiting.
 */

public class BuggyCounterApplet extends Applet
                                implements Runnable{
  private int totalNum = 0;
  private int loopLimit = 5;

  // Start method of applet, not the start method of the thread.
  // The applet start method is called by the browser after init is
  // called.
  public void start() {
    Thread t;
    for(int i=0; i<3; i++) {
      t = new Thread(this);
      t.start();
    }
  }
```

```
  private void pause(double seconds) {
    try { Thread.sleep(Math.round(1000.0*seconds)); }
    catch(InterruptedException ie) {}
  }
  public void run() {
    int currentNum = totalNum;
    System.out.println("Setting currentNum to " + currentNum);
    totalNum = totalNum + 1;
    for(int i=0; i<loopLimit; i++) {
      System.out.println("Counter " + currentNum + ": " + i);
      pause(Math.random());
    }
  }
}
```

Listing 16.11 Usual `BuggyCounterApplet` Output

```
> appletviewer BuggyCounterApplet.html
Setting currentNum to 0
Counter 0: 0
Setting currentNum to 1
Counter 1: 0
Setting currentNum to 2
Counter 2: 0
Counter 2: 1
Counter 1: 1
Counter 0: 1
Counter 2: 2
Counter 0: 2
Counter 1: 2
Counter 1: 3
Counter 0: 3
Counter 2: 3
Counter 1: 4
Counter 2: 4
Counter 0: 4
```

In the vast majority of cases, the output is correct as in Listing 16.11, and the temptation would be to assume that the class is indeed correct and to leave the class as it stands. In fact, however, the class suffers from the flawed assumption that no new thread will be created and read `totalNum` between the time the previous thread reads `totalNum` and increments the value. That is, the operation of this code depends on the previous thread "winning the race" to the increment operation. This assumption is unsafe, and, in fact, a small percent of the time the already activated thread will lose the race, as shown in Listing 16.12, obtained after running the same applet over and over *many* times.

Listing 16.12 Occasional `BuggyCounterApplet` Output

```
> appletviewer BuggyCounterApplet.html
Setting currentNum to 0
Counter 0: 0
Setting currentNum to 1
Setting currentNum to 1
Counter 0: 1
Counter 1: 0
```

```
Counter 1: 0
Counter 0: 2
Counter 0: 3
Counter 1: 1
Counter 0: 4
Counter 1: 1
Counter 1: 2
Counter 1: 3
Counter 1: 2
Counter 1: 3
Counter 1: 4
Counter 1: 4
```

Now, one obvious "solution" is to perform the updating of the thread number in a single step, as follows, rather than first reading the value, then incrementing the value a couple of lines later.

```
public void run() {
  int currentNum = totalNum++;
  System.out.println("Setting currentNum to " +
                     currentNum);
  for(int i=0; i<loopLimit; i++) {
    System.out.println("Counter " + currentNum +
                       ": " + i);
    pause(Math.random());
  }
}
```

Although the idea of performing the update in a single step is a good one, the Java Virtual Machine does not guarantee that this code really *will* be done in a single step. Sure, the program statement is a single line of Java source code, but who knows what goes on behind the scenes? The most likely scenario is that this approach simply shortens the race so the error occurs less frequently. Less frequent errors are *worse,* not better, because they are more likely to survive unnoticed until some critical moment. Fortunately, the Java programming language has a construct (`synchronized`) that lets you guarantee that a thread can complete a designated series of operations before another thread gets to execute any of the same operations. This concept is so important that we devote the entire next section to the topic.

16.3 Synchronization

Synchronization is a way to arbitrate contention for shared resources. When you synchronize a section of code, a "lock" (or "monitor") is set when the first thread enters that section of code. Unless the thread explicitly gives up the lock, no other thread can enter that section of code until the first one exits. In fact, synchronization can be even stronger than just locking a single section of code. A synchronized block has an `Object` as a tag, and once a thread enters a synchronized section of code, no other thread can enter *any* other section of code that is locked with the same tag.

Synchronizing a Section of Code

The way to protect a section of code that accesses shared resources is to place the code inside a `synchronized` block, as follows:

```
synchronized(someObject) {
  code
}
```

The synchronization statement tells the system to block other threads from entering this section of

code when the section is already in use. Once a thread enters the enclosed code, no other thread will be allowed to enter until the first thread exits or voluntarily gives up the lock through `wait`. Note that this lock does *not* mean that the designated code is executed uninterrupted; the thread scheduler can suspend a thread in the middle of the synchronized section to let another thread run. The key point is that the other thread will be executing a *different* section of code.

Also note that you lock sections of *code,* not *objects.* Every object has an associated flag that can be used as a monitor to control a synchronization lock. Using `someObject` as a label on the lock in no way "locks" `someObject`; the `synchronized` statement simply sets the flag. Other threads can still access the object, and race conditions are still possible if a different section of code accesses the same resources as those inside the `synchronized` block. However, you are permitted to use the same label on more than one block of code. Thus, once a thread enters a block that is synchronized on `someObject`, no other section of code that is also synchronized on `someObject` can run until either the first thread exits the synchronized section or the section of code explicitly gives up the lock. In fact, a single thread is allowed to hold the same lock multiple times, as when one synchronized block calls another block that is synchronized with the same lock.

Core Note

The `synchronized` construct locks sections of code, not objects.

Synchronizing an Entire Method

If you want to synchronize all of the code in a method, the Java programming language provides a shorthand method using the `synchronized` keyword, as follows:

```
public synchronized void someMethod() {
  body
}
```

This declaration tells the system to perform `someMethod` in a synchronized manner, using the current object instance (i.e., `this`) as the lock label. That is, once a thread starts executing `someMethod`, no other thread can enter the method or any other section of code that is synchronized on the current object (`this`) until the current thread exits from `someMethod` or gives up the lock explicitly with `wait`. Thus, the following are equivalent.

```
public synchronized void someMethod() {
  body
}

public void someMethod() {
  synchronized(this) {
    body
  }
}
```

Now, after all the dire warnings about race conditions, programmers are sometimes tempted to synchronize everything in sight. Unfortunately, this can have performance penalties and can result in coarser-grained threading. For an extreme case, consider what would happen if you marked the `run` method as `synchronized`: you'd be forcing your code to run completely serially!

A `static` method that specifies `synchronized` has the same effect as one whose body is synchronized on the class object. The `synchronized` modifier of a method is *not* inherited, so if `someMethod` is overridden in a subclass, the method is no longer synchronized unless the

`synchronized` keyword is repeated.

Core Warning

Overridden methods in subclasses do not inherit the `synchronized` declaration.

Common Synchronization Bug

When extending the `Thread` class, a common bug is to synchronize on `this` when sharing data across multiple instances of the threaded class. Consider the following example, where `SomeThreadedClass` contains a `static` object, `someSharedObject`, that has classwide visibility and is shared by all instances of the class. To avoid race conditions between each thread and to preserve data integrity of the shared object, the method, `doSomeOperation`, is synchronized. Does this block the other threads from entering the same method and modifying the shared data?

```
public class SomeThreadedClass extends Thread {
  private static RandomClass someSharedObject;
  ...
  public synchronized void doSomeOperation() {
    accessSomeSharedObject();
  }
  ...
  public void run() {
    while(someCondition) {
      doSomeOperation();    // Accesses shared data.
      doSomeOtherOperation();// No shared data.
    }
  }
}
```

Remember that a `synchronized` method declaration is equivalent to

```
synchronized(this){
  ...
}
```

But if there are multiple instances of `SomeThreadedClass`, then there are multiple *different* `this` references. For this situation, each `this` reference is unique and not an appropriate lock to protect the shared data. Correcting the bug is simply a matter of selecting an appropriate lock object.

Synchronize on the Shared Data

One solution is to synchronize on the shared data object. Again, the internal data fields of the object are not "locked" and can be modified. The object is just a label (acting as a tag), telling all other threads looking at the same label (tag) whether or not they can enter the block of code.

```
public void doSomeOperation() {
  synchronized(someSharedObject) {
    accessSomeSharedObject();
  }
}
```

Synchronize on the Class Object

Another solution is to synchronize on the `class` object. Java uses a unique `Class` object to represent information about each class. The syntax is as follows:

```
public void doSomeOperation() {
  synchronized(SomeThreadedClass.class) {
    accessSomeSharedObject();
  }
}
```

Note that if you synchronize a `static` method, the lock is the corresponding `Class` object, not `this`.

Synchronize on an Arbitrary Object

The last solution is to simply create a new, arbitrary, shared (`static`) object in `SomeThreadedClass` to function as the lock monitor. Specifically,

```
public class SomeThreadedClass extends Thread {
  private static Object lockObject = new Object();
  ...
  public void doSomeOperation() {
    synchronized(lockObject) {
      accessSomeSharedObject();
    }
  }
  ...
}
```

This last approach allows you to select an appropriate name for the lock to simplify code maintenance. In addition, if the class has multiple shared objects requiring synchronization in different methods, then creating a new lock object is probably the easiest to implement.

The synchronization problem that occurs with multiple `this` references is not a concern for a class that implements the `Runnable` interface. In this situation, all the associated threads are executing in the *same* `run` method of the class. Each thread has the same `this`. To avoid race conditions with the shared data, synchronization on `this` (either explicitly in a synchronized block or implicitly by means of the synchronized keyword) is perfectly appropriate.

16.4 Creating a Multithreaded Method

Occasionally, you will realize, after the fact, that some of the code that you wrote just begs to be threaded. Often this occurs because the program moved from a single- user environment to a multiuser environment. Here, processing each user serially is no longer acceptable. Or, another possibility is that your program invokes a computation-intensive method, leaving the user to simply wait until the process has completed before continuing. Not pretty.

In many cases, you can create a new class that runs the designated task in the background without modifying the original class at all. Conceptually, the idea is to inherit from the original class, make the class `Runnable`, and override the original method to produce the threaded behavior and invoke the original method located in the superclass. For this approach to work, the following two restrictions should hold:

- The method performs a task only; no side effects are produced on other variables.
- The method does not return data; the method simply returns `void`.

For the first requirement, the understanding is that the method does not modify other variables (either in the class or outside the class) that might be required by a subsequent thread requiring the data; otherwise, a race condition may be produced. The second requirement is also necessary because the calling program would otherwise simply wait for the return result anyway. If these two conditions do not hold, then most likely the original class will require modification to prevent race conditions. Real examples where this approach is common include sending messages to a log file, processing clients on an HTTP (Web) server, and handling mail through an SMTP server.

To illustrate this approach, assume that the class, `SomeClass`, contains a method, `foo`, with a single parameter argument, `randomArg`, and that `foo` produces no side effects on other variables or objects. The basic template for threading the `foo` method without modifying the original class is:

```
public class ThreadedSomeClass extends SomeClass
                               implements Runnable {
  //foo returns void, since no side effects are allowed.
  public void foo(RandomClass randomArg) {
    MyThread t = new MyThread(this, randomArg);
    t.start();
  }

  public void run() {
    MyThread t = (MyThread)Thread.currentThread();
    RandomClass randomArg =
                (RandomClass)t.getValueSavedEarlier();
    super.foo(randomArg);
  }
}
```

Wrapping a thread around the `foo` method (`super.foo`) permits `foo` to return immediately while the task continues to run in the background.

The basic template for `MyThread` is:

```
public class MyThread extends Thread {
  private Object data;
  public MyThread(Runnable runnable, Object data) {
    super(runnable);
    this.data = data;
  }
  public Object getValueSavedEarlier() {
    return data;
  }
}
```

The `foo` method argument, `randomArg`, is passed to the constructor of `MyThread` and saved as a *local* copy inside `MyThread` to avoid the possibility of a race condition. Saving a local copy of `randomArg` inside `foo` doesn't prevent the race condition because `randomArg` might be modified between the time that the thread is started and the time that the same thread is able to call `super.foo` in `run`. Synchronization in `foo` does not solve the problem either because program execution immediately jumps to another method. Once the thread is running, the data saved earlier in `MyThread` is retrieved and the original `foo`, located in the superclass, is then called.

The template for `MyThread` made one subtle assumption: that a "new" `RandomClass` object is passed into `foo` each time, or `RandomClass` is an immutable class (instance variables cannot change), for example, `String`, `Integer`, `Float`, `Double`, etc. `MyThread` only saves a

reference to the `randomArg` object; any changes to instance variables of `randomArg` are also seen by `data`. Thus, if `foo` doesn't pass in a new `RandomClass` object each time, then object cloning in `MyThread` is required to truly preserve a local copy, as in

```
this.data = data.clone();
```

The `clone` method in `RandomClass` should perform *deep* cloning (copying of all internal objects—primitive instance variables are always copied by default).

An example of creating a background process from an originally nonthreaded class method is shown in Listing 16.13 and Listing 16.14. The nonthreaded class, `RSAKey`, provides a method, `calculateKey`, to calculate an RSA public-private key pair, where the minimum number of digits for the public key is passed in as a parameter. The calculated RSA pair, along with the modulus, `N`, is printed to `System.out`. The source for `Primes.java` can be downloaded from the on-line archive at http://corewebprogramming.com/.

Listing 16.13 `RSAKey.java`

```
import java.math.BigInteger;

/** Calculate RSA public key pairs with a minimum number of
 *  required digits.
 */

public class RSAKey {
  private static final BigInteger ONE = new BigInteger("1");

  // Determine the encryption and decryption key.
  // To encrypt an integer M, compute R = M^e mod N.
  // To decrypt the encrypted integer R, compute M = R^d mod N,
  // where e is the public key and d is the private key.
  // For a discussion of the algorithm see section 7.4.3 of
  // Weiss's Data Structures and Problem Solving with Java.

  public void computeKey(String strNumDigits) {
    BigInteger p, q, n, m, encrypt, decrypt;
    int numDigits = Integer.parseInt(strNumDigits);
    if (numDigits%2==1) {
      numDigits++;
    }
    do {
      p = Primes.nextPrime(Primes.random(numDigits/2));
      q = Primes.nextPrime(Primes.random(numDigits/2));

      n = p.multiply(q);
      m = (p.subtract(ONE)).multiply(q.subtract(ONE));

      // Find encryption key, relatively prime to m.
      encrypt = Primes.nextPrime(Primes.random(numDigits));
      while (!encrypt.gcd(m).equals(ONE)) {
        encrypt = Primes.nextPrime(encrypt);
      }
      // Decrypt key is multiplicative inverse of encrypt mod m.
      decrypt = encrypt.modInverse(m);
```

```
    // Ensure public and private key have size numDigits.
    }while ((decrypt.toString().length() != numDigits) ||
            (encrypt.toString().length() != numDigits) ||
            (n.toString().length() != numDigits));
    System.out.println("\nN       => " + n);
    System.out.println("public  => " + encrypt);
    System.out.println("private => " + decrypt);
  }
}
```

Listing 16.14 ThreadedRSAKey.java

```
import java.io.*;

/** An example of creating a background process for an
 *  originally nonthreaded, class method. Normally,
 *  the program flow will wait until computeKey is finished.
 */

public class ThreadedRSAKey extends RSAKey implements Runnable {

  // Store strNumDigits into the thread to prevent race
  // conditions.
  public void computeKey(String strNumDigits) {
    RSAThread t = new RSAThread(this, strNumDigits);
    t.start();
  }

  // Retrieve the stored strNumDigits and call the original
  // method.  Processing is now done in the background.
  public void run() {
    RSAThread t = (RSAThread)Thread.currentThread();
    String strNumDigits = t.getStrDigits();
    super.computeKey(strNumDigits);
  }

  public static void main(String[] args){
    ThreadedRSAKey key = new ThreadedRSAKey();
    for (int i=0; i<args.length ; i++) {
        key.computeKey(args[i]);
    }
  }
}

class RSAThread extends Thread {
   protected String strNumDigits;

   public RSAThread(Runnable rsaObject, String strNumDigits) {
      super(rsaObject);
      this.strNumDigits = strNumDigits;
   }

   public String getStrDigits() {
     return(strNumDigits);
```

```
    }
}
```

Finding very large prime numbers is an extremely CPU-intensive process, so a natural extension is to thread the `calculateKey` method as a background process instead of waiting for the results to finish before performing another task. The class, `ThreadedRSAKey`, provides a threaded version of `calculateKey`, saving a local copy of the argument in an instance of `RSAThread`. An example of the output from the `ThreadedRSAKey` application is shown in Listing 16.15. As shown, calculation of the key pair with 5 digits (first command-line argument) is completed before the key pair with 25 digits (second command-line argument).

Prior to October 2000, the RSA algorithm was protected under United States Patent No. 4,405,829. Commercial use of the algorithm to encode and decode data without a license is now permitted.

Listing 16.15 `ThreadedRSAKey` Output

```
>java ThreadedRSAKey 50 8

N       => 22318033
public  => 99371593
private => 13439917

N       => 80587805972834425980516431184482416019941499846039
public  => 82145673210793850346670822324910704743113917481417
private => 54738576754079530157967908359197723401677283881913
```

An additional example of creating a multithreaded method in an originally nonthreaded class is given in Section 17.8 (Example: A Simple HTTP Server). Here, a simple HTTP server handles client connections serially. After the `handleConnection` method is threaded, the number of clients that can make a connection to the HTTP server in a fixed time interval is limited by how fast a `Socket` object is obtained and how quickly a new thread can be launched.

16.5 Thread Methods

The following subsections summarize the constructors, constants, and methods in the `Thread` class. Included also are the `wait`, `notify`, and `notifyAll` methods (which really belong to `Object`, not just to `Thread`).

Constructors

public Thread()

Using this constructor on the original `Thread` class is not very useful because once started, the thread will call its own `run` method, which is empty. However, a zero-argument constructor is commonly used for thread *subclasses* that have overridden the `run` method. Calling `new Thread()` is equivalent to calling `new Thread(null, null, "Thread-N")`, where *N* is automatically chosen by the system.

public Thread(Runnable target)

When you create a thread with a `Runnable` as a target, the target's `run` method will be used when start is called. This constructor is equivalent to `Thread(null, target, "Thread-N")`.

public Thread(ThreadGroup group, Runnable target)

This method creates a thread with the specified target (whose `run` method will be used), placing the thread in the designated `ThreadGroup` as long as that group's `checkAccess` method permits this action. A `ThreadGroup` is a collection of threads that can be operated on as a set; see Section 16.6 for details. In fact, all threads belong to a `ThreadGroup`; if one is not specified, then `ThreadGroup` of the thread creating the new thread is used. This constructor is equivalent to `Thread(group, target, "Thread-N")`.

public Thread(String name)

When threads are created, they are automatically given a name of the form `"Thread-N"` if a name is not specified. This constructor lets you supply your own name. Thread names can be retrieved with the `getName` method. This constructor is equivalent to `Thread(null, null, name)`, meaning that the calling thread's `ThreadGroup` will be used and the new thread's own `run` method will be called when the thread is started.

public Thread(ThreadGroup group, String name)

This constructor creates a thread in the given group with the specified name. The thread's own `run` method will be used when the thread is started. This constructor is equivalent to `Thread(group, null, name)`.

public Thread(Runnable target, String name)

This constructor creates a thread with the specified target and name. The target's `run` method will be used when the thread is started; this constructor is equivalent to `Thread(null, target, name)`.

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

This constructor creates a thread with the given group, target, and name. If the group is not `null`, the new thread is placed in the specified group unless the group's `checkAccess` method throws a `SecurityException`. If the group is `null`, the calling thread's group is used. If the target is not `null`, the passed-in thread's `run` method is used when started; if the target is `null`, the thread's own `run` is used.

Constants

public final int MAX_PRIORITY

This priority is the highest assignable priority of a thread.

public final int MIN_PRIORITY

This priority is the lowest assignable priority of a thread.

public final int NORM_PRIORITY

This priority given to the first user thread. Subsequent threads are automatically given the priority of their creating thread.

Typical implementations of the JVM define `MAX_PRIORITY` equal to 10, `NORM_PRIORITY` equal to 5, and `MIN_PRIORITY` equal to 1. However, be advised that the priority of a Java thread may map differently to the thread priorities supported by the underlying operating system. For instance, the Solaris Operating Environment supports 2^{31} priority levels, whereas Windows NT supports only 7 user priority levels. Thus, the 10 Java priority levels will map differently on the two operating systems. In fact, on Windows NT, two Java thread priorities can map to a single OS thread priority. If your program

design is strongly dependent on thread priorities, thoroughly test your implementation before production release.

Methods

public static int activeCount()

This method returns the number of active threads in the thread's `ThreadGroup` (and all subgroups).

public void checkAccess()

This method determines if the currently running thread has permission to modify the thread. The `checkAccess` method is used in applets and other applications that implement a `SecurityManager`.

public static native Thread currentThread()

This method returns a reference to the currently executing thread. Note that this is a `static` method and can be called by arbitrary methods, not just from within a `Thread` object.

public void destroy()

This method kills the thread without performing any cleanup operations. If the thread locked any locks, they remain locked. As of JDK 1.2, this method is not implemented.

public static void dumpStack()

This method prints a stack trace to `System.err`.

public static int enumerate(Thread[ ] groupThreads)

This method finds all active threads in the `ThreadGroup` belonging to the *currently executing* thread (not a particular specified thread), placing the references in the designated array. Use `activeCount` to determine the size of the array needed.

public ClassLoader getContextClassLoader() [Java 2]

This method returns the `ClassLoader` used by the thread to load resources and other classes. Unless explicitly assigned through `setContextClassLoader`, the `ClassLoader` for the thread is the same as the `ClassLoader` for the parent thread.

public final String getName()

This method gives the thread's name.

public final int getPriority()

This method gives the thread's priority. See `setPriority` for a discussion of the way Java schedules threads of different priorities.

public final ThreadGroup getThreadGroup()

This method returns the `ThreadGroup` to which the thread belongs. All threads belong to a group; if none is specified in the `Thread` constructor, the calling thread's group is used.

public void interrupt()

This method can force two possible outcomes. First, if the thread is executing the `join`, `sleep`, or `wait` methods, then the corresponding method will throw an `InterruptedException`. Second, the method sets a flag in the thread that can be detected by `isInterrupted`. In that case, the thread is responsible for checking the status of the interrupted flag and taking action if required. Calling `interrupted` resets the flag; calling `isInterrupted` does not reset the flag.

public static boolean interrupted()

This static method checks whether the *currently executing* thread is interrupted (i.e., has its interrupted flag set through `interrupt`) and clears the flag. This method differs from `isInterrupted`, which only checks whether the *specified* thread is interrupted.

public final native boolean isAlive()

This method returns `true` for running or suspended threads, `false` for threads that have completed the `run` method. A thread calling `isAlive` on itself is not too useful, since if you can call *any* method, then you are obviously alive. The method is used by external methods to check the liveness of thread references they hold.

public final boolean isDaemon()

This method determines whether the thread is a daemon thread. A Java program will exit when the only active threads remaining are daemon threads. A thread initially has the same status as the creating thread, but this can be changed with `setDaemon`. The garbage collector is an example of a daemon thread.

public boolean isInterrupted()

This method checks whether the thread's interrupt flag has been set and does so without modifying the status of the flag. You can reset the flag by calling `interrupted` from within the `run` method of the flagged thread.

public final void join() throws InterruptedException

public final synchronized join(long milliseconds) throws InterruptedException

public final synchronized join(long milliseconds,int nanoseconds) throws InterruptedException

This method suspends the calling thread until either the specified timeout has elapsed or the thread on which `join` was called has terminated (i.e., would return `false` for `isAlive`). For those not familiar with the terminology of "thread joining," naming this method `sleepUntilDead` would have been clearer. This method provides a convenient way to wait until one thread has finished before starting another, but without polling or consuming too many CPU cycles. A thread should never try to `join` itself; the thread would simply wait forever (a bit boring, don't you think?). For more complex conditions, use `wait` and `notify` instead.

public final native void notify()

public final native void notifyAll()

Like `wait`, the `notify` and `notifyAll` method are really methods of `Object`, not just of `Thread`. The first method (`notify`) wakes up a single thread, and the second method (`notifyAll`) wakes all threads that are waiting for the specified object's lock. Only code that holds an object's lock (i.e., is inside a block of code synchronized on the

object) can send a `notify` or `notifyAll` request; the thread or threads being notified will not actually be restarted until the process issuing the notify request gives up the lock. See the discussion of `wait` for more details.

public void run()

In this method the user places the actions to be performed. When `run` finishes, the thread exits. The method creating the thread should not call `run` directly on the thread object; the method should instead call `start` on the thread object.

public void setContextClassLoader(ClassLoader loader) [Java 2]

This method sets the `ClassLoader` for the thread. The `ClassLoader` defines the manner in which the Java Virtual Machine loads classes and resources used by the thread. If a `SecurityManager` prevents the assignment of a different `ClassLoader`, then a `SecurityException` is thrown. If not explicitly assigned, the `ClassLoader` for the thread assumes the same loader as for the parent thread.

public final void setDaemon(boolean becomeDaemon)

This method sets the daemon status of the thread. A thread initially has the same status as that of the creating thread, but this status can be changed with `setDaemon`. A Java program will exit when the only active threads remaining are daemon threads.

public final void setName(String threadName)

This method changes the name of the thread.

public final void setPriority(int threadPriority)

This method changes the thread's priority; higher-priority threads are supposed to be executed in favor of lower-priority ones. Legal values range from `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`. A thread's default priority is the same as the creating thread. The priority cannot be set higher than the `MAX_PRIORITY` of the thread's thread group. Be careful because starvation can occur: almost all current implementations use a completely preemptive scheduler where lower-priority threads will *never* be executed unless the higher-priority threads terminate, sleep, or wait for I/O.

public static native void sleep(long milliseconds) throws InterruptedException

public static native void sleep(long milliseconds, int nanoseconds) throws InterruptedException

This method does a nonbusy wait for at least the specified amount of time unless the thread is interrupted. Since `sleep` is a `static` method, `sleep` is used by nonthreaded applications as well.

Core Approach

You can use `Thread.sleep` from any method, not just in threads.

public synchronized native void start()

This method is called to initialize the thread and then call `run`. If the thread is created with

a `null` target (see the constructors earlier in this section), then `start` calls its own `run` method. But if some `Runnable` is supplied, then `start` calls the `run` method of that `Runnable`.

Note that applets also have a `start` method that is called after `init` is finished and before the first call to `paint`. Don't confuse the applet's `start` method with the `start` method of threads, although the applet's `start` method is a convenient place for applets to initiate threads.

public final void wait() throws InterruptedException

public final void wait(long milliseconds) throws InterruptedException

public final void wait(long milliseconds, int nanoseconds) throws InterruptedException

These methods give up the lock and suspend the current thread. The thread is restarted by `notify` or `notifyAll`. This method is actually a member of `Object`, not just of `Thread`, but can only be called from within a synchronized method or block of code. For example,

```
public synchronized void someMethod() {
  doSomePreliminaries();
  while (!someContinueCondition()) {
    try {
      // Give up the lock and suspend ourselves.
      // We'll rely on somebody else to wake us up,
      // but will check someContinueCondition before
      // proceeding, just in case we get woken up
      // for the wrong reason.
      wait();
    } catch(InterruptedException ie) {}
  }
  continueOperations();
}
```

You call `wait` on the object that is used to set up (tag) the synchronized block, so if the synchronization object is not the current object (`this`), simply call `wait` on the synchronizing object explicitly, as in the following example.

```
public void someOtherMethod() {
  doSomeUnsynchronizedStuff();
  synchronized(someObject) {
    doSomePreliminaries();
    while (!someContinueCondition()) {
      try {
        someObject.wait();
      } catch(InterruptedException ie) {}
    }
    continueOperations();
  }
  doSomeMoreUnsynchronizedStuff();
}
```

See Listing 16.16 for an example of using `wait` and `notify`.

public static native void yield()

If two threads of the same priority are running and neither thread sleeps or waits for I/O, they may or may not alternate execution. Using `yield` gives up execution to any other process of the same priority that is waiting and is a good practice to ensure that time-slicing takes place. Although leaving the details of time-slicing to the implementation definitely seems appropriate, in our opinion, not requiring that threads be time-sliced is a significant drawback in an otherwise excellent thread specification. Fortunately, virtually all Java 2 implementations perform time-slicing properly.

Stopping a Thread

In the original Java thread model, a `stop` method was included to terminate a live thread. This method was immediately deprecated in the next release of the Java platform since termination of a thread by `stop` immediately released all associated locks, potentially placing these locks in an inconsistent state for the remaining threads; housecleaning was not performed. The correct approach for terminating a thread is to set a flag that causes the `run` method to complete. For example,

```
class ThreadExample implements Runnable {
  private boolean running;
  public ThreadExample()
     Thread thread = new Thread(this);
     thread.start();
  }
  public void run(){
    running = true;
    while (running) {
    ...
    }
  }
  public void setRunning(boolean running) {
    this.running = running;
  }
}
```

Setting the flag `running` to `false` anywhere in the program will terminate the `while` loop the next time the thread is granted CPU time and the loop test is performed. This approach works well for a single thread executing in the `run` method or if all threads executing in the `run` method should be terminated through the same single flag.

For classes that inherit from `Thread`, each instance of the class contains a separate `run` method. The best approach for this situation is to add a public method to the class for setting an internal flag to terminate the `run` method. Consider the template shown in Listing 16.16 (originally written by Scott Oaks and published in *Java Report*, Vol. 2, No. 11, 1997, p. 87). The class provides a public method, `setState`, to change the thread state to `STOP`, `RUN`, or `WAIT`. In the `run` method of the class, the state of the thread is continuously polled, and when the state is changed to `STOP` (either through the object itself or through another object invoking the public `setState` method), the `while` loop terminates and the instance of the thread dies. In addition, the thread can conveniently be placed in a waiting state (`WAIT`), where the thread will remain until interrupted, `thread.interrupt()`, or the state is set to `RUN`. Each instance of a thread from the class can be controlled individually through this technique.

Listing 16.16 `StoppableThread.java`

```
/** A template to control the state of a thread through setting
 *  an internal flag.
```

```
 */

public class StoppableThread extends Thread {

   public static final int STOP    = 0;
   public static final int RUN     = 1;
   public static final int WAIT    = 2;
   private int state = RUN;

  /** Public method to permit setting a flag to stop or
   *  suspend the thread.  The state is monitored through the
   *  corresponding checkState method.
   */

   public synchronized void setState(int state) {
      this.state = state;
      if (state==RUN) {
         notify();
      }
   }

  /** Returns the desired state of the thread (RUN, STOP, WAIT).
   *  Normally, you may want to change the state or perform some
   *  other task if an InterruptedException occurs.
   */

   private synchronized int checkState() {
      while (state==WAIT) {
        try {
          wait();
        } catch (InterruptedException e) { }
      }
      return state;
   }

  /** An example of thread that will continue to run until
   *  the creating object tells the thread to STOP.
   */

   public void run() {
      while (checkState()!=STOP) {
         ...
      }
   }
}
```

Stopping Threads in an Applet

Applets have a `stop` method that is called whenever the Web page is exited or, in the case of Netscape, when the browser is resized. Depending on the browser, threads are not automatically stopped when transferring to another Web page, so you should normally put code in the applet's `stop` method to set flags to terminate all active threads.

Core Approach

If you make a multithreaded applet, you should halt the threads in the applet's `stop` method, restarting them in the applet's `start` method.

16.6 Thread Groups

The `ThreadGroup` class provides a convenient mechanism for controlling sets of threads. A `ThreadGroup` can contain other thread groups in addition to threads, letting you arrange groups hierarchically. The constructors and methods for the `ThreadGroup` class are summarized in the following subsections.

Constructors

public ThreadGroup(String groupName)

This constructor creates a named `ThreadGroup` that belongs to the same group as the thread that called the constructor.

public ThreadGroup(ThreadGroup parent, String groupName)

This constructor creates a named `ThreadGroup` that belongs to the specified parent group.

Methods

public synchronized int activeCount()

This method gives the number of active threads directly or indirectly in the thread group as of the time the method call was initiated.

public synchronized int activeGroupCount()

This method gives the number of active thread groups in the group as of the time the method call was initiated.

public final void checkAccess()

This method determines whether the calling thread is allowed to modify the group. The `checkAccess` method is used by applets and applications with a custom `SecurityManager`.

public final synchronized void destroy()

If the group contains no threads, the group (and any empty subgroups) is destroyed and removed from the parent group. The method throws an `IllegalThreadStateException` if the group contains any threads when this method is called.

public int enumerate(Thread[ ] threads)

public int enumerate(Thread[ ] threads, boolean recurse)

public int enumerate(ThreadGroup[ ] groups)

public int enumerate(ThreadGroup[ ] groups, boolean recurse)

These methods copy the references of active threads or thread groups into the specified

array. If the `recurse` flag is `true`, the method recursively descends child groups.

public final int getMaxPriority()

This method returns the maximum priority that can be assigned to any thread in the group. See `setMaxPriority` for setting the value.

public final String getName()

This method returns the name of the group.

public final ThreadGroup getParent()

This method returns the parent group. The first group created in the system will have a `null` parent.

public final void interrupt()

This method invokes the `interrupt` method of each thread in the group and all subgroups if permitted by the security manager.

public final boolean isDaemon()

This methods tells you whether the group is a daemon group. Daemon groups are automatically destroyed when they are empty.

public final boolean isDestroyed()

This method determines whether the thread group is destroyed. A daemon group is automatically destroyed if the last thread is no longer running or the last subgroup is destroyed. Adding a `Thread` or `ThreadGroup` to a destroyed thread group is not permitted.

public synchronized void list()

This method prints all the threads and subgroups in the group to `System.out` and is a useful debugging tool.

public final boolean parentOf(ThreadGroup descendant)

This method determines whether the group is an ancestor (not necessarily the direct parent) of the specified descendant group.

public final void setDaemon(boolean becomeDaemon)

A group automatically gets the daemon status of the parent group; this method can change that initial status. A daemon group is automatically destroyed when the group becomes empty.

public final synchronized void setMaxPriority(int max)

This method gives the maximum priority that any thread in the group can be explicitly given by `setPriority`. Threads in the thread group with higher priorities remain unchanged. The threads can still inherit a higher priority from their creating thread, however.

public void uncaughtException(Thread thread,Throwable error)

When a thread throws an exception that is not caught, execution flow is transferred to the `uncaughtException` method first. The default behavior is to print a stack trace.

16.7 Multithreaded Graphics and Double Buffering

One common application of threads is to develop dynamic graphics. Various standard approaches are available, each of which has various advantages and disadvantages.

- **Redraw Everything in paint.** This approach is simple and easy, but if things change quickly, drawing everything in `paint` is slow and can result in a flickering display.
- **Implement the Dynamic Part as a Separate Component.** This approach is relatively easy and eliminates the flickering problem but requires a `null` layout manager and can be quite slow.
- **Have Routines Other Than paint Draw Directly.** This approach is easy, efficient, and flicker free but results in "transient" drawing that is lost the next time the screen is redrawn.
- **Override update and Have paint Do Incremental Updating.** This approach eliminates the flicker and improves efficiency somewhat but requires the graphics to be nonoverlapping.
- **Use Double Buffering.** This approach is the most efficient option and has no problem with overlapping graphics. However, double buffering is more complex than other approaches and requires additional memory resources.

Redraw Everything in paint

A common technique in graphical Java programs is to have processes set parameters that describe the appearance of the window, rather than drawing the graphics themselves. The various routines call `repaint` to schedule a call to `paint`, and `paint` does the necessary drawing based on the parameters that have been set. The `repaint` method actually calls `update`, which clears the screen and then calls `paint` with the appropriate `Graphics` object. The `paint` method also is called automatically after the applet is initialized, whenever part of the applet is obscured and reexposed and when certain other resizing or layout events occur. A rectangular region can also be supplied when the `repaint` method is called to determine the clipping region given to the `Graphics` object used in `update` and `paint`. However, determining the proper area is difficult, and the entire region, not just some particular graphical items, is still redrawn.

For instance, suppose you are developing a user interface for a naval simulation system and you need to animate the icons that represent the ships. A background thread (or threads) could be updating the positions, then periodically invoking `repaint`. The `paint` method simply loops over all the simulation objects, drawing them in their current positions, as shown in the simplified example of Listing 16.17.

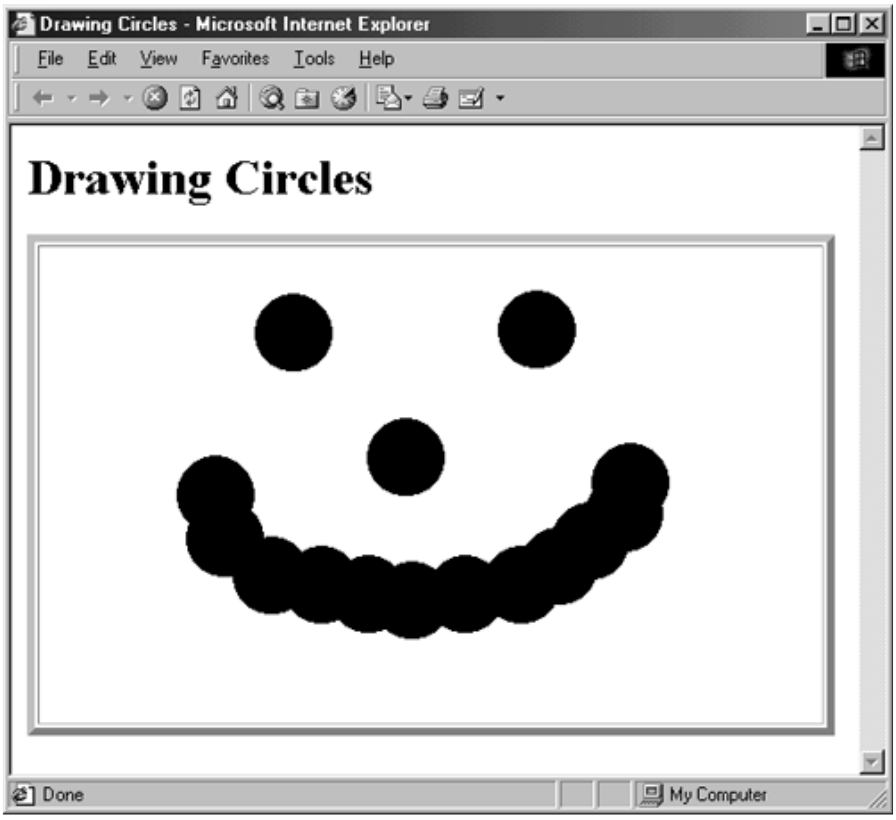
Listing 16.17 `ShipSimulation.java`

```
import java.applet.Applet;
import java.awt.*;

public class ShipSimulation extends Applet implements Runnable {
  ...
    public void run() {
    Ship s;
    for(int i=0; i<ships.length; i++) {
      s = ships[i];
      s.move(); // Update location.
    }
    repaint();
  }
```

```
  ...

  public void paint(Graphics g) {
    Ship s;
    for(int i=0; i<ships.length; i++) {
      s = ships[i];
      g.draw(s); // Draw at current location.
    }
  }
}
```

Alternatively, you might only initiate changes to the interface at user request but still redraw everything each time. For example, suppose that you want to draw some sort of image wherever the user clicks the mouse. Listing 16.18 shows a solution using the "store and redraw" approach, with a sample result shown in Figure 16-1. Here, when the user clicks the mouse, a new `SimpleCircle` object is created and added to the data structure. The `repaint` method is immediately called and completely redraws each `SimpleCircle` in the data structure on the applet.

Figure 16-1. By storing results in a permanent data structure and redrawing the whole structure every time `paint` is invoked, you cause the drawing to persist even after the window is covered up and reexposed.



Listing 16.18 `DrawCircles.java`

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

/** An applet that draws a small circle where you click.*/
```

```
public class DrawCircles extends Applet {

  private Vector circles;

  /** When you click the mouse, create a SimpleCircle,
   *  put it in the Vector, and tell the system
   *  to repaint (which calls update, which clears
   *  the screen and calls paint).
   */

  private class CircleDrawer extends MouseAdapter {
    public void mousePressed(MouseEvent event) {
      circles.addElement(
         new SimpleCircle(event.getX(),event.getY(),25));
      repaint();
    }
  }

  public void init() {
    circles = new Vector();
    addMouseListener(new CircleDrawer());
    setBackground(Color.white);
  }

  /** This loops down the available SimpleCircle objects,
   *  drawing each one.
   */

  public void paint(Graphics g) {
    SimpleCircle circle;
    for(int i=0; i<circles.size(); i++) {
      circle = (SimpleCircle)circles.elementAt(i);
      circle.draw(g);
    }
  }
}
```

Listing 16.19 presents the `SimpleCircle` class used in `DrawCircles`.

Listing 16.19 `SimpleCircle.java`

```
import java.awt.*;

/** A class to store an x, y, and radius, plus a draw method.
 */

public class SimpleCircle {
  private int x, y, radius;

  public SimpleCircle(int x, int y, int radius) {
    setX(x);
    setY(y);
    setRadius(radius);
```

```
  }

  /** Given a Graphics, draw the SimpleCircle
   *  centered around its current position.
   */

  public void draw(Graphics g) {
    g.fillOval(x -- radius, y -- radius,
               radius * 2, radius * 2);
  }

  public int getX() { return(x); }

  public void setX(int x) { this.x = x; }

  public int getY() { return(y); }

  public void setY(int y) { this.y = y; }

  public int getRadius() { return(radius); }

  public void setRadius(int radius) {
    this.radius = radius;
  }
}
```

Pros and Cons

This approach is quite simple and was well suited to the circle-drawing application. However, the approach is poorly suited to applications that have complicated, time-consuming graphics (because all or part of the window gets redrawn each time) or for ones that have frequent changes (because of the flicker caused by the clearing of the screen).

Implement the Dynamic Part as a Separate Component

Components know how to update themselves, so the `Container` that uses a `Component` need not explicitly draw it. For instance, in the simulation example above, the ships could be implemented as custom subclasses of `Canvas` and placed in a window with a `null` layout manager. To update the positions, the main simulation process could simply call `move` on the components. Similarly, drawing triggered by user events is a simple matter of creating a `Component`, setting its x and y locations, and adding the component to the current `Container`. An example of drawing circles this way is shown in Section 13.9 (Serializing Windows).

Pros and Cons

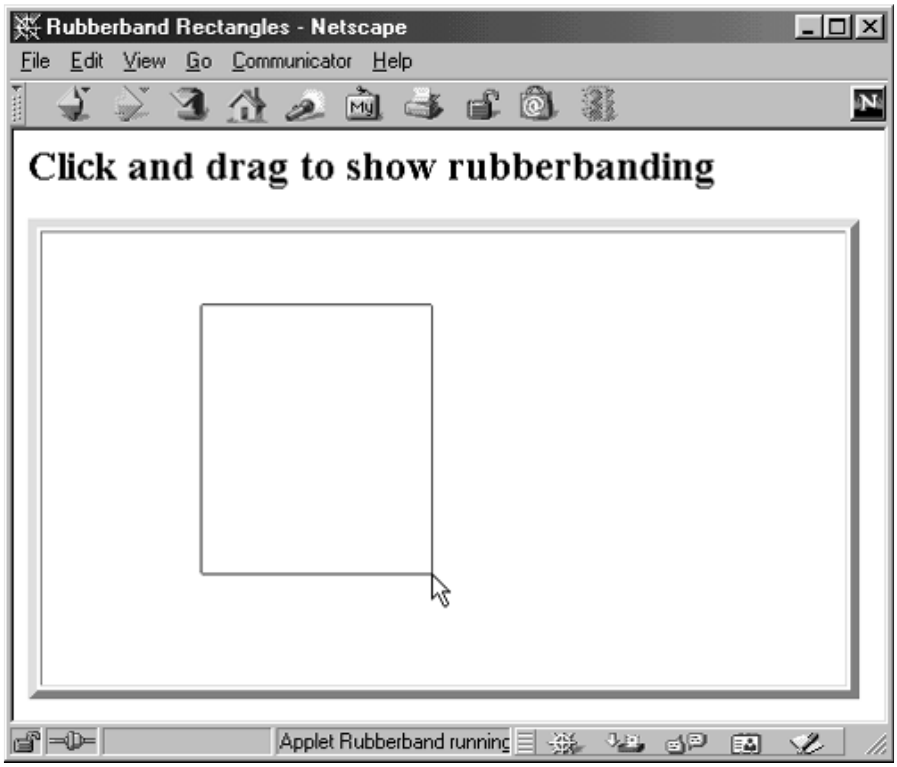
In some situations, this approach is easier than doing things explicitly in `paint` because the movement can be controlled directly by the `Component`, with few changes required in the code for the `Container`. However, this approach suffers from problems with overlapping components and also suffers from poor speed performance and memory usage. Creating a separate `Canvas` for each drawing is expensive, and moving the component involves more than just redrawing.

Have Routines Other Than paint Draw Directly

In some cases, you don't want to bother to call `paint` *at all,* but you want to do drawing directly. You can do the drawing by getting the `Graphics` object, using `getGraphics`, setting the drawing

mode to use XOR, then directly calling the `drawXxx` methods of the `Graphics` object. You can erase the original drawing by drawing in the same location a second time, at least as long as multiple drawing does not overlap. To illustrate this technique, Listing 16.20 creates a simple `Applet` that lets the user create and stretch "rubberband" rectangles. Figure 16-2 shows a typical result.

Figure 16-2. Direct drawing from threads or event handlers is easy to implement and is very fast but gives transient results.



Listing 16.20 `Rubberband.java`

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

/** Draw "rubberband" rectangles when the user drags
 *  the mouse.
 */

public class Rubberband extends Applet {
  private int startX, startY, lastX, lastY;

  public void init() {
    addMouseListener(new RectRecorder());
    addMouseMotionListener(new RectDrawer());
    setBackground(Color.white);
  }

  /** Draw the rectangle, adjusting the x, y, w, h
   *  to correctly accommodate for the opposite corner of the
   *  rubberband box relative to the start position.
   */
```

```
  private void drawRectangle(Graphics g, int startX, int startY,
                             int stopX, int stopY ) {
    int x, y, w, h;
    x = Math.min(startX, stopX);
    y = Math.min(startY, stopY);
    w = Math.abs(startX -- stopX);
    h = Math.abs(startY -- stopY);
    g.drawRect(x, y, w, h);
  }
  private class RectRecorder extends MouseAdapter {

    /** When the user presses the mouse, record the
     *  location of the top--left corner of rectangle.
     */

    public void mousePressed(MouseEvent event) {
      startX = event.getX();
      startY = event.getY();
      lastX = startX;
      lastY = startY;
    }

    /** Erase the last rectangle when the user releases
     *  the mouse.
     */

    public void mouseReleased(MouseEvent event) {
      Graphics g = getGraphics();
      g.setXORMode(Color.lightGray);
      drawRectangle(g, startX, startY, lastX, lastY);
    }
  }

  private class RectDrawer extends MouseMotionAdapter {

    /** This draws a rubberband rectangle, from the location
     *  where the mouse was first clicked to the location
     *  where the mouse is dragged.
     */

    public void mouseDragged(MouseEvent event) {
      int x = event.getX();
      int y = event.getY();

      Graphics g = getGraphics();
      g.setXORMode(Color.lightGray);
      drawRectangle(g, startX, startY, lastX, lastY);
      drawRectangle(g, startX, startY, x, y);

      lastX = x;
      lastY = y;
    }
```

```
  }
}
```

Pros and Cons

This approach (direct drawing instead of setting variables that `paint` will use) is appropriate for temporary drawing; direct drawing is simple and efficient. However, if `paint` is ever triggered, this drawing will be lost. So, this approach is not appropriate for permanent drawing.

Override update and Have paint Do Incremental Updating

Suppose that some graphical objects need to be moved around on the screen. Suppose further that the objects never overlap one another. In such a case, rather than clearing the screen and completely redrawing the new situation, you could use the `paint` method to erase the object at its old location (say, by drawing a solid rectangle in the background color) and then redraw the object at the new location, saving the time required to redraw the rest of the window. The `paint` method could be triggered from some mouse or keyboard event or by a background thread calling the applet's `repaint` method. For this technique to work without flickering, however, you must define a new version of `update` that does not clear the screen before calling `paint`.

```
public void update(Graphics g) {
  paint(g);
}
```

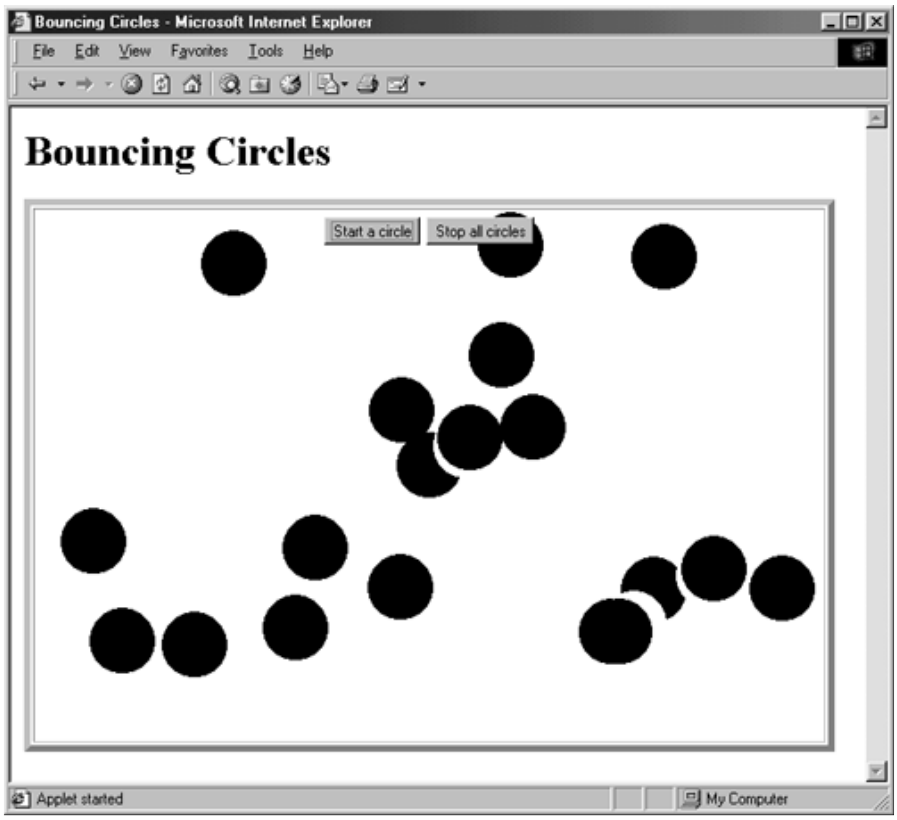
Also, because `paint` runs from the same foreground thread that watches for mouse and keyboard events, keeping the time spent in `paint` as short as possible is important to prevent the system from being unresponsive to buttons, scrollbars, and the like.

Core Approach

Minimize the amount of time spent in `paint` because event handling executes in the same thread.

To illustrate this approach, Listing 16.21 shows an applet that lets you create any number of small circles that bounce around in the window. As can be seen in Figure 16-3, the incremental drawing works well everywhere except where the circles overlap, where erasing one circle can accidentally erase part of another circle.

Figure 16-3. Incremental updating from `paint` can be flicker free and relatively fast, but it does not easily handle overlapping items.



Listing 16.21 Bounce.java

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

/** Bounce circles around on the screen. Doesn't use double
 *  buffering, so has problems with overlapping circles.
 *  Overrides update to avoid flicker problems.
 */

public class Bounce extends Applet implements Runnable,
                                               ActionListener {
  private Vector circles;
  private int width, height;
  private Button startButton, stopButton;
  private Thread animationThread = null;

  public void init() {
    setBackground(Color.white);
    width = getSize().width;
    height = getSize().height;
    circles = new Vector();
    startButton = new Button("Start a circle");
    startButton.addActionListener(this);
    add(startButton);
    stopButton = new Button("Stop all circles");
    stopButton.addActionListener(this);
```

```
    add(stopButton);
  }

  /** When the "start" button is pressed, start the animation
   *  thread if it is not already started. Either way, add a
   *  circle to the Vector of circles that are being bounced.
   *  <P>
   *  When the "stop" button is pressed, stop the thread and
   *  clear the Vector of circles.
   */

  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == startButton) {
      if (circles.size() == 0) {
        // Erase any circles from previous run.
        getGraphics().clearRect(0, 0, getSize().width,
                                      getSize().height);
        animationThread = new Thread(this);
        animationThread.start();
      }
      int radius = 25;
      int x = radius + randomInt(width -- 2 * radius);
      int y = radius + randomInt(height -- 2 * radius);
      int deltaX = 1 + randomInt(10);
      int deltaY = 1 + randomInt(10);
      circles.addElement(new MovingCircle(x, y, radius, deltaX,
                                          deltaY));
    } else if (event.getSource() == stopButton) {
      if (animationThread != null) {
        animationThread = null;
        circles.removeAllElements();
      }
    }
    repaint();
  }

  /** Each time around the loop, call paint and then take a
   *  short pause. The paint method will move the circles and
   *  draw them.
   */
  public void run() {
    Thread myThread = Thread.currentThread();
    // Really while animationThread not null
    while(animationThread==myThread) {
      repaint();
      pause(100);
    }
  }

  /** Skip the usual screen--clearing step of update so that
   *  there is no flicker between each drawing step.
   */
```

```
  public void update(Graphics g) {
    paint(g);
  }

  /** Erase each circle's old position, move it, then draw it
   *  in new location.
   */

  public void paint(Graphics g) {
    MovingCircle circle;
    for(int i=0; i<circles.size(); i++) {
      circle = (MovingCircle)circles.elementAt(i);
      g.setColor(getBackground());
      circle.draw(g);  // Old position.
      circle.move(width, height);
      g.setColor(getForeground());
      circle.draw(g);  // New position.
    }
  }

  // Returns an int from 0 to max (inclusive),
  // yielding max + 1 possible values.

  private int randomInt(int max) {
    double x =
      Math.floor((double)(max + 1) * Math.random());
    return((int)(Math.round(x)));
  }

  // Sleep for the specified amount of time.

  private void pause(int milliseconds) {
    try {
      Thread.sleep((long)milliseconds);
    } catch(InterruptedException ie) {}
  }
}
```

The `Bounce` class makes use of `MovingCircle`, a class that encapsulates the movement of the circle as well as the position and size, which are inherited from `SimpleCircle`. `MovingCircle` is shown in Listing 16.22.

Listing 16.22 `MovingCircle.java`

```
/** An extension of SimpleCircle that can be moved around
 *  according to deltaX and deltaY values. Movement will
 *  continue in a given direction until the edge of the circle
 *  reaches a wall, when it will "bounce" and move in the other
 *  direction.
 */
public class MovingCircle extends SimpleCircle {
  private int deltaX, deltaY;

  public MovingCircle(int x, int y, int radius, int deltaX,
```

```
                    int deltaY) {
    super(x, y, radius);
    this.deltaX = deltaX;
    this.deltaY = deltaY;
  }

  public void move(int windowWidth, int windowHeight) {
    setX(getX() + getDeltaX());
    setY(getY() + getDeltaY());
    bounce(windowWidth, windowHeight);
  }

  private void bounce(int windowWidth, int windowHeight) {
    int x = getX(), y = getY(), radius = getRadius(),
        deltaX = getDeltaX(), deltaY = getDeltaY();
    if ((x -- radius < 0) && (deltaX < 0)) {
      setDeltaX(--deltaX);
    } else if ((x + radius > windowWidth) && (deltaX > 0)) {
      setDeltaX(--deltaX);
    }
    if ((y --radius < 0) && (deltaY < 0)) {
      setDeltaY(--deltaY);
    } else if((y + radius > windowHeight) && (deltaY > 0)) {
      setDeltaY(--deltaY);
    }
  }

  public int getDeltaX() {
    return(deltaX);
  }

  public void setDeltaX(int deltaX) {
    this.deltaX = deltaX;
  }

  public int getDeltaY() {
    return(deltaY);
  }

  public void setDeltaY(int deltaY) {
    this.deltaY = deltaY;
  }
}
```

Pros and Cons

This approach takes a bit more effort than previous ones and requires that your objects be nonoverlapping. However, incremental updating lets you implement "permanent" drawing more quickly than redrawing everything every time.

Use Double Buffering

Consider a scenario that involves many different moving objects or overlapping objects. Drawing several different objects individually is expensive, and, if the objects overlap, reliably erasing them in

their old positions is difficult. In such a situation, double buffering is often employed. In this approach, an off-screen image (pixmap) is created, and all of the drawing operations are done into this image. The actual "drawing" of the window consists of a single step: draw the image on the screen. As before, the `update` method is overridden to avoid clearing of the screen. Even though the image replaces the applet graphics each time something is drawn to the image, the default implementation of `update` would still clear the applet in the background color before drawing the image.

Double buffering is automatically built into Swing components. However, for AWT components, double buffering must be provided by the programmer. Although different variations are available for double buffering, the basic approach involves five steps.

1. Override update to simply call `paint`. This step prevents the flicker that would normally occur each time `update` clears the screen before calling `paint`.

2. Allocate an `Image` by using `createImage`. Since this image uses native window-system support, the allocation cannot be done until a window actually appears. For instance, you should call `createImage` in an applet from `init` (or later), not in the direct initialization of an instance variable. For an application, you should wait until after the initial frame has been displayed (e.g., by `setVisible`) before calling `createImage`. However, calling `createImage` for a component that has no peer results in the return of `null`. Normally, the peer is created when the component is first displayed. Or, you can call `addNotify` to force creation of the peer prior to `setVisible`.

   Core Warning

   Calling `createImage` from a component that is not visible results in `null`.

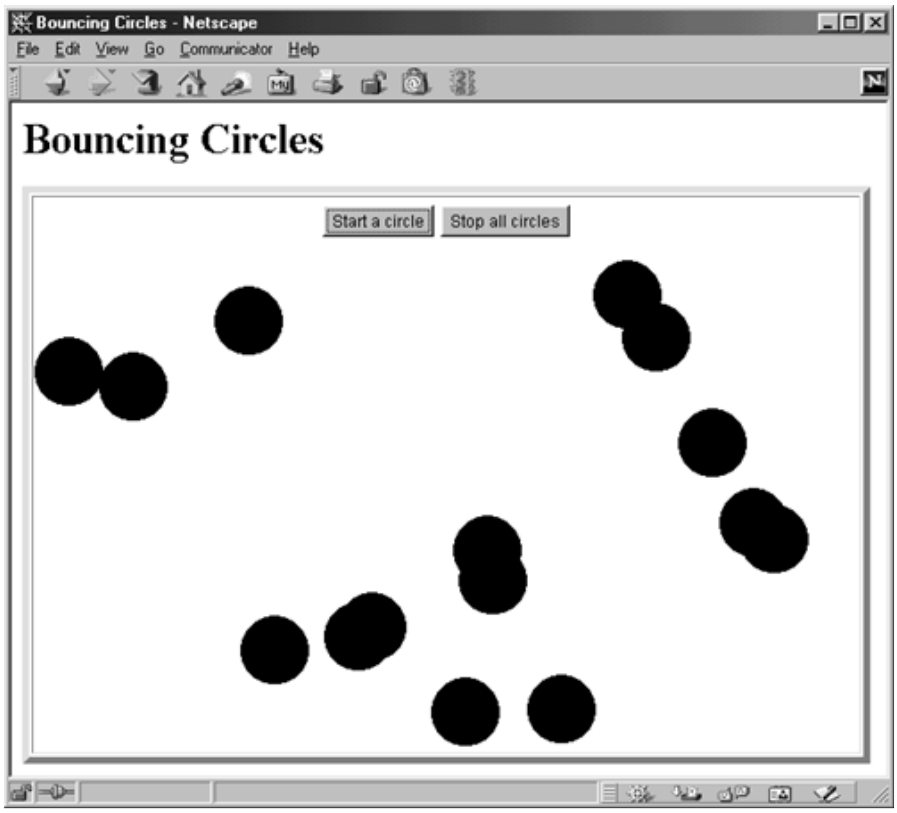
3. Look up the `Graphics object by using getGraphics`. Unlike the case with windows, where you need to look up the `Graphics` context each time you draw, with images you can reliably get the `Graphics` object associated with the image once, store the object in a reference, and reuse the same reference thereafter.

4. **For each step, clear the image and redraw all objects.** This step will be dramatically faster than drawing onto a visible window. In Swing, you achieve this result by calling `super.paintComponent`.

5. **Draw the off-screen image onto the window.** Use `drawImage` for this step.

Listing 16.23 shows a concrete implementation of this approach. `DoubleBufferBounce` changes the previous `Bounce` applet to use double buffering, resulting in improved performance and eliminating the problems with overlapping circles. Figure 16-4 shows the result.

Figure 16-4. Double buffering allows fast, flicker-free updating of possibly overlapping images, but at the cost of some complexity and additional memory usage.



Listing 16.23 `DoubleBufferBounce.java`

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

/** Bounce circles around on the screen, using double buffering
 *  for speed and to avoid problems with overlapping circles.
 *  Overrides update to avoid flicker problems.
 */

public class DoubleBufferBounce extends Applet implements
                                        Runnable, ActionListener {
  private Vector circles;
  private int width, height;
  private Image offScreenImage;
  private Graphics offScreenGraphics;
  private Button startButton, stopButton;
  private Thread animationThread = null;

  public void init() {
    setBackground(Color.white);
    width = getSize().width;
    height = getSize().height;
    offScreenImage = createImage(width, height);
    offScreenGraphics = offScreenImage.getGraphics();
    // Automatic in some systems, not in others.
    offScreenGraphics.setColor(Color.black);
    circles = new Vector();
```

```
    startButton = new Button("Start a circle");
    startButton.addActionListener(this);
    add(startButton);
    stopButton = new Button("Stop all circles");
    stopButton.addActionListener(this);
    add(stopButton);
  }

  /** When the "start" button is pressed, start the animation
   *  thread if it is not already started. Either way, add a
   *  circle to the Vector of circles that are being bounced.
   *  <P>
   *  When the "stop" button is pressed, stop the thread and
   *  clear the Vector of circles.
   */

  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == startButton) {
      if (circles.size() == 0) {
        animationThread = new Thread(this);
        animationThread.start();
      }
      int radius = 25;
      int x = radius + randomInt(width -- 2 * radius);
      int y = radius + randomInt(height -- 2 * radius);
      int deltaX = 1 + randomInt(10);
      int deltaY = 1 + randomInt(10);
      circles.addElement(new MovingCircle(x, y, radius, deltaX,
                                          deltaY));
      repaint();
    } else if (event.getSource() == stopButton) {
      if (animationThread != null) {
        animationThread = null;
        circles.removeAllElements();
      }
    }
  }

  /** Each time around the loop, move each circle based on its
   *  current position and deltaX/deltaY values. These values
   *  reverse when the circles reach the edge of the window.
   */
  public void run() {
    MovingCircle circle;
    Thread myThread = Thread.currentThread();
    // Really while animationThread not null.
    while(animationThread==myThread) {
      for(int j=0; j<circles.size(); j++) {
        circle = (MovingCircle)circles.elementAt(j);
        circle.move(width, height);
      }
      repaint();
      pause(100);
```

```
    }
  }

  /** Skip the usual screen--clearing step of update so that
   *  there is no flicker between each drawing step.
   */

  public void update(Graphics g) {
    paint(g);
  }

  /** Clear the off--screen pixmap, draw each circle onto it, then
   *  draw that pixmap onto the applet window.
   */

  public void paint(Graphics g) {
    offScreenGraphics.clearRect(0, 0, width, height);
    MovingCircle circle;
    for(int i=0; i<circles.size(); i++) {
      circle = (MovingCircle)circles.elementAt(i);
      circle.draw(offScreenGraphics);
    }
    g.drawImage(offScreenImage, 0, 0, this);
  }

  // Returns an int from 0 to max (inclusive), yielding max + 1
  // possible values.

  private int randomInt(int max) {
    double x = Math.floor((double)(max + 1) * Math.random());
    return((int)(Math.round(x)));
  }

  // Sleep for the specified amount of time.

  private void pause(int milliseconds) {
    try {
      Thread.sleep((long)milliseconds);
    } catch(InterruptedException ie) {}
  }
}
```

Pros and Cons

Drawing into an off-screen image and then drawing that image once to the screen is much faster than drawing directly to the screen. Double buffering allows fast, flicker-free updating even when no reliable way to erase the old graphics is available. However, double buffering is more complex than most of the previously discussed options and requires extra memory for the off-screen image.

Don't underestimate the beauty and strength of double buffering. Granted, additional programing is required to achieve double buffering in the AWT model. However, once you move to Swing components (covered in Chapter 14, Basic Swing), everything you've learned about double buffering becomes commonplace, because double buffering is built right into the Swing model. What a plus!

16.8 Animating Images

An exciting application for threads is animation of images. Many Web pages contain animated GIF files (GIF89A) that cycle through a sequence of images. Typically, these animated GIF files are created with graphical packages like Adobe PhotoShop and Quark, which permit selection of a sequence of equal-sized images to combine into a single GIF file. When loaded into a browser, each subimage in the GIF sequence is presented to the user in a time-sliced fashion. The Java programming language provides better control over animated GIF89A files because the sequence of images can be explicitly controlled, as can the timing interval between image updates. The basic concept for creating animated images is as follows:

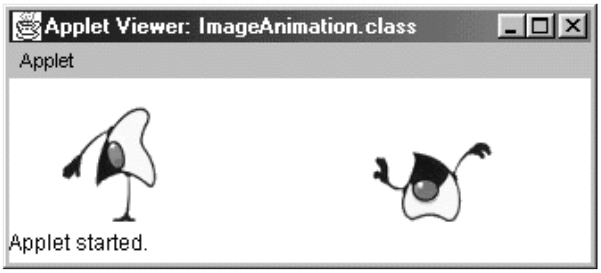
- Read the sequence of images into an `Image` array.
- Define an index variable to sequence through the `Image` array.
- Start a thread to continuously cycle through a sequence of index values, calling `repaint` and `sleep` after each change of the index value.
- In `paint` (AWT) or `paintComponent` (Swing), draw the indexed image to the `Graphics` or `Graphics2D` object.

The order in which the images are presented to the user is controlled by the index sequence, and the delay between image updates is controlled by the sleeping interval of the thread (as well as by the OS). See Section 9.12 (Drawing Images) for additional information on loading and drawing images.

An example of creating an animated image is shown in Listing 16.24. In this applet, each `Duke` object represents a unique thread cycling an index through an array of 15 images. One thread cycles through the index values in increasing order (`tumbleDirection` of `1`), while the other thread cycles through the index values in decreasing order (`tumbleDirection` of `-1`). For each thread to have knowledge of the `repaint` method in the applet, a reference to the parent `Applet` is passed in to the `Duke` constructor. Whenever an index value is changed in either thread, redrawing of the applet is requested by a call to `repaint`. The `paint` method queries each thread for the corresponding image index value and then draws the correct Duke™ image stored in the array to the `Graphics` object.

Additionally, the `Duke` class provides a public `setState` method so that the thread can be courteously started and stopped when the HTML file is loaded and removed from the browser. The result for this applet is shown in Figure 16-5. An excellent enhancement to this animation would be to preload the images before any drawing is performed. See Section 9.13 (Preloading Images) and Section 9.14 (Controlling Image Loading: Waiting for Images and Checking Status) for additional details.

Figure 16-5. The use of threads to perform animation of Duke. [Duke™ is a registered trademark of Sun Microsystems, Inc. Images used with permission. All rights reserved.]



Listing 16.24 `ImageAnimation.java`

```
import java.applet.Applet;
import java.awt.*;
```

```
public class ImageAnimation extends Applet {

/** Sequence through an array of 15 images to perform the
 *  animation. A separate Thread controls each tumbling Duke.
 *  The Applet's stop method calls a public service of the
 *  Duke class to terminate the thread. Override update to
 *  avoid flicker problems.
 */

  private static final int NUMDUKES  = 2;
  private Duke[] dukes;
  private int i;

  public void init() {
    dukes = new Duke[NUMDUKES];
    setBackground(Color.white);
  }

  /** Start each thread, specifing a direction to sequence
   *  through the array of images.
   */

  public void start() {
    int tumbleDirection;
    for (int i=0; i<NUMDUKES ; i++) {
      tumbleDirection = (i%2==0) ? 1 :--1;
      dukes[i] = new Duke(tumbleDirection, this);
      dukes[i].start();
    }
  }

  /** Skip the usual screen--clearing step of update so that
   *  there is no flicker between each drawing step.
   */

  public void update(Graphics g) {
    paint(g);
  }

  public void paint(Graphics g) {
    for (i=0 ; i<NUMDUKES ; i++) {
      if (dukes[i] != null) {
        g.drawImage(Duke.images[dukes[i].getIndex()],
                    200*i, 0, this);
      }
    }
  }

  /** When the Applet's stop method is called, use the public
   *  service, setState, of the Duke class to set a flag and
   *  terminate the run method of the thread.
   */
```

```
  public void stop() {
    for (int i=0; i<NUMDUKES ; i++) {
      if (dukes[i] != null) {
        dukes[i].setState(Duke.STOP);
      }
    }
  }
}
```

Listing 16.25 Duke.java

```
import java.applet.Applet;
import java.awt.*;

/** Duke is a Thread that has knowledge of the parent applet
 *  (highly coupled) and thus can call the parent's repaint
 *  method. Duke is mainly responsible for changing an index
 *  value into an image array.
 */

public class Duke extends Thread {
  public static final int STOP = 0;
  public static final int RUN  = 1;
  public static final int WAIT = 2;
  public static Image[] images;
  private static final int NUMIMAGES = 15;
  private static Object lock = new Object();
  private int state = RUN;
  private int tumbleDirection;
  private int index = 0;
  private Applet parent;

  public Duke(int tumbleDirection, Applet parent) {
    this.tumbleDirection = tumbleDirection;
    this.parent = parent;
    synchronized(lock) {
      if (images==null) {  // If not previously loaded.
        images = new Image[ NUMIMAGES ];
        for (int i=0; i<NUMIMAGES; i++) {
          images[i] = parent.getImage( parent.getCodeBase(),
                                       "images/T" + i + ".gif");
        }
      }
    }
  }

  /** Return current index into image array.  */

  public int getIndex() { return index; }

  /** Public method to permit setting a flag to stop or
   *  suspend the thread. State is monitored through
   *  corresponding checkState method.
```

```
   */

  public synchronized void setState(int state) {
    this.state = state;
    if (state==RUN) {
      notify();
    }
  }

  /** Returns the desired state (RUN, STOP, WAIT) of the
   *  thread. If the thread is to be suspended, then the
   *  thread method wait is continuously called until the
   *  state is changed through the public method setState.
   */

  private synchronized int checkState() {
    while (state==WAIT) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    return state;
  }

  /** The variable index (into image array) is incremented
   *  once each time through the while loop, calls repaint,
   *  and pauses for a moment. Each time through the loop the
   *  state (flag) of the thread is checked.
   */

  public void run() {
    while (checkState()!=STOP) {
      index += tumbleDirection;
      if (index < 0) {
        index = NUMIMAGES -- 1;
      }
      if (index >= NUMIMAGES) {
        index = 0;
      }

      parent.repaint();

      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
        break;   // Break while loop.
      }
    }
  }
}
```

Interestingly, Java does support GIF89A files. Simply load the GIF file into an `Image` object and then draw the image in `paint` as normally would be done. Behind the scenes, Java sets up a *separate*

Animator thread which continuously calls `repaint` and adjusts for the subimage embedded in the GIF89A file when `paint` is called. Ok, so you've placed an animated GIF (GIF89A) file in your applet and the user goes to a different HTML page. How do you gracefully stop the thread that is controlling the animation? What flag would you set and for which `run` method? Good question. In Internet Explorer, the Animator thread is automatically terminated when the user goes to a new Web page. Sadly though, in Netscape the browser must be closed to terminate the Animator thread.

16.9 Timers

Timers are useful for numerous tasks, including image animation, starting and stopping simulations, and timing out secure network connections. The previous section (Animating Images) illustrated the capability of animating images by creating a user-defined thread that periodically invokes a specified action and then goes to sleep. To simplify this common and useful task, a robust `Timer` class, `javax.swing.Timer`, was added to the Swing package. A `Timer` can run for a single cycle or be set to ring periodically.

Creating a Swing `Timer` is as simple as instantiating a `Timer` object, specifying the period in milliseconds that the `Timer` should fire an `ActionEvent` and specifying a listener that should receive the event. Once the timer is created, activate the timer by calling the `start` method, as in:

```
Timer timer = new Timer(period, listener);
timer.start();
```

By default, a `Timer` creates a thread that periodically fires an `ActionEvent` event every `period` milliseconds to *all* attached listeners (see `addActionListener`). Alternatively, the `Timer` can be defined to only fire one event, using `setRepeats(false)`, and then stop. However, unlike a normal `Thread` that once stopped, cannot be restarted, calling `restart` on a `Timer` begins the timing sequence over again.

When the `Timer` fires, an event object is sent to the event queue. Depending on how many events are already in the queue and how heavily the system is tasked, multiple `Timer` events can be queued before the first event is processed. By default, `Timer` events are coalesced. That is, if an earlier event is already in the queue, then the next trigger will not add a new event object into the queue. If *every* trigger event does indeed require processing, set coalescing to `false` with `setCoalese(false)`. By turning on logging, `setLogTimers(true)`, you send a print message to the standard output each time a `Timer` fires an `ActionEvent`, thus providing an easy means for determining when the timers are firing. The `setLogTimers` method is a `static` method; thus, *all* active timers in the program will generate a log message.

The application of a `Timer` to control image animation is shown in Listing 16.26 and Listing 16.27. Here, the use of timers *greatly* simplifies the animation, since *explicit* creation and control of `Thread` objects is no longer required. In this example, the applet creates two separate `TimedDuke` objects, each with an internal `Timer`. The triggering period for each timer is different. As more than one `TimedDuke` object is instantiated, the synchronization block in the constructor of `TimedDuke` (along with the `loaded` flag) prevents the potential race condition of loading the images twice. Once the `MediaTracker` acknowledges loading of the animation images into the array, each timer is started by the applet. When a timer fires, the `actionPerformed` method is eventually called on the appropriate `TimedDuke` object, which in turn increments an internal index into the image array and then calls `repaint` on the applet. Support methods permit the applet to start and stop the timers when the user transfers to a new HTML page and returns.

Listing 16.26 `TimedAnimation.java`

```
import java.awt.*;
import javax.swing.*;
```

```
/** An example of performing animation through Swing timers.
 *  Two timed Dukes are created with different timer periods.
 */

public class TimedAnimation extends JApplet {
  private static final int NUMDUKES = 2;
  private TimedDuke[] dukes;
  private int i, index;

  public void init() {
    dukes = new TimedDuke[NUMDUKES];
    setBackground(Color.white);
    dukes[0] = new TimedDuke( 1, 100, this);
    dukes[1] = new TimedDuke(-1, 500, this);

  }

  //  Start each Duke timer.
  public void start() {
    for (int i=0; i<NUMDUKES ; i++) {
      dukes[i].startTimer();
    }
  }

  public void paint(Graphics g) {
    for (i=0 ; i<NUMDUKES ; i++) {
      if (dukes[i] != null) {
        index = dukes[i].getIndex();
        g.drawImage(TimedDuke.images[index], 200*i, 0, this);
      }
    }
  }

  //  Stop each Duke timer.

  public void stop() {
    for (int i=0; i<NUMDUKES ; i++) {
      dukes[i].stopTimer();
    }
  }
}
```

Listing 16.27 `TimedDuke.java`

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/** Duke facilitates animation by creating an internal timer.
 *  When the timer fires, an actionPerformed event is
 *  triggered, which in turn calls repaint on the parent
 *  Applet.
```

```
 */

public class TimedDuke implements ActionListener {
  private static final int NUMIMAGES = 15;
  private static boolean loaded = false;
  private static Object lock = new Object();
  private int tumbleDirection;
  private int msec;
  private int index = 0;
  private Applet parent;
  private Timer timer;
  public static Image[] images = new Image[NUMIMAGES];

  public TimedDuke(int tumbleDirection, int msec,
                                        Applet parent) {
    this.tumbleDirection = tumbleDirection;
    this.msec = msec;
    this.parent = parent;

    synchronized (lock) {
      if (!loaded) {
         MediaTracker tracker = new MediaTracker(parent);
         for (int i=0; i<NUMIMAGES; i++) {
           images[i] = parent.getImage(parent.getCodeBase(),
                                       "images/T" + i + ".gif");
           tracker.addImage(images[i],0);
         }
         try {
           tracker.waitForAll();
         } catch (InterruptedException ie) {}
         if (!tracker.isErrorAny()) {
           loaded = true;
         }
      }
    }

    timer = new Timer(msec, this);
  }

  // Return current index into image array.

  public int getIndex() { return index; }

  // Receives timer firing event.  Increments the index into
  // image array and forces repainting of the new image.

  public void actionPerformed(ActionEvent event) {
    index += tumbleDirection;
    if (index < 0){
      index = NUMIMAGES - 1;
    }
    if (index >= NUMIMAGES) {
      index = 0;
```

```
    }
    parent.repaint();
  }

  // Public service to start the timer.
  public void startTimer() {
    timer.start();
  }

  // Public service to stop the timer.
  public void stopTimer() {
    timer.stop();
  }
}
```

Constructor

The `Timer` class has only one constructor:

public Timer(int period, ActionListener listener)

The `Timer` class has a single constructor that sets the timing period and listener to receive the timing event. By default, the timer will fire (ring) an `ActionEvent` event every `period` milliseconds. The fired event is delivered to all registered listeners.

Other Timer Methods

public void addActionListener(ActionListener listener)

This method registers an `ActionListener` with the `Timer`. Each queued `ActionEvent` is delivered to the `actionPerformed` method of each registered listener.

public boolean isRunning()

This method returns `true` if the timer is actively executing; otherwise, returns `false`.

public void removeActionListener(ActionListener listener)

This method removes the `ActionListener` from the list of listeners registered with the `Timer`.

public void restart()

This method cancels any undelivered events and immediately starts the timer again from an initial starting point.

public void setCoalesce(boolean flag)

The `setCoalesce` method turns on (`true`) or off (`false`) `ActionEvent` coalescing. By default, if an `ActionEvent` from the timer is already waiting in the event queue for processing, the timer will not create a new `ActionEvent` at the next firing interval. Turning coalescing off forces queueing and processing of all the timer events.

public void setDelay(int period)

This method permits changing the timing event `period` to a new value. The new period is applied immediately.

public void setInitialDelay(int delay)

The `setInitialDelay` method applies an initial offset of `delay` milliseconds until the first firing (ringing) of the timer. Setting the initial delay has no effect once the timer is started.

public static void setLogTimers(boolean flag)

This `static` method enables or disables logging of event triggers for *all* timers. The event is recorded to `System.out` in the form: `Timer ringing: TimedDuke@3f345a`.

public void setRepeats(boolen repeat)

This method sets the timer to ring once (`false`) or to ring periodically (`true`). Periodic ringing is the default.

public void start()

The `start` method begins the initial timing sequence of the `Timer`. If the timer had been halted by `stop`, the timing begins from the point at which the timer was stopped; the elapsed time is not set to 0.

public void stop()

The `stop` method halts the timing sequence; no further `ActionEvent`s are generated from the `Timer`.

16.10 Summary

Threads can be used for a variety of applications. In some cases, using threads makes software design simpler by letting you separate various pieces of work into independent chunks rather than coordinating them in a central routine. In other cases, threads can improve efficiency by letting you continue processing while a routine is waiting for user input or a network connection.

However, threaded programs tend to be more difficult to understand and debug, and improper synchronization can lead to inconsistent results. So use threads with some caution.

A particular difficulty arises when threads are used for animation or when graphics change dynamically based on user interaction. A variety of potential solutions are possible, but one of the most generally applicable ones is double buffering, where graphics operations are performed in an off-screen pixmap which is then drawn to the screen.

Concurrent processing can be particularly beneficial in network programming. The next chapter discusses a variety of issues concerning that topic, including how servers should be made multithreaded.

 CONTENTS