# DailyAdviceServer code

This program makes a ServerSocket and waits for client requests. When it gets a client request (i.e. client said new Socket() for this application), the server makes a new Socket connection to that client. The server makes a PrintWriter (using the Socket's output stream) and sends a message to the client.

```
import java.io.*;
import java.net.*;

public class DailyAdviceServer {

   String[] adviceList = {"Take smaller bites", "Go for the tight jeans. No they do NOT
make you look fat.", "One word: inappropriate", "Just for today, be honest. Tell your
boss what you *really* think", "You might want to rethink that haircut."};

   public void go() {

      try {
         ServerSocket serverSock = new ServerSocket(4242);

         while(true) {

            Socket sock = serverSock.accept();

            PrintWriter writer = new PrintWriter(sock.getOutputStream());
            String advice = getAdvice();
            writer.println(advice);
            writer.close();
            System.out.println(advice);
         }

      } catch(IOException ex) {
         ex.printStackTrace();
      }
   } // close go

   private String getAdvice() {
      int random = (int) (Math.random() * adviceList.length);
      return adviceList[random];
   }

   public static void main(String[] args) {
      DailyAdviceServer server = new DailyAdviceServer();
      server.go();
   }
}
```

remember the imports

daily advice comes from this array

(remember, these Strings were word-wrapped by the code editor. Never hit return in the middle of a String!)

ServerSocket makes this server application 'listen' for client requests on port 4242 on the machine this code is running on.

The server goes into a permanent loop, waiting for (and servicing) client requests

the accept method blocks (just sits there) until a request comes in, and then the method returns a Socket (on some anonymous port) for communicating with the client

now we use the Socket connection to the client to make a PrintWriter and send it (println()) a String advice message. Then we close the Socket because we're done with this client.

## Brain Barbell

***How does the server know how to communicate with the client?***

The client knows the IP address and port number of the server, but how is the server able to make a Socket connection with the client (and make input and output streams)?

Think about how / when / where the server gets knowledge about the client.

## there are no Dumb Questions

**Q: The advice server code on the opposite page has a VERY serious limitation—it looks like it can handle only one client at a time!**

A: Yes, that's right. It can't accept a request from a client until it has finished with the current client and started the next iteration of the infinite loop (where it sits at the accept() call until a request comes in, at which time it makes a Socket with the new client and starts the process over again).

**Q: Let me rephrase the problem: how can you make a server that can handle multiple clients concurrently??? This would *never* work for a chat server, for instance.**

A: Ah, that's simple, really. Use separate threads, and give each new client Socket to a new thread. We're just about to learn how to do that!

## BULLET POINTS

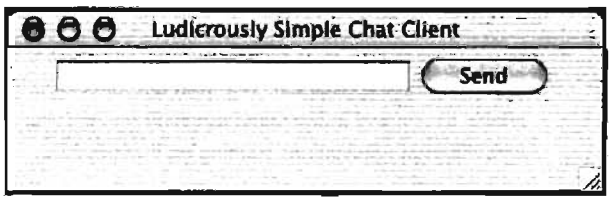
- Client and server applications communicate over a Socket connection.
- A Socket represents a connection between two applications which may (or may not) be running on two different physical machines.
- A client must know the IP address (or domain name) and TCP port number of the server application.
- A TCP port is a 16-bit unsigned number assigned to a specific server application. TCP port numbers allow different clients to connect to the same machine but communicate with different applications running on that machine.
- The port numbers from 0 through 1023 are reserved for 'well-known services' including HTTP, FTP, SMTP, etc.
- A client connects to a server by making a Server socket
  `Socket s = new Socket("127.0.0.1", 4200);`
- Once connected, a client can get input and output streams from the socket. These are low-level 'connection' streams.
  `sock.getInputStream();`
- To read text data from the server, create a BufferedReader, chained to an InputStreamReader, which is chained to the input stream from the Socket.
- InputStreamReader is a 'bridge' stream that takes in bytes and converts them to text (character) data. It's used primarily to act as the middle chain between the high-level BufferedReader and the low-level Socket input stream.
- To write text data to the server, create a PrintWriter chained directly to the Socket's output stream. Call the print() or println() methods to send Strings to the server.
- Servers use a ServerSocket that waits for client requests on a particular port number.
- When a ServerSocket gets a request, it 'accepts' the request by making a Socket connection with the client.

# Writing a Chat Client

We'll write the Chat client application in two stages. First we'll make a send-only version that sends messages to the server but doesn't get to read any of the messages from other participants (an exciting and mysterious twist to the whole chat room concept).

Then we'll go for the full chat monty and make one that both sends *and* receives chat messages.

## Version One: send-only




Type a message, then press 'Send' to send it to the server. We won't get any messages FROM the server in this version, so there's no scrolling text area.

## Code outline

```
public class SimpleChatClientA {

   JTextField outgoing;
   PrintWriter writer;
   Socket sock;

   public void go() {
        // make gui and register a listener with the send button
        // call the setUpNetworking() method
   }

   private void setUpNetworking() {
        // make a Socket, then make a PrintWriter
        // assign the PrintWriter to writer instance variable
   }

   public class SendButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
         // get the text from the text field and
         // send it to the server using the writer (a PrintWriter)
      }
   }  // close SendButtonListener inner class

} // close outer class
```

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

imports for the streams (java.io), Socket (java.net) and the GUI stuff

```
public class SimpleChatClientA {

    JTextField outgoing;
    PrintWriter writer;
    Socket sock;

    public void go() {
        JFrame frame = new JFrame("Ludicrously Simple Chat Client");
        JPanel mainPanel = new JPanel();
        outgoing = new JTextField(20);
        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(new SendButtonListener());
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);
        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        setUpNetworking();
        frame.setSize(400,500);
        frame.setVisible(true);
    } // close go
```

build the GUI, nothing new here, and nothing related to networking or I/O.

```
    private void setUpNetworking() {
        try {
            sock = new Socket("127.0.0.1", 5000);
            writer = new PrintWriter(sock.getOutputStream());
            System.out.println("networking established");
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    } // close setUpNetworking
```

we're using localhost so you can test the client and server on one machine

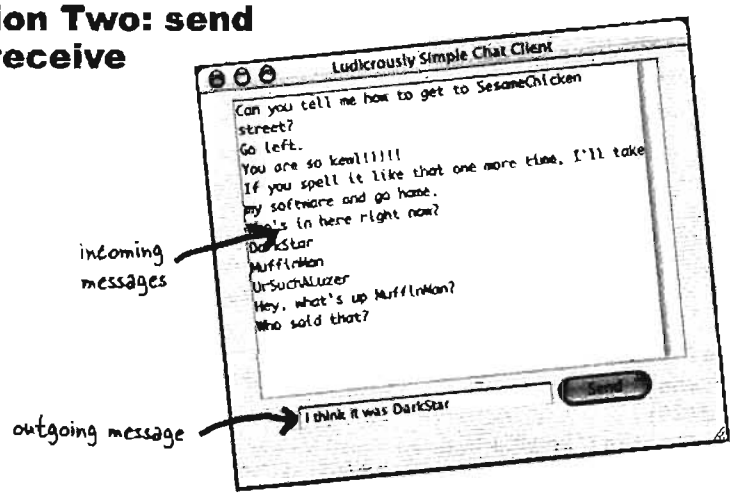
This is where we make the Socket and the PrintWriter (it's called from the go() method right before displaying the app GUI)

```
  public class SendButtonListener implements ActionListener {
     public void actionPerformed(ActionEvent ev) {
         try {
            writer.println(outgoing.getText());
            writer.flush();

         } catch(Exception ex) {
            ex.printStackTrace();
         }
         outgoing.setText("");
         outgoing.requestFocus();
     }
  } // close SendButtonListener inner class

  public static void main(String[] args) {
     new SimpleChatClientA().go();
  }
} // close outer class
```

Now we actually do the writing. Remember, the writer is chained to the input stream from the Socket, so whenever we do a println(), it goes over the network to the server!

**If you want to try this now, type in the Ready-bake chat server code listed at the end of this chapter. First, start the server in one terminal. Next, use another terminal to start this client.**

## Version Two: send and receive




The Server sends a message to all client participants, as soon as the message is received by the server. When a client sends a message, it doesn't appear in the incoming message display area until the server sends it to everyone.

### Big Question: *HOW* do you get messages from the server?

Should be easy; when you set up the networking make an input stream as well (probably a BufferedReader). Then read messages using readLine().

### Bigger Question: *WHEN* do you get messages from the server?

Think about that. What are the options?

#### Option One: Poll the server every 20 seconds

**Pros:** Well, it's do-able

**Cons:** How does the server know what you've seen and what you haven't? The server would have to store the messages, rather than just doing a distribute-and-forget each time it gets one. And why 20 seconds? A delay like this affects usability, but as you reduce the delay, you risk hitting your server needlessly. Inefficient.

#### Option Two: Read something in from the server each time the user sends a message.

**Pros:** Do-able, very easy

Cons: Stupid. Why choose such an arbitrary time to check for messages? What if a user is a lurker and doesn't send anything?

#### Option Three: Read messages as soon as they're sent from the server

**Pros:** Most efficient, best usability

Cons: How do you do you do two things at the same time? Where would you put this code? You'd need a loop somewhere that was always waiting to read from the server. But where would that go? Once you launch the GUI, nothing happens until an event is fired by a GUI component.

In Java you really CAN walk and chew gum at the same time.

## You know by now that we're going with option three.

We want something to run continuously, checking for messages from the server, but *without interrupting the user's ability to interact with the GUI!* So while the user is happily typing new messages or scrolling through the incoming messages, we want something *behind the scenes* to keep reading in new input from the server.

That means we finally need a new thread. A new, separate stack

We want everything we did in the Send-Only version (version one) to work the same way, while a new *process* runs along side that reads information from the server and displays it in the incoming text area.

Well, not quite. Unless you have multiple processors on your computer, each new Java thread is not actually a separate process running on the OS. But it almost *feels* as though it is.

## Multithreading in Java

Java has multiple threading built right into the fabric of the language. And it's a snap to make a new thread of execution:

```
Thread t = new Thread();
t.start();
```

That's it. By creating a new Thread *object*, you've launched a separate *thread of execution*, with its very own call stack.

***Except for one problem.***

That thread doesn't actually *do* anything, so the thread "dies" virtually the instant it's born. When a thread dies, its new stack disappears again. End of story.

So we're missing one key component—the thread's *job*. In other words, we need the code that you want to have run by a separate thread.

Multiple threading in Java means we have to look at both the *thread* and the *job* that's *run* by the thread. And we'll also have to look at the Thread *class* in the java.lang package. (Remember, java.lang is the package you get imported for free, implicitly, and it's where the classes most fundamental to the language live, including String and System.)

# Java has multiple threads but only one Thread class

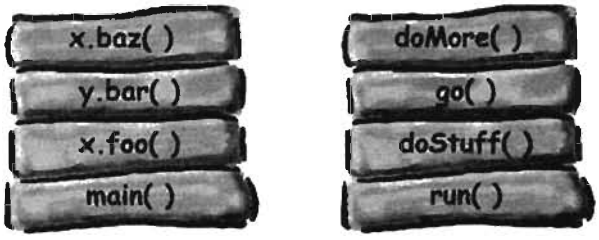
We can talk about *thread* with a lower-case 't' and **Thread** with a capital 'T'. When you see *thread*, we're talking about a separate thread of execution. In other words, a separate call stack. When you see **Thread**, think of the Java naming convention. What, in Java, starts with a capital letter? Classes and interfaces. In this case, **Thread** is a class in the java.lang package. A **Thread** object represents a *thread of execution*; you'll create an instance of class **Thread** each time you want to start up a new *thread* of execution.

A thread is a separate 'thread of execution'. In other words, a separate call stack.

A Thread is a Java class that represents a thread.

To make a thread, make a Thread.

**thread**

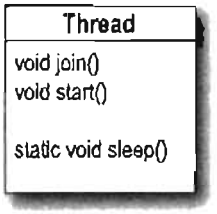



**main thread**

**another thread started by the code**

A thread (lower-case 't') is a separate thread of execution. That means a separate call stack. Every Java application starts up a main thread—the thread that puts the main() method on the bottom of the stack. The JVM is responsible for starting the main thread (and other threads, as it chooses, including the garbage collection thread). As a programmer, you can write code to start other threads of your own.

**Thread**




**java.lang.Thread class**

Thread (capital 'T') is a class that represents a thread of execution. It has methods for starting a thread, joining one thread with another, and putting a thread to sleep. (It has more methods; these are just the crucial ones we need to use now).

# What does it mean to have more than one call stack?

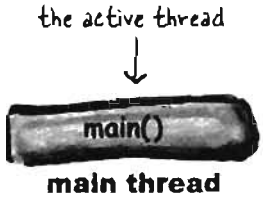
With more than one call stack, you get the *appearance* of having multiple things happen at the same time. In reality, only a true multiprocessor system can actually do more than one thing at a time, but with Java threads, it can *appear* that you're doing several things simultaneously. In other words, execution can move back and forth between stacks so rapidly that you feel as though all stacks are executing at the same time. Remember, Java is just a process running on your underlying OS. So first, Java *itself* has to be 'the currently executing process' on the OS. But once Java gets its turn to execute, exactly *what* does the JVM *run*? Which bytecodes execute? Whatever is on the top of the currently-running stack! And in 100 milliseconds, the currently executing code might switch to a *different* method on a *different* stack.

One of the things a thread must do is keep track of which statement (of which method) is currently executing on the thread's stack.

It might look something like this:

**1** **The JVM calls the main() method.**

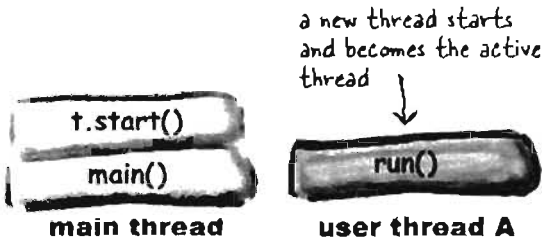
```
public static void main(String[] args) {
...
}
```



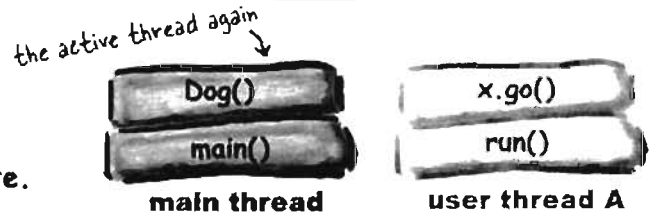

**2** **main() starts a new thread. The main thread is temporarily frozen while the new thread starts running.**

```
Runnable r = new MyThreadJob();
Thread t = new Thread(r);
t.start();
Dog d = new Dog();
```

you'll learn what this means in just a moment...




**3** **The JVM switches between the new thread (user thread A) and the original main thread, until both threads complete.**



## How to launch a new thread:

### 1 Make a Runnable object (the thread's job)

```
Runnable threadJob = new MyRunnable();
```

Runnable is an interface you'll learn about on the next page. You'll write a class that implements the Runnable interface, and that class is where you'll define the work that a thread will perform. In other words, the method that will be run from the thread's new call stack.


### 2 Make a Thread object (the worker) and give it a Runnable (the job)

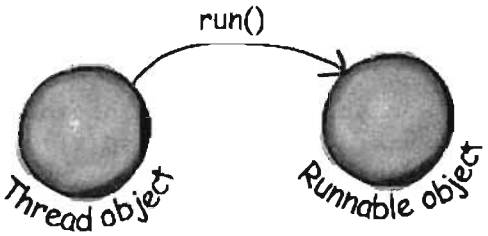
```
Thread myThread = new Thread(threadJob);
```

Pass the new Runnable object to the Thread constructor. This tells the new Thread object which method to put on the bottom of the new stack—the Runnable's run() method.


### 3 Start the Thread

```
myThread.start();
```

Nothing happens until you call the Thread's start() method. That's when you go from having just a Thread instance to having a new thread of execution. When the new thread starts up, it takes the Runnable object's run() method and puts it on the bottom of the new thread's stack.



# Every Thread needs a job to do. A method to put on the new thread stack.


Runnable is to a Thread what a job is to a worker. A Runnable is the job a thread is supposed to run.

A Runnable holds the method that goes on the bottom of the new thread's stack: run().

A Thread object needs a job. A job the thread will run when the thread is started. That job is actually the first method that goes on the new thread's stack, and it must always be a method that looks like this:

```
public void run() {
  // code that will be run by the new thread
}
```

The Runnable interface defines only one method, public void run(). (Remember, it's an interface so the method is public regardless of whether you type it in that way.)

How does the thread know which method to put at the bottom of the stack? Because Runnable defines a contract. Because Runnable is an interface. A thread's job can be defined in any class that implements the Runnable interface. The thread cares only that you pass the Thread constructor an object of a class that implements Runnable.

When you pass a Runnable to a Thread constructor, you're really just giving the Thread a way to get to a run() method. You're giving the Thread its job to do.

# To make a job for your thread, implement the Runnable interface

Runnable is in the java.lang package, so you don't need to import it

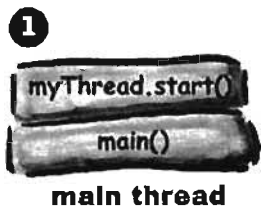
```
public class MyRunnable implements Runnable {

    public void run() {
        go();
    }

    public void go() {
        doMore();
    }

    public void doMore() {
        System.out.println("top o' the stack");
    }
}
```

Runnable has only one method to implement: public void run() (with no arguments). This is where you put the JOB the thread is supposed to run. This is the method that goes at the bottom of the new stack.

```
class ThreadTester {

    public static void main (String[] args) {

        Runnable threadJob = new MyRunnable();
        Thread myThread = new Thread(threadJob);

        myThread.start();

        System.out.println("back in main");
    }
}
```

Pass the new Runnable instance into the new Thread constructor. This tells the thread what method to put on the bottom of the new stack. In other words, the first method that the new thread will run.

You won't get a new thread of execution until you call start() on the Thread instance. A thread is not really a thread until you start it. Before that, it's just a Thread instance, like any other object, but it won't have any real 'threadness'.



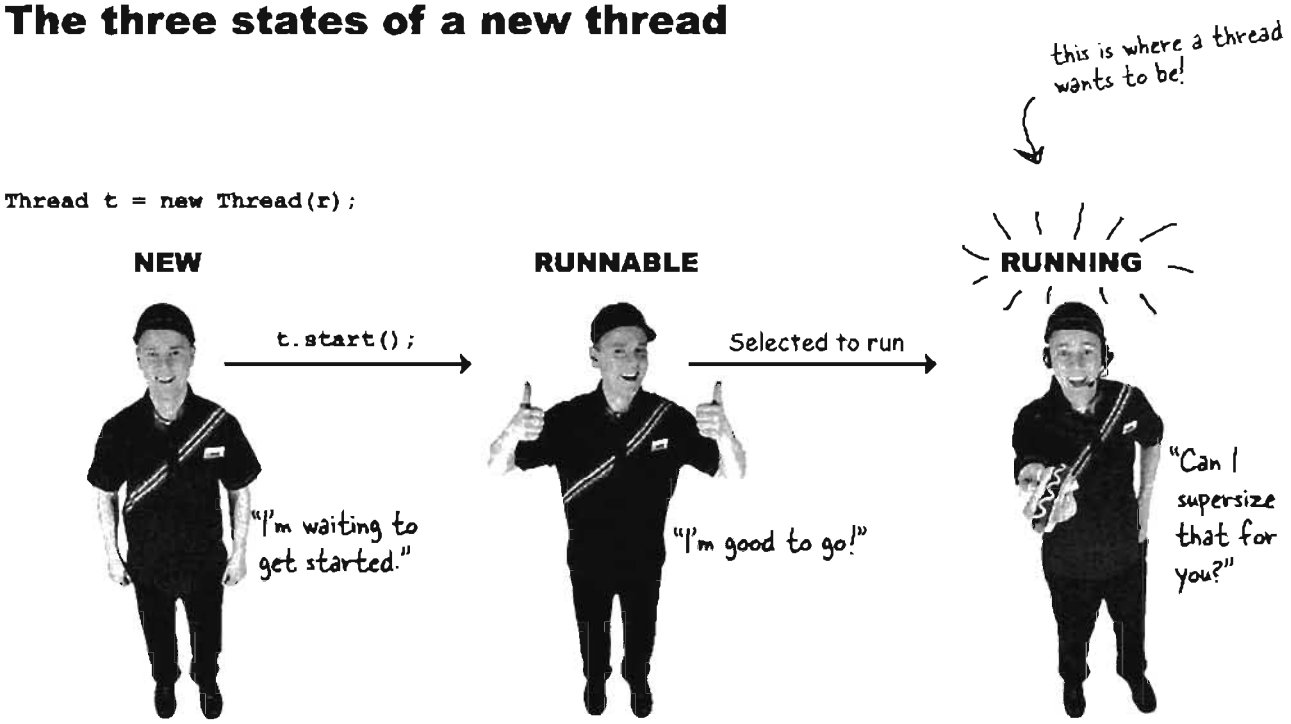
main thread

new thread

## Brain Barbell

What do you think the output will be if you run the ThreadTester class? (we'll find out in a few pages)

# The three states of a new thread




`Thread t = new Thread(r);`

A Thread instance has been created but not started. In other words, there is a Thread *object*, but no *thread of execution*.

`t.start();`

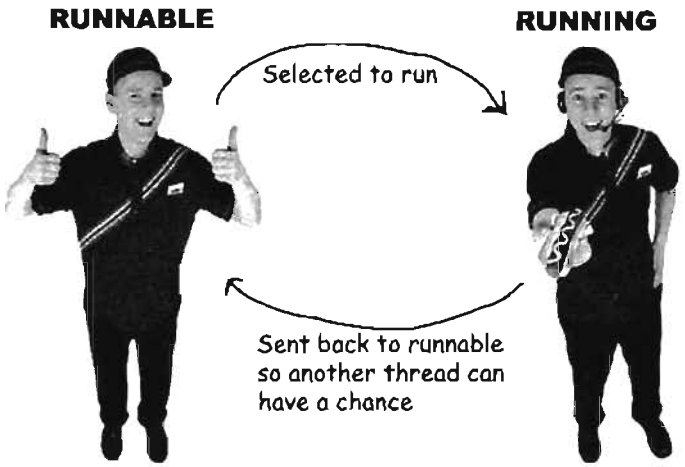
When you start the thread, it moves into the runnable state. This means the thread is ready to run and just waiting for its Big Chance to be selected for execution. At this point, there is a new call stack for this thread.

This is the state all threads lust after! To be The Chosen One. The Currently Running Thread. Only the JVM thread scheduler can make that decision. You can sometimes *influence* that decision, but you cannot force a thread to move from runnable to running. In the running state, a thread (and ONLY this thread) has an active call stack, and the method on the top of the stack is executing.

**But there's more. Once the thread becomes runnable, it can move back and forth between runnable, running, and an additional state: *temporarily not runnable* (also known as 'blocked').**

## Typical runnable/running loop

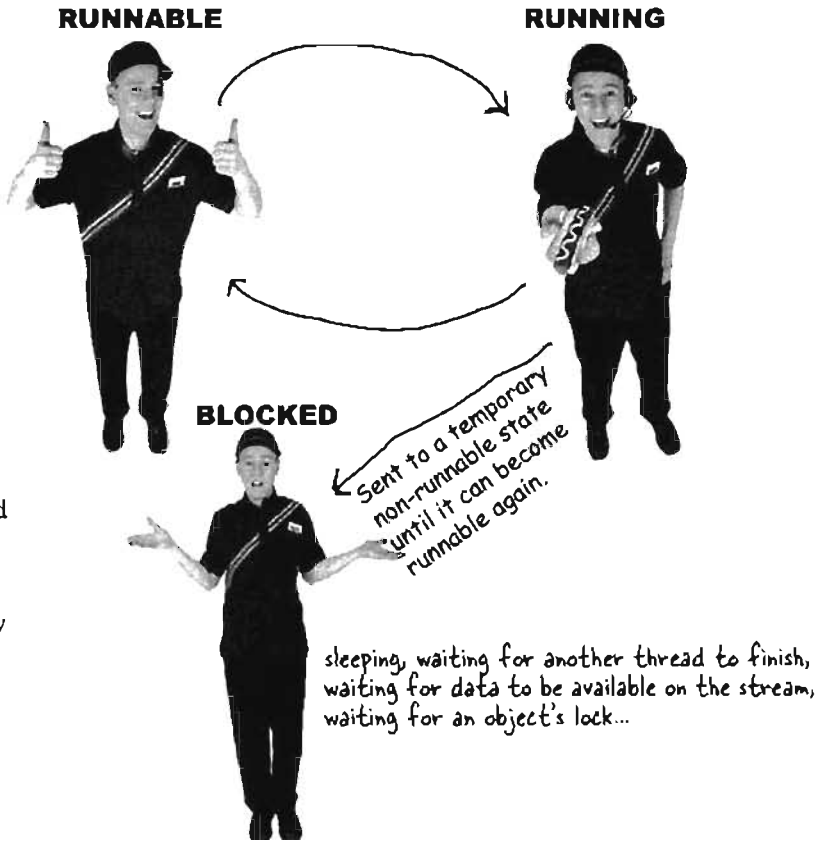
Typically, a thread moves back and forth between runnable and running, as the JVM thread scheduler selects a thread to run and then kicks it back out so another thread gets a chance.




## A thread can be made temporarily not-runnable

The thread scheduler can move a running thread into a blocked state, for a variety of reasons. For example, the thread might be executing code to read from a Socket input stream, but there isn't any data to read. The scheduler will move the thread out of the running state until something becomes available. Or the executing code might have told the thread to put itself to sleep (sleep()). Or the thread might be waiting because it tried to call a method on an object, and that object was 'locked'. In that case, the thread can't continue until the object's lock is freed by the thread that has it.

All of those conditions (and more) cause a thread to become temporarily not-runnable.



# The Thread Scheduler

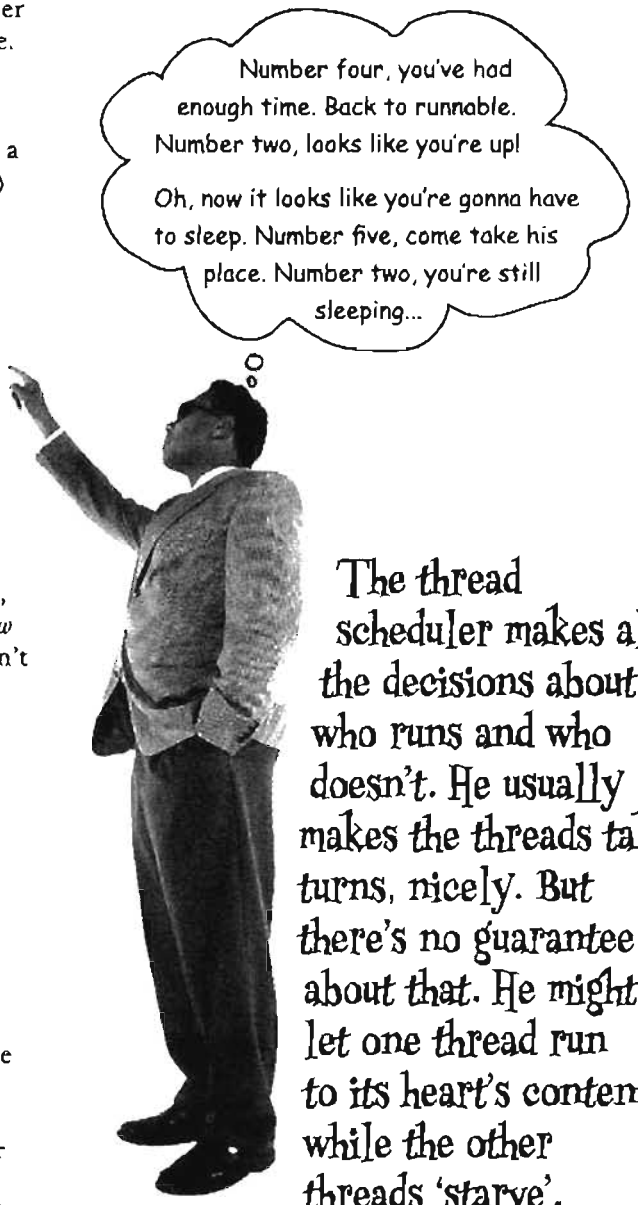
The thread scheduler makes all the decisions about who moves from runnable to running, and about when (and under what circumstances) a thread leaves the running state. The scheduler decides who runs, and for how long, and where the threads go when the scheduler decides to kick them out of the currently-running state.

You can't control the scheduler. There is no API for calling methods on the scheduler. Most importantly, there are no guarantees about scheduling! (There are a few *almost*-guarantees, but even those are a little fuzzy.)

The bottom line is this: ***do not base your program's correctness on the scheduler working in a particular way!*** The scheduler implementations are different for different JVM's, and even running the same program on the same machine can give you different results. One of the worst mistakes new Java programmers make is to test their multi-threaded program on a single machine, and assume the thread scheduler will always work that way, regardless of where the program runs.

So what does this mean for write-once-run-anywhere? It means that to write platform-independent Java code, your multi-threaded program must work no matter *how* the thread scheduler behaves. That means that you can't be dependent on, for example, the scheduler making sure all the threads take nice, perfectly fair and equal turns at the running state. Although highly unlikely today, your program might end up running on a JVM with a scheduler that says, "OK thread five, you're up, and as far as I'm concerned, you can stay here until you're done, when your run() method completes."

The secret to almost everything is *sleep*. That's right, *sleep*. Putting a thread to sleep, even for a few milliseconds, forces the currently-running thread to leave the running state, thus giving another thread a chance to run. The thread's sleep() method does come with *one* guarantee: a sleeping thread will *not* become the currently-running thread before the the length of its sleep time has expired. For example, if you tell your thread to sleep for two seconds (2,000 milliseconds), that thread can never become the running thread again until sometime *after* the two seconds have passed.




The thread scheduler makes all the decisions about who runs and who doesn't. He usually makes the threads take turns, nicely. But there's no guarantee about that. He might let one thread run to its heart's content while the other threads 'starve'.

# An example of how unpredictable the scheduler can be...

## Running this code on one machine:

```
public class MyRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {
      doMore();
   }

   public void doMore() {
      System.out.println("top o' the stack");
   }
}


class ThreadTestDrive {

   public static void main (String[] args) {

      Runnable threadJob = new MyRunnable();
      Thread myThread = new Thread(threadJob);

      myThread.start();

      System.out.println("back in main");
   }
}
```

## Produced this output:

```
File Edit Window Help PickMe
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
back in main
top o' the stack
```

Notice how the order changes randomly. Sometimes the new thread finishes first, and sometimes the main thread finishes first

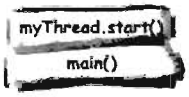
# How did we end up with different results?

## Sometimes it runs like this:

main() starts the new thread


main thread

The scheduler sends the main thread out of running and back to runnable, so that the new thread can run.



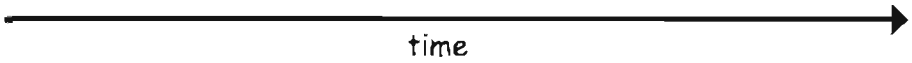

main thread

The scheduler lets the new thread run to completion, printing out "top o' the stack"


new thread

The new thread goes away, because its run() completed. The main thread once again becomes the running thread, and prints "back in main"


main thread

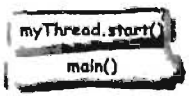



## And sometimes it runs like this:

main() starts the new thread


main thread

The scheduler sends the main thread out of running and back to runnable, so that the new thread can run.




main thread

The scheduler lets the new thread run for a little while, not long enough for the run() method to complete.


new thread

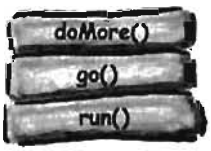
The scheduler sends the new thread back to runnable.


new thread

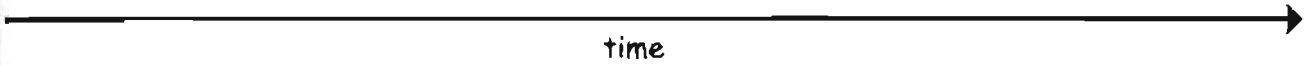
The scheduler selects the main thread to be the running thread again. Main prints out "back in main"


main thread

The new thread returns to the running state and prints out "top o' the stack".




new thread



## there are no Dumb Questions

**Q:** **I've seen examples that don't use a separate Runnable implementation, but instead just make a subclass of Thread and override the Thread's run() method. That way, you call the Thread's no-arg constructor when you make the new thread;**

**Thread t = new Thread(); // no Runnable**

**A:** Yes, that *is* another way of making your own thread, but think about it from an OO perspective. What's the purpose of subclassing? Remember that we're talking about two different things here—the *Thread* and the thread's *job*. From an OO view, those two are very separate activities, and belong in separate classes. The only time you want to subclass/extend the Thread class, is if you are making a new and more specific type of Thread. In other words, if you think of the Thread as the worker, don't extend the Thread class unless you need more specific *worker* behaviors. But if all you need is a new *job* to be run by a Thread/worker, then implement Runnable in a separate, *job*-specific (not *worker*-specific) class.

This is a design issue and not a performance or language issue. It's perfectly legal to subclass Thread and override the run() method, but it's rarely a good idea.

**Q:** **Can you reuse a Thread object? Can you give it a new job to do and then restart it by calling start() again?**

**A:** No. Once a thread's run() method has completed, the thread can never be restarted. In fact, at that point the thread moves into a state we haven't talked about—***dead***. In the dead state, the thread has finished its run() method and can never be restarted. The Thread object might still be on the heap, as a living object that you can call other methods on (if appropriate), but the Thread object has permanently lost its 'threadness'. In other words, there is no longer a separate call stack, and the Thread object is no longer a *thread*. It's just an object, at that point, like all other objects.

But, there are design patterns for making a pool of threads that you can keep using to perform different jobs. But you don't do it by restarting() a dead thread.

## BULLET POINTS

- A thread with a lower-case 't' is a separate thread of execution in Java.
- Every thread in Java has its own call stack.
- A Thread with a capital 'T' is the java.lang.Thread class. A Thread object represents a thread of execution.
- A Thread needs a job to do. A Thread's job is an instance of something that implements the Runnable interface.
- The Runnable interface has just a single method, run(). This is the method that goes on the bottom of the new call stack. In other words, it is the first method to run in the new thread.
- To launch a new thread, you need a Runnable to pass to the Thread's constructor.
- A thread is in the NEW state when you have instantiated a Thread object but have not yet called start().
- When you start a thread (by calling the Thread object's start() method), a new stack is created, with the Runnable's run() method on the bottom of the stack. The thread is now in the RUNNABLE state, waiting to be chosen to run.
- A thread is said to be RUNNING when the JVM's thread scheduler has selected it to be the currently-running thread. On a single-processor machine, there can be only one currently-running thread.
- Sometimes a thread can be moved from the RUNNING state to a BLOCKED (temporarily non-runnable) state. A thread might be blocked because it's waiting for data from a stream, or because it has gone to sleep, or because it is waiting for an object's lock.
- Thread scheduling is not guaranteed to work in any particular way, so you cannot be certain that threads will take turns nicely. You can help influence turn-taking by putting your threads to sleep periodically.

# Putting a thread to sleep

One of the best ways to help your threads take turns is to put them to sleep periodically. All you need to do is call the static sleep() method, passing it the sleep duration, in milliseconds.

For example:

```
Thread.sleep(2000);
```

will knock a thread out of the running state, and keep it out of the runnable state for two seconds. The thread *can't* become the running thread again until after at least two seconds have passed.

A bit unfortunately, the sleep method throws an InterruptedException, a checked exception, so all calls to sleep must be wrapped in a try/catch (or declared). So a sleep call really looks like this:

```
try {
  Thread.sleep(2000);
} catch(InterruptedException ex) {
    ex.printStackTrace();
}
```

Your thread will probably *never* be interrupted from sleep; the exception is in the API to support a thread communication mechanism that almost nobody uses in the Real World. But, you still have to obey the handle or declare law, so you need to get used to wrapping your sleep() calls in a try/catch.

Now you know that your thread won't wake up *before* the specified duration, but is it possible that it will wake up some time *after* the 'timer' has expired? Yes and no. It doesn't matter, really, because when the thread wakes up, ***it always goes back to the runnable state!*** The thread won't automatically wake up at the designated time and become the currently-running thread. When a thread wakes up, the thread is once again at the mercy of the thread scheduler. Now, for applications that don't require perfect timing, and that have only a few threads, it might appear as though the thread wakes up and resumes running right on schedule (say, after the 2000 milliseconds). But don't bet your program on it.

Put your thread to sleep if you want to be sure that other threads get a chance to run.

When the thread wakes up, it always goes back to the runnable state and waits for the thread scheduler to choose it to run again.

# Using sleep to make our program more predictable.

Remember our earlier example that kept giving us different results each time we ran it? Look back and study the code and the sample output. Sometimes main had to wait until the new thread finished (and printed "top o' the stack"), while other times the new thread would be sent back to runnable before it was finished, allowing the main thread to come back in and print out "back in main". How can we fix that? Stop for a moment and answer this question: "Where can you put a sleep() call, to make sure that "back in main" always prints before "top o' the stack"?

We'll wait while you work out an answer (there's more than one answer that would work).

Figure it out?

This is what we want—a consistent order of print statements:

```
File Edit Window Help SnoozeButton
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
back in main
top o' the stack
```

```
public class MyRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {

      try {
        Thread.sleep(2000);
      } catch(InterruptedException ex) {
        ex.printStackTrace();
      }

      doMore();
   }

   public void doMore() {
      System.out.println("top o' the stack");
   }
}

class ThreadTestDrive {
   public static void main (String[] args) {
      Runnable theJob = new MyRunnable();
      Thread t = new Thread(theJob);
      t.start();
      System.out.println("back in main");
   }
}
```

Calling sleep here will force the new thread to leave the currently-running state!

The main thread will become the currently-running thread again, and print out "back in main". Then there will be a pause (for about two seconds) before we get to this line, which calls doMore() and prints out "top o' the stack"

# Making and starting *two* threads

Threads have names. You can give your threads a name of your choosing, or you can accept their default names. But the cool thing about names is that you can use them to tell which thread is running. The following example starts two threads. Each thread has the same job: run in a loop, printing the currently-running thread's name with each iteration.

```
public class RunThreads implements Runnable {

    public static void main(String[] args) {
        RunThreads runner = new RunThreads();
        Thread alpha = new Thread(runner);
        Thread beta = new Thread(runner);
        alpha.setName("Alpha thread");
        beta.setName("Beta thread");
        alpha.start();
        beta.start();
    }

    public void run() {
        for (int i = 0; i < 25; i++) {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " is running");
        }
    }
}
```

Make one Runnable instance.

Make two threads, with the same Runnable (the same job--we'll talk more about the "two threads and one Runnable" in a few pages).

Name the threads.

Start the threads.

Each thread will run through this loop, printing its name each time.

# What will happen?

Will the threads take turns? Will you see the thread names alternating? How often will they switch? With each iteration? After five iterations?

You already know the answer: *we don't know!* It's up to the scheduler. And on your OS, with your particular JVM, on your CPU, you might get very different results.

Running under OS X 10.2 (Jaguar), with five or fewer iterations, the Alpha thread runs to completion, then the Beta thread runs to completion. Very consistent. Not guaranteed, but very consistent.

But when you up the loop to 25 or more iterations, things start to wobble. The Alpha thread might not get to complete all 25 iterations before the scheduler sends it back to runnable to let the Beta thread have a chance.

Part of the output when the loop iterates 25 times.

```
File Edit Window Help Centauri
Alpha thread is running
Alpha thread is running
Alpha thread is running
Beta thread is running
Alpha thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Beta thread is running
Alpha thread is running
```


## Um, yes. There IS a dark side. Threads can lead to concurrency 'issues'.

Concurrency issues lead to race conditions. Race conditions lead to data corruption. Data corruption leads to fear... you know the rest.

It all comes down to one potentially deadly scenario: two or more threads have access to a single object's *data*. In other words, methods executing on two different stacks are both calling, say, getters or setters on a single object on the heap.

It's a whole 'left-hand-doesn't-know-what-the-right-hand-is-doing' thing. Two threads, without a care in the world, humming along executing their methods, each thread thinking that he is the One True Thread. The only one that matters. After all, when a thread is not running, and in runnable (or blocked) it's essentially knocked unconscious. When it becomes the currently-running thread again, it doesn't know that it ever stopped.

# Marriage in Trouble.

## Can this couple be saved?

***Next, on a very special Dr.Steve Show***

[Transcript from episode #42]

Welcome to the Dr. Steve show.

We've got a story today that's centered around the top two reasons why couples split up—finances and sleep.

Ryan and Monica: victims of the "two people, one account" problem.

Today's troubled pair, Ryan and Monica, share a bed and a bank account. But not for long if we can't find a solution. The problem? The classic "two people—one bank account" thing.

Here's how Monica described it to me:

"Ryan and I agreed that neither of us will overdraw the checking account. So the procedure is, whoever wants to withdraw money *must* check the balance in the account *before* making the withdrawal. It all seemed so simple. But suddenly we're bouncing checks and getting hit with overdraft fees!

I thought it wasn't possible, I thought our procedure was safe. But then *this* happened:

Ryan needed $50, so he checked the balance in the account, and saw that it was $100. No problem. So, he plans to withdraw the money. **But first he falls asleep!**

Ryan falls asleep after he checks the balance but before he makes the withdrawal. When he wakes up, he immediately makes the withdrawl without checking the balance again.

And that's where *I* come in, while Ryan's still asleep, and now *I* want to withdraw $100. I check the balance, and it's $100 (because Ryan's still asleep and hasn't yet made his withdrawal), so I think, no problem. So I make the withdrawal, and again no problem. But then Ryan wakes up, completes *his* withdrawal, and we're suddenly overdrawn! He didn't even know that he fell asleep, so he just went ahead and completed his transaction without checking the balance again. You've got to help us Dr. Steve!"

Is there a solution? Are they doomed? We can't stop Ryan from falling asleep, but can we make sure that Monica can't get her hands on the bank account until after he wakes up?

Take a moment and think about that while we go to a commercial break.

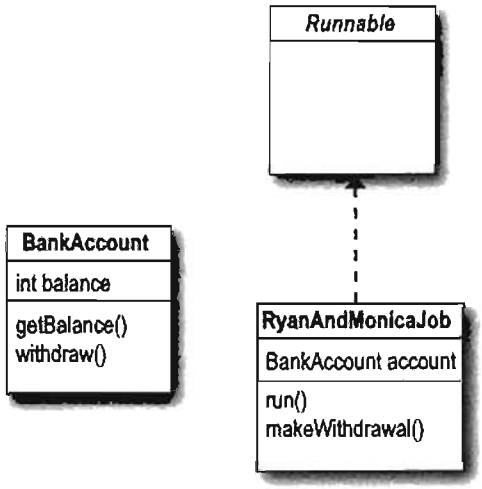
# The Ryan and Monica problem, in code

The following example shows what can happen when *two* threads (Ryan and Monica) share a *single* object (the bank account).

The code has two classes, BankAccount, and MonicaAndRyanJob. The MonicaAndRyanJob class implements Runnable, and represents the behavior that Ryan and Monica both have—checking the balance and making withdrawals. But of course, each thread falls asleep *in between* checking the balance and actually making the withdrawal.

The MonicaAndRyanJob class has an instance variable of type BankAccount., that represents their shared account.

The code works like this:




### Make one instance of RyanAndMonicaJob.

The RyanAndMonicaJob class is the Runnable (the job to do), and since both Monica and Ryan do the same thing (check balance and withdraw money), we need only one instance.

```
RyanAndMonicaJob  theJob = new RyanAndMonicaJob();
```

### Make two threads with the same Runnable

(the RyanAndMonicaJob instance)

```
Thread one = new Thread(theJob);
Thread two = new Thread(theJob);
```

### Name and start the threads

```
one.setName("Ryan");
two.setName("Monica");
one.start();
two.start();
```

### Watch both threads execute the run() method

(check the balance and make a withdrawal)

One thread represents Ryan, the other represents Monica. Both threads continually check the balance and then make a withdrawal, but only if it's safe!

```
if (account.getBalance() >= amount) {
   try {
      Thread.sleep(500);
   } catch(InterruptedException ex) {ex.printStackTrace(); }
}
```

**In the run() method, do exactly what Ryan and Monica would do—check the balance and, if there's enough money, make the withdrawal.**

**This should protect against overdrawing the account.**

**Except... Ryan and Monica always fall asleep after they check the balance but before they finish the withdrawal.**

## The Ryan and Monica example

```
class BankAccount {
   private int balance = 100;
   public int getBalance() {
      return balance;
   }
   public void withdraw(int amount) {
      balance = balance - amount;
   }
}

 public class RyanAndMonicaJob implements Runnable {

   private BankAccount account = new BankAccount();

   public static void main (String [] args) {
      RyanAndMonicaJob  theJob = new RyanAndMonicaJob();
      Thread one = new Thread(theJob);
      Thread two = new Thread(theJob);
      one.setName("Ryan");
      two.setName("Monica");
      one.start();
      two.start();
    }

  public void run() {
   for (int x = 0; x < 10; x++) {
      makeWithdrawl(10);
      if (account.getBalance() < 0) {
        System.out.println("Overdrawn!");
      }
    }
 }

  private void makeWithdrawal(int amount) {
     if (account.getBalance() >= amount) {
     System.out.println(Thread.currentThread().getName() + " is about to withdraw");
      try {
        System.out.println(Thread.currentThread().getName() + " is going to sleep");
        Thread.sleep(500);
      } catch(InterruptedException ex) {ex.printStackTrace(); }
      System.out.println(Thread.currentThread().getName() + " woke up.");
      account.withdraw(amount);
      System.out.println(Thread.currentThread().getName() + " completes the withdrawl");
      }
    else {
      System.out.println("Sorry, not enough for " + Thread.currentThread().getName());
    }
  }
}
```

The account starts with a balance of $100.

There will be only ONE instance of the RyanAndMonicaJob. That means only ONE instance of the bank account. Both threads will access this one account

Instantiate the Runnable (job)

Make two threads, giving each thread the same Runnable job. That means both threads will be accessing the one account instance variable in the Runnable class.

In the run() method, a thread loops through and tries to make a withdrawal with each iteration. After the withdrawal, it checks the balance once again to see if the account is overdrawn.

Check the account balance, and if there's not enough money, we just print a message. If there IS enough, we go to sleep, then wake up and complete the withdrawal, just like Ryan did.

We put in a bunch of print statements so we can see what's happening as it runs.

Ryan and Monica output

```
File Edit Window Help Visa
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawl
Monica is about to withdraw
Monica is going to sleep
Ryan woke up.
Ryan completes the withdrawl
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawl
Monica is about to withdraw
Monica is going to sleep
Ryan woke up.
Ryan completes the withdrawl
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up.
Monica completes the withdrawl
Sorry, not enough for Monica
Sorry, not enough for Monica
Sorry, not enough for Monica
Sorry, not enough for Monica
Sorry, not enough for Monica
Ryan woke up.
Ryan completes the withdrawl
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
```

How did this happen?

**The makeWithdrawal() method always checks the balance before making a withdrawal, but still we overdraw the account.**

**Here's one scenario:**

Ryan checks the balance, sees that there's enough money, and then falls asleep.

Meanwhile, Monica comes in and checks the balance. She, too, sees that there's enough money. She has no idea that Ryan is going to wake up and complete a withdrawal.

Monica falls asleep.

Ryan wakes up and completes his withdrawal.

Monica wakes up and completes her withdrawal. Big Problem! In between the time when she checked the balance and made the withdrawal, Ryan woke up and pulled money from the account.

**Monica's check of the account was not valid, because Ryan had already checked and was still in the middle of making a withdrawal.**

Monica must be stopped from getting into the account until Ryan wakes up and finishes his transaction. And vice-versa.

# They need a lock for account access!

**The lock works like this:**

1. There's a lock associated with the bank account transaction (checking the balance and withdrawing money). There's only one key, and it stays with the lock until somebody wants to access the account.

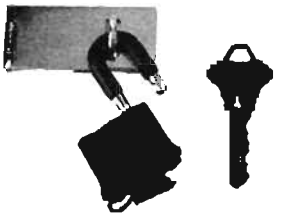
**The bank account transaction is unlocked when nobody is using the account.**

2. When Ryan wants to access the bank account (to check the balance and withdraw money), he locks the lock and puts the key in his pocket. Now nobody else can access the account, since the key is gone.

**When Ryan wants to access the account, he secures the lock and takes the key.**

3. **Ryan keeps the key in his pocket until he finishes the transaction.** He has the only key, so Monica can't access the account (or the checkbook) until Ryan unlocks the account and returns the key.

   Now, even if Ryan falls asleep after he checks the balance, he has a guarantee that the balance will be the same when he wakes up, because he kept the key while he was asleep!



**When Ryan is finished, he unlocks the lock and returns the key. Now the key is available for Monica (or Ryan again) to access the account.**

# We need the makeWithdrawal() method to run as one atomic thing.

We need to make sure that once a thread enters the makeWithdrawal() method, *it must be allowed to finish the method* before any other thread can enter.

In other words, we need to make sure that once a thread has checked the account balance, that thread has a guarantee that it can wake up and finish the withdrawal *before any other thread can check the account balance!*

Use the **synchronized** keyword to modify a method so that only one thread at a time can access it.

That's how you protect the bank account! You don't put a lock on the bank account itself; you lock the method that does the banking transaction. That way, one thread gets to complete the whole transaction, start to finish, even if that thread falls asleep in the middle of the method!

So if you don't lock the back account, then what exactly *is* locked? Is it the method? The Runnable object? The thread itself?

We'll look at that on the next page. In code, though, it's quite simple—just add the synchronized modifier to your method declaration:

**The synchronized keyword means that a thread needs a key in order to access the synchronized code.**

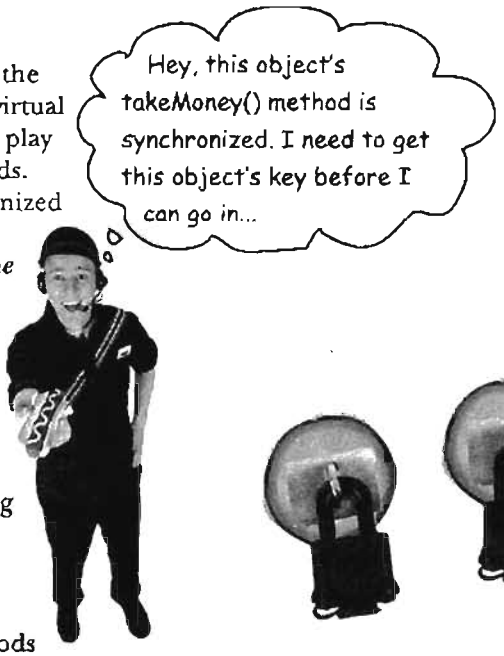
**To protect your *data* (like the bank account), synchronize the *methods* that act on that data.**

```
private synchronized void makeWithdrawal(int amount) {

   if (account.getBalance() >= amount) {
      System.out.println(Thread.currentThread().getName() + " is about to withdraw");
      try {
         System.out.println(Thread.currentThread().getName() + " is going to sleep");
         Thread.sleep(500);
      } catch(InterruptedException ex) {ex.printStackTrace(); }
      System.out.println(Thread.currentThread().getName() + " woke up.");
      account.withdraw(amount);
      System.out.println(Thread.currentThread().getName() + " completes the withdrawl");
   } else {
      System.out.println("Sorry, not enough for " + Thread.currentThread().getName());
   }
}
```

(Note for you physics-savvy readers: yes, the convention of using the word 'atomic' here does not reflect the whole subatomic particle thing. Think Newton, not Einstein, when you hear the word 'atomic' in the context of threads or transactions. Hey, it's not OUR convention. If WE were in charge, we'd apply Heisenberg's Uncertainty Principle to pretty much everything related to threads.)

# Using an object's lock

Every object has a lock. Most of the time, the lock is unlocked, and you can imagine a virtual key sitting with it. Object locks come into play only when there are synchronized methods. When an object has one or more synchronized methods, ***a thread can enter a synchronized method only if the thread can get the key to the object's lock!***



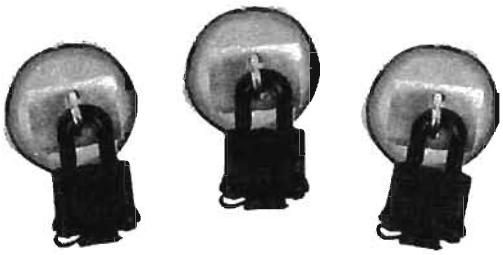

The locks are not per *method*, they are per *object*. If an object has two synchronized methods, it does not simply mean that you can't have two threads entering the same method. It means you can't have two threads entering *any* of the synchronized methods.

Think about it. If you have multiple methods that can potentially act on an object's instance variables, all those methods need to be protected with synchronized.

The goal of synchronization is to protect critical data. But remember, you don't lock the data itself, you synchronize the methods that *access* that data.

So what happens when a thread is cranking through its call stack (starting with the run() method) and it suddenly hits a synchronized method? The thread recognizes that it needs a key for that object before it can enter the method. It looks for the key (this is all handled by the JVM; there's no API in Java for accessing object locks), and if the key is available, the thread grabs the key and enters the method.

From that point forward, the thread hangs on to that key like the thread's life depends on it. The thread won't give up the key until it completes the synchronized method. So while that thread is holding the key, no other threads can enter *any* of that object's synchronized methods, because the one key for that object won't be available.



Every Java object has a lock. A lock has only one key.

Most of the time, the lock is unlocked and nobody cares.

But if an object has synchronized methods, a thread can enter one of the synchronized methods ONLY if the key for the object's lock is available. In other words, only if another thread hasn't already grabbed the one key.

# The dreaded "Lost Update" problem

Here's another classic concurrency problem, that comes from the database world. It's closely related to the Ryan and Monica story, but we'll use this example to illustrate a few more points.

The lost update revolves around one process:

Step 1: Get the balance in the account

```
int i = balance;
```

Step 2: Add 1 to that balance

```
balance = i + 1;
```

The trick to showing this is to force the computer to take two steps to complete the change to the balance. In the real world, you'd do this particular move in a single statement: `balance++;`

But by forcing it into *two* steps, the problem with a non-atomic process will become clear. So imagine that rather than the trivial "get the balance and then add 1 to the current balance" steps, the two (or more) steps in this method are much more complex, and couldn't be done in one statement.

In the "Lost Update" problem, we have two threads, both trying to increment the balance.

```
class TestSync implements Runnable {

    private int balance;

    public void run() {
       for(int i = 0; i < 50; i++) {
        increment();
        System.out.println("balance is " + balance);
      }
    }

    public void increment() {
       int i = balance;
       balance = i + 1;
    }
}

public class TestSyncTest {
   public static void main (String[] args) {
     TestSync job = new TestSync();
     Thread a = new Thread(job);
     Thread b = new Thread(job);
     a.start();
     b.start();
   }
}
```

each thread runs 50 times, incrementing the balance on each iteration

Here's the crucial part! We increment the balance by adding 1 to whatever the value of balance was AT THE TIME WE READ IT (rather than adding 1 to whatever the CURRENT value is)

# Let's run this code...

1. Thread A runs for awhile


Put the value of balance into variable i.
Balance is 0, so i is now 0.
Set the value of balance to the result of i + 1.
Now balance is 1.
Put the value of balance into variable i.
Balance is 1, so i is now 1.
Set the value of balance to the result of i + 1.
Now balance is 2.

2. Thread B runs for awhile


Put the value of balance into variable i.
Balance is 2, so i is now 2.
Set the value of balance to the result of i + 1.
Now balance is 3.
Put the value of balance into variable i.
Balance is 3, so i is now 3.

*[now thread B is sent back to runnable,*
***before** it sets the value of balance to 4]*

3. Thread A runs again, picking up where it left off


Put the value of balance into variable i.
Balance is 3, so i is now 3.
Set the value of balance to the result of i + 1.
Now balance is 4.
Put the value of balance into variable i.
Balance is 4, so i is now 4.
Set the value of balance to the result of i + 1.
Now balance is 5.

4. Thread B runs again, and picks up exactly where it left off!


Set the value of balance to the result of i + 1.
**Now balance is 4.**

Yikes!!

Thread A updated it to 5, but now B came back and stepped on top of the update A made, as if A's update never happened.

**We lost the last updates that Thread A made! Thread B had previously done a 'read' of the value of balance, and when B woke up, it just kept going as if it never missed a beat.**

# Make the increment() method atomic. Synchronize it!

Synchronizing the increment() method solves the "Lost Update" problem, because it keeps the two steps in the method as one unbreakable unit.

```
public synchronized void increment() {
    int i = balance;
    balance = i + 1;
}
```

**Once a thread enters the method, we have to make sure that all the steps in the method complete (as one atomic process) before any other thread can enter the method.**

## there are no Dumb Questions

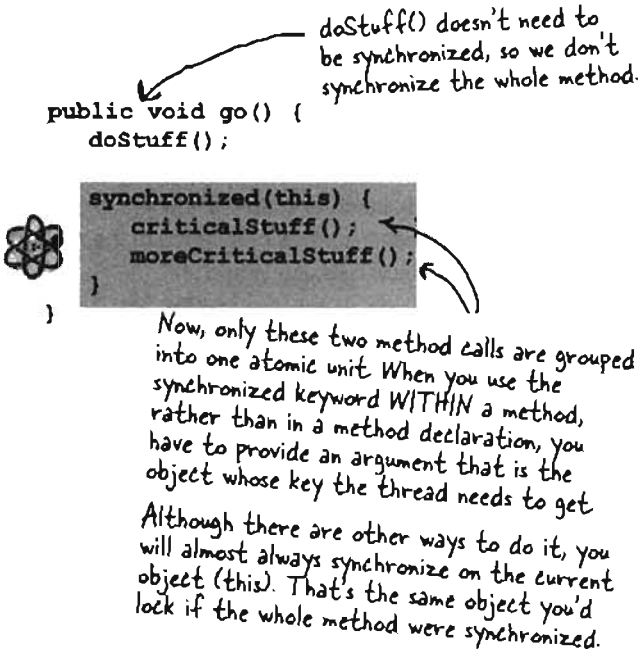
**Q: Sounds like it's a good idea to synchronize everything, just to be thread-safe.**

A: Nope, it's not a good idea. Synchronization doesn't come for free. First, a synchronized method has a certain amount of overhead. In other words, when code hits a synchronized method, there's going to be a performance hit (although typically, you'd never notice it) while the matter of "is the key available?" is resolved.

Second, a synchronized method can slow your program down because synchronization restricts concurrency. In other words, a synchronized method forces other threads to get in line and wait their turn. This might not be a problem in your code, but you have to consider it.

Third, and most frightening, synchronized methods can lead to deadlock! (See page 516.)

A good rule of thumb is to synchronize only the bare minimum that should be synchronized. And in fact, you can synchronize at a granularity that's even smaller than a method. We don't use it in the book, but you can use the synchronized keyword to synchronize at the more fine-grained level of one or more statements, rather than at the whole-method level.



### 1 Thread A runs for awhile


Attempt to enter the increment() method.

The method is synchronized, so **get the key** for this object
Put the value of balance into variable i.
Balance is 0, so i is now 0.
Set the value of balance to the result of i + 1.
Now balance is 1.
**Return the key** (it completed the increment() method).
Re-enter the increment() method and **get the key**.
Put the value of balance into variable i.
Balance is 1, so i is now 1.

*[now thread A is sent back to runnable, but since it has not completed the synchronized method, Thread A keeps the key]*

### 2 Thread B is selected to run


Attempt to enter the increment() method. The method is synchronized, so we need to get the key.

**The key is not available.**

*[now thread B is sent into a 'object lock not available lounge]*

### 3 Thread A runs again, picking up where it left off (remember, it still has the key)


Set the value of balance to the result of i + 1.
Now balance is 2.
**Return the key**.

*[now thread A is sent back to runnable, but since it has completed the increment() method, the thread does NOT hold on to the key]*

### 4 Thread B is selected to run


Attempt to enter the increment() method. The method is synchronized, so we need to get the key.

This time, the key IS available, get the key.

Put the value of balance into variable i.

[continues to run...]

# The deadly side of synchronization

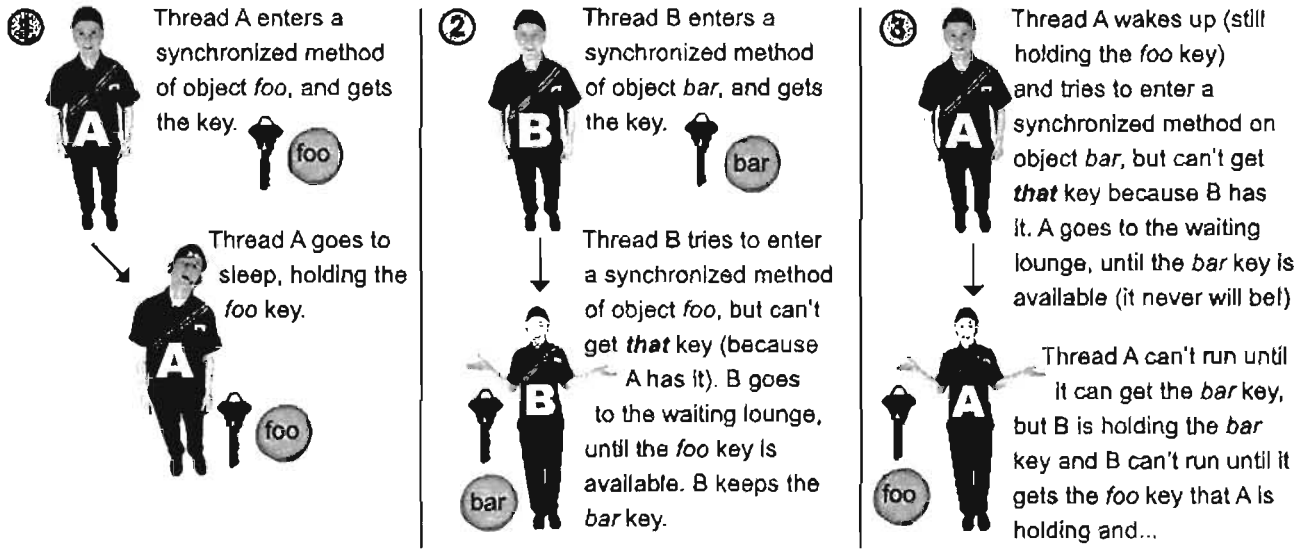
Be careful when you use synchronized code, because nothing will bring your program to its knees like thread deadlock. Thread deadlock happens when you have two threads, both of which are holding a key the other thread wants. There's no way out of this scenario, so the two threads will simply sit and wait. And wait. And wait.

If you're familiar with databases or other application servers, you might recognize the problem; databases often have a locking mechanism somewhat like synchronization. But a real transaction management system can sometimes deal with deadlock. It might assume, for example, that deadlock might have occurred when two transactions are taking too long to complete. But unlike Java, the application server can do a "transaction rollback" that returns the state of the rolled-back transaction to where it was before the transaction (the atomic part) began.

Java has no mechanism to handle deadlock. It won't even *know* deadlock occurred. So it's up to you to design carefully. If you find yourself writing much multithreaded code, you might want to study "Java Threads" by Scott Oaks and Henry Wong for design tips on avoiding deadlock. One of the most common tips is to pay attention to the order in which your threads are started.

**All it takes for deadlock are two objects and two threads.**


## A simple deadlock scenario:



## BULLET POINTS

- The static Thread.sleep() method forces a thread to leave the running state for at least the duration passed to the sleep method. Thread.sleep(200) puts a thread to sleep for 200 milliseconds.
- The sleep() method throws a checked exception (InterruptedException), so all calls to sleep() must be wrapped in a try/catch, or declared.
- You can use sleep() to help make sure all threads get a chance to run, although there's no guarantee that when a thread wakes up it'll go to the end of the runnable line. It might, for example, go right back to the front. In most cases, appropriately-timed sleep() calls are all you need to keep your threads switching nicely.
- You can name a thread using the (yet another surprise) setName() method. All threads get a default name, but giving them an explicit name can help you keep track of threads, especially if you're debugging with print statements.
- You can have serious problems with threads if two or more threads have access to the same object on the heap.
- Two or more threads accessing the same object can lead to data corruption if one thread, for example, leaves the running state while still in the middle of manipulating an object's critical state.
- To make your objects thread-safe, decide which statements should be treated as one atomic process. In other words, decide which methods must run to completion before another thread enters the same method on the same object.
- Use the keyword **`synchronized`** to modify a method declaration, when you want to prevent two threads from entering that method.
- Every object has a single lock, with a single key for that lock. Most of the time we don't care about that lock; locks come into play only when an object has synchronized methods.
- When a thread attempts to enter a synchronized method, the thread must get the key for the object (the object whose method the thread is trying to run). If the key is not available (because another thread already has it), the thread goes into a kind of waiting lounge, until the key becomes available.
- Even if an object has more than one synchronized method, there is still only one key. Once any thread has entered a synchronized method on that object, no thread can enter any other synchronized method on the same object. This restriction lets you protect your data by synchronizing any method that manipulates the data.

# New and improved SimpleChatClient

Way back near the beginning of this chapter, we built the SimpleChatClient that could *send* outgoing messages to the server but couldn't receive anything. Remember? That's how we got onto this whole thread topic in the first place, because we needed a way to do two things at once: send messages *to* the server (interacting with the GUI) while simultaneously reading incoming messages *from* the server, displaying them in the scrolling text area.

Yes, there really IS an end to this chapter. But not yet...

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleChatClient {

    JTextArea incoming;
    JTextField outgoing;
    BufferedReader reader;
    PrintWriter writer;
    Socket sock;

    public static void main(String[] args) {
       SimpleChatClient client = new SimpleChatClient();
       client.go();
    }

    public void go() {
```

This is mostly GUI code you've seen before. Nothing special except the highlighted part where we start the new 'reader' thread.

```
        JFrame frame = new JFrame("Ludicrously Simple Chat Client");
        JPanel mainPanel = new JPanel();
        incoming = new JTextArea(15,50);
        incoming.setLineWrap(true);
        incoming.setWrapStyleWord(true);
        incoming.setEditable(false);
        JScrollPane qScroller = new JScrollPane(incoming);
        qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        outgoing = new JTextField(20);
        JButton sendButton = new JButton("Send");
        sendButton.addActionListener(new SendButtonListener());
        mainPanel.add(qScroller);
        mainPanel.add(outgoing);
        mainPanel.add(sendButton);
        setUpNetworking();

        Thread readerThread = new Thread(new IncomingReader());
        readerThread.start();

        frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
        frame.setSize(400,500);
        frame.setVisible(true);

    } // close go
```

We're starting a new thread, using a new inner class as the Runnable (job) for the thread. The thread's job is to read from the server's socket stream, displaying any incoming messages in the scrolling text area.

```
private void setUpNetworking() {

      try {
         sock = new Socket("127.0.0.1", 5000);
         InputStreamReader streamReader = new InputStreamReader(sock.getInputStream());
         reader = new BufferedReader(streamReader);
         writer = new PrintWriter(sock.getOutputStream());
         System.out.println("networking established");
      } catch(IOException ex) {
         ex.printStackTrace();
      }
  } // close setUpNetworking
```

We're using the socket to get the input and output streams. We were already using the output stream to send to the server, but now we're using the input stream so that the new 'reader' thread can get messages from the server.

```
public class SendButtonListener implements ActionListener {
   public void actionPerformed(ActionEvent ev) {
        try {
           writer.println(outgoing.getText());
           writer.flush();

        } catch(Exception ex) {
           ex.printStackTrace();
        }
        outgoing.setText("");
        outgoing.requestFocus();
   }
 }  // close inner class
```

Nothing new here. When the user clicks the send button, this method sends the contents of the text field to the server.

```
public class IncomingReader implements Runnable {
   public void run() {
         String message;
         try {

           while ((message = reader.readLine()) != null) {
              System.out.println("read " + message);
              incoming.append(message + "\n");

            } // close while
         } catch(Exception ex) {ex.printStackTrace();}
     } // close run
 } // close inner class
} // close outer class
```

This is what the thread does!!

In the run() method, it stays in a loop (as long as what it gets from the server is not null), reading a line at a time and adding each line to the scrolling text area (along with a new line character).


## The really really simple Chat Server

You can use this server code for both versions of the Chat Client. Every possible disclaimer ever disclaimed is in effect here. To keep the code stripped down to the bare essentials, we took out a lot of parts that you'd need to make this a real server. In other words, it works, but there are at least a hundred ways to break it. If you want a Really Good Sharpen Your Pencil for after you've finished this book, come back and make this server code more robust.

Another possible Sharpen Your Pencil, that you could do right now, is to annotate this code yourself. You'll understand it much better if you work out what's happening than if we explained it to you. Then again, this is Ready-bake code, so you really don't have to understand it at all. It's here just to support the two versions of the Chat Client.

**To run the chat client, you need two terminals. First, launch this server from one terminal, then launch the client from another terminal**

```
import java.io.*;
import java.net.*;
import java.util.*;

public class VerySimpleChatServer {

    ArrayList clientOutputStreams;

    public class ClientHandler implements Runnable {
         BufferedReader reader;
         Socket sock;

         public ClientHandler(Socket clientSocket) {
           try {
             sock = clientSocket;
             InputStreamReader isReader = new InputStreamReader(sock.getInputStream());
             reader = new BufferedReader(isReader);

           } catch(Exception ex) {ex.printStackTrace();}
          } // close constructor

        public void run() {
           String message;
           try {
             while ((message = reader.readLine()) != null) {
                System.out.println("read " + message);
                tellEveryone(message);

              } // close while
           } catch(Exception ex) {ex.printStackTrace();}
       } // close run
    } // close inner class
```

```
 public static void main (String[] args) {
      new VerySimpleChatServer().go();
 }

 public void go() {
    clientOutputStreams = new ArrayList();
    try {
      ServerSocket serverSock = new ServerSocket(5000);

      while(true) {
         Socket clientSocket = serverSock.accept();
         PrintWriter writer = new PrintWriter(clientSocket.getOutputStream());
         clientOutputStreams.add(writer);

         Thread t = new Thread(new ClientHandler(clientSocket));
         t.start();
         System.out.println("got a connection");
      }

    } catch(Exception ex) {
      ex.printStackTrace();
    }
} // close go

public void tellEveryone(String message) {

   Iterator it = clientOutputStreams.iterator();
   while(it.hasNext()) {
      try {
         PrintWriter writer = (PrintWriter) it.next();
         writer.println(message);
         writer.flush();
      } catch(Exception ex) {
          ex.printStackTrace();
      }

   } // end while

} // close tellEveryone
} close class
```

## there are no Dumb Questions

**Q: What about protecting static variable state? If you have static methods that change the static variable state, can you still use synchronization?**

A: Yes! Remember that static methods run against the class and not against an individual instance of the class. So you might wonder whose object's lock would be used on a static method? After all, there might not even *be* any instances of that class. Fortunately, just as each *object* has its own lock, each loaded *class* has a lock. That means that if you have three Dog objects on your heap, you have a total of four Dog-related locks. Three belonging to the three Dog instances, and one belonging to the Dog class itself. When you synchronize a static method, Java uses the lock of the class itself. So if you synchronize two static methods in a single class, a thread will need the class lock to enter *either* of the methods.

**Q: What are thread priorities? I've heard that's a way you can control scheduling.**

A: Thread priorities *might* help you influence the scheduler, but they still don't offer any guarantee. Thread priorities are numerical values that tell the scheduler (if it cares) how important a thread is to you. In general, the scheduler will kick a lower priority thread out of the running state if a higher priority thread suddenly becomes runnable. But... one more time, say it with me now, "there is no guarantee." We recommend that you use priorities only if you want to influence *performance*, but never, ever rely on them for program correctness.

**Q: Why don't you just synchronize all the getters and setters from the class with the data you're trying to protect? Like, why couldn't we have synchronized just the checkBalance() and withdraw() methods from class BankAccount, instead of synchronizing the makeWithdrawal() method from the Runnable's class?**

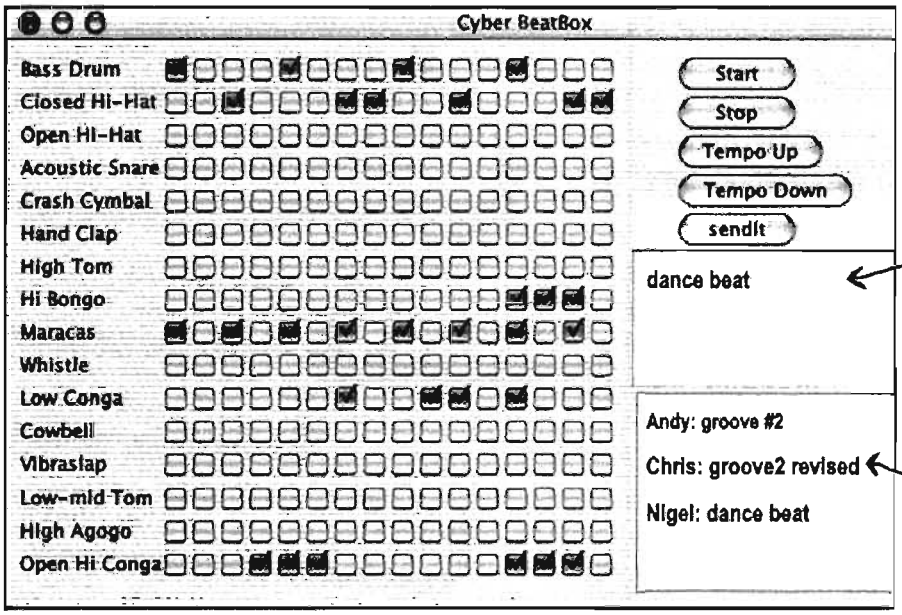
A: Actually, we *should* have synchronized those methods, to prevent other threads from accessing those methods in other ways. We didn't bother, because our example didn't have any other code accessing the account.

But synchronizing the getters and setters (or in this case the checkBalance() and withdraw()) isn't enough. Remember, the point of synchronization is to make a specific section of code work ATOMICALLY. In other words, it's not just the individual methods we care about, it's methods that require ***more than one step to complete***! Think about it. If we had not synchronized the makeWithdrawal() method, Ryan would have checked the balance (by calling the synchronized checkBalance()), and then immediately exited the method and returned the key!

Of course he would grab the key again, after he wakes up, so that he can call the synchronized withdraw() method, but this still leaves us with the same problem we had before synchronization! Ryan can check the balance, go to sleep, and Monica can come in and also check the balance before Ryan has a chance to wakes up and completes his withdrawal.

So synchronizing all the access methods is probably a good idea, to prevent other threads from getting in, but you still need to synchronize the methods that have statements that must execute as one atomic unit.

# Code Kitchen




your message gets sent to the other players, along with your current beat pattern, when you hit "sendIt"

incoming messages from players. Click one to load the pattern that goes with it, and then click 'Start' to play it

This is the last version of the BeatBox!

It connects to a simple MusicServer so that you can send and receive beat patterns with other clients.

The code is really long, so the complete listing is actually in Appendix A.


# Code Magnets

A working Java program is scrambled up on the fridge. Can you add the code snippets on the next page to the empty classes below, to make a working Java program that produces the output listed? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```
public class TestThreads {
```

```
class Accum {
```

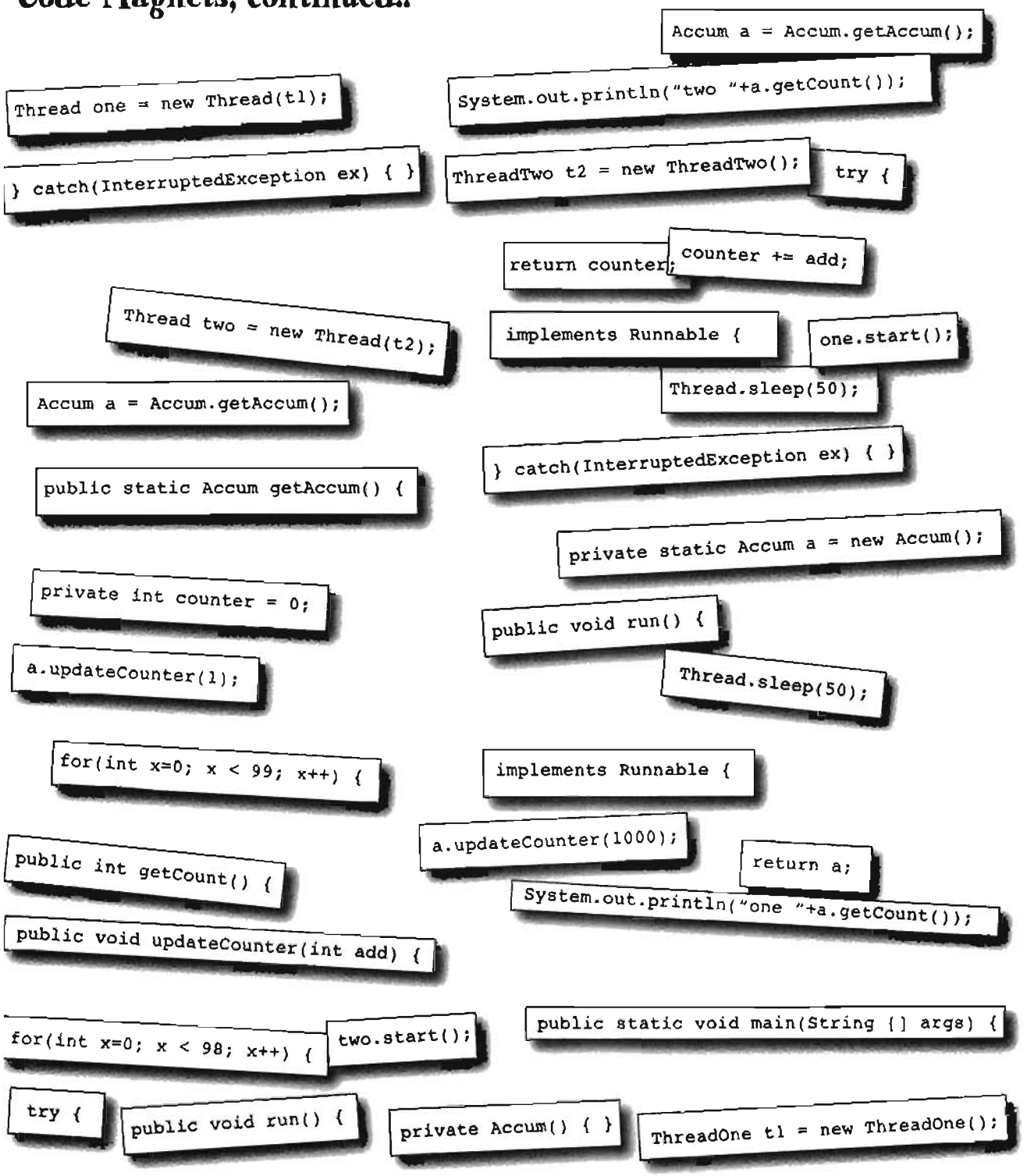
```
class ThreadOne
```

```
class ThreadTwo
```

```
File Edit Window Help Sewing
% java TestThreads
one 98098
two 98099
```

**Bonus Question:** Why do you think we used the modifiers we did in the Accum class?

# Code Magnets, continued..



# Exercise Solutions

```
public class TestThreads {
  public static void main(String [] args) {
    ThreadOne t1 = new ThreadOne();
    ThreadTwo t2 = new ThreadTwo();
    Thread one = new Thread(t1);
    Thread two = new Thread(t2);
    one.start();
    two.start();
  }
}

class Accum {
  private static Accum a = new Accum();
  private int counter = 0;

  private Accum() { }

  public static Accum getAccum() {
    return a;
  }

  public void updateCounter(int add) {
    counter += add;
  }

  public int getCount() {
    return counter;
  }
}

class ThreadOne implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 98; x++) {
      a.updateCounter(1000);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("one "+a.getCount());
  }
}
```

create a static instance of class Accum

A private constructor

Threads from two different classes are updating the same object in a third class, because both threads are accessing a single instance of Accum. The line of code:

private static Accum a = new Accum(); creates a static instance of Accum (remember static means one per class), and the private constructor in Accum means that no one else can make an Accum object. These two techniques (private constructor and static getter method) used together, create what's known as a 'Singleton' - an OO pattern to restrict the number of instances of an object that can exist in an application. (Usually, there's just a single instance of a Singleton—hence the name), but you can use the pattern to restrict the instance creation in whatever way you choose.)

```
class ThreadTwo implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 99; x++) {
      a.updateCounter(1);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("two "+a.getCount());
  }
}
```

## Near-miss at the Airlock

As Sarah joined the on-board development team's design review meeting , she gazed out the portal at sunrise over the Indian Ocean. Even though the ship's conference room was incredibly claustrophobic, the sight of the growing blue and white crescent overtaking night on the planet below filled Sarah with awe and appreciation.

Five-Minute Mystery

This morning's meeting was focused on the control systems for the orbiter's airlocks. As the final construction phases were nearing their end, the number of spacewalks was scheduled to increase dramatically, and traffic was high both in and out of the ship's airlocks. "Good morning Sarah", said Tom, "Your timing is perfect, we're just starting the detailed design review."

"As you all know", said Tom, "Each airlock is outfitted with space-hardened GUI terminals, both inside and out. Whenever spacewalkers are entering or exiting the orbiter they will use these terminals to initiate the airlock sequences." Sarah nodded, "Tom can you tell us what the method sequences are for entry and exit?" Tom rose, and floated to the whiteboard, "First, here's the exit sequence method's pseudocode", Tom quickly wrote on the board.

```
orbiterAirlockExitSequence()

  verifyPortalStatus();

  pressurizeAirlock();

  openInnerHatch();

  confirmAirlockOccupied();

  closeInnerHatch();

  decompressAirlock();

  openOuterHatch();

  confirmAirlockVacated();

  closeOuterHatch();
```

"To ensure that the sequence is not interrupted, we have synchronized all of the methods called by the orbiterAirlockExitSequence() method", Tom explained. "We'd hate to see a returning spacewalker inadvertently catch a buddy with his space pants down!"

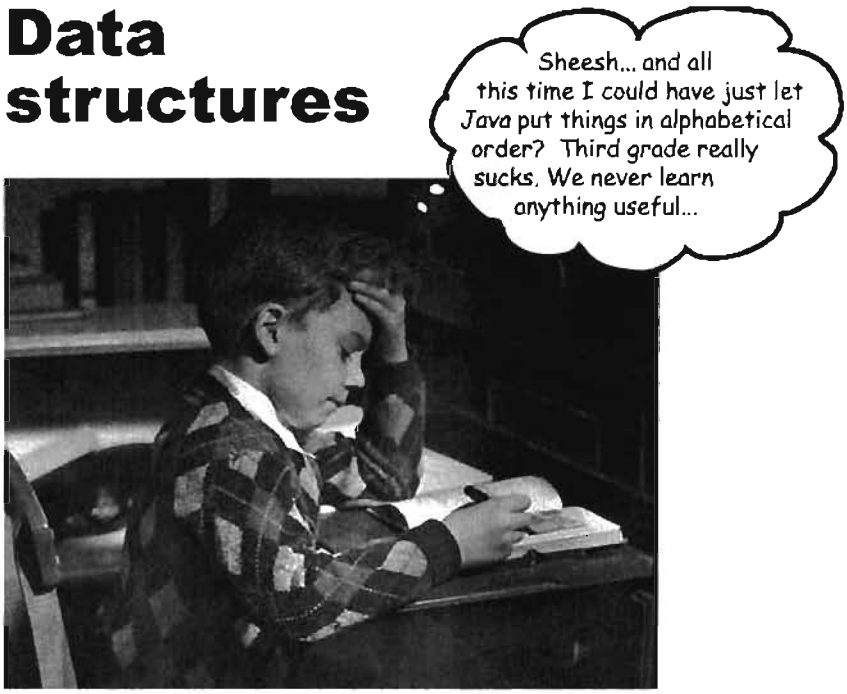
Everyone chuckled as Tom erased the whiteboard, but something didn't feel right to Sarah and it finally clicked as Tom began to write the entry sequence pseudocode on the whiteboard. "Wait a minute Tom!", cried Sarah, "I think we've got a big flaw in the exit sequence design, let's go back and revisit it, it could be critical!"

*Why did Sarah stop the meeting? What did she suspect?*

## What did Sarah know?

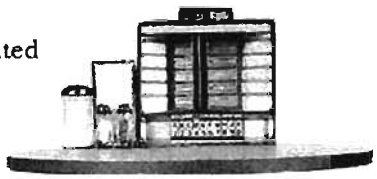
Sarah realized that in order to ensure that the entire exit sequence would run without interruption the `orbiterAirlockExitSequence()` method needed to be synchronized. As the design stood, it would be possible for a returning spacewalker to interrupt the Exit Sequence! The Exit Sequence thread couldn't be interrupted in the middle of any of the lower level method calls, but it *could* be interrupted in *between* those calls. Sarah knew that the entire sequence should be run as one atomic unit, and if the `orbiterAirlockExitSequence()` method was synchronized, it could not be interrupted at any point.

# Data structures




**Sorting is a snap in Java.** You have all the tools for collecting and manipulating your data without having to write your own sort algorithms (unless you're reading this right now sitting in your Computer Science 101 class, in which case, trust us—you are SO going to be writing sort code while the rest of us just call a method in the Java API). The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep adding to? Want to find something by name? Want to create a list that automatically takes out all the duplicates? Sort your co-workers by the number of times they've stabbed you in the back? Sort your pets by number of tricks learned? It's all here...

# Tracking song popularity on your jukebox



Congratulations on your new job—managing the automated jukebox system at Lou's Diner. There's no Java inside the jukebox itself, but each time someone plays a song, the song data is appended to a simple text file.

Your job is to manage the data to track song popularity, generate reports, and manipulate the playlists. You're not writing the entire app—some of the other software developer/waiters are involved as well, but you're responsible for managing and sorting the data inside the Java app. And since Lou has a thing against databases, this is strictly an in-memory data collection. All you get is the file the jukebox keeps adding to. Your job is to take it from there.

You've already figured out how to read and parse the file, and so far you've been storing the data in an ArrayList.

**SongList.txt**

```
Pink Moon/Nick Drake
Somersault/Zero 7
Shiva Moon/Prem Joshua
Circles/BT
Deep Channel/Afro Celts
Passenger/Headmix
Listen/Tahiti 80
```

This is the file the jukebox device writes. Your code must read the file, then manipulate the song data.

## Challenge #1
## Sort the songs in alphabetical order

You have a list of songs in a file, where each line represents one song, and the title and artist are separated with a forward slash. So it should be simple to parse the line, and put all the songs in an ArrayList.

Your boss cares only about the song titles, so for now you can simply make a list that just has the song titles.

But you can see that the list is not in alphabetical order... what can you do?

You know that with an ArrayList, the elements are kept in the order in which they were inserted into the list, so putting them in an ArrayList won't take care of alphabetizing them, unless... maybe there's a sort() method in the ArrayList class?

# Here's what you have so far, without the sort:

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }

}
```

We'll store the song titles in an ArrayList of Strings.

The method that starts loading the file and then prints the contents of the songList ArrayList

Nothing special here... just read the file and call the addSong() method for each line.

The addSong method works just like the QuizCard in the I/O chapter--you break the line (that has both the title and artist) into two pieces (tokens) using the split() method.

We only want the song title, so add only the first token to the SongList (the ArrayList).

```
File Edit Window Help Dance
%java Jukebox1
[Pink Moon, Somersault,
Shiva Moon, Circles,
Deep Channel, Passenger,
Listen]
```

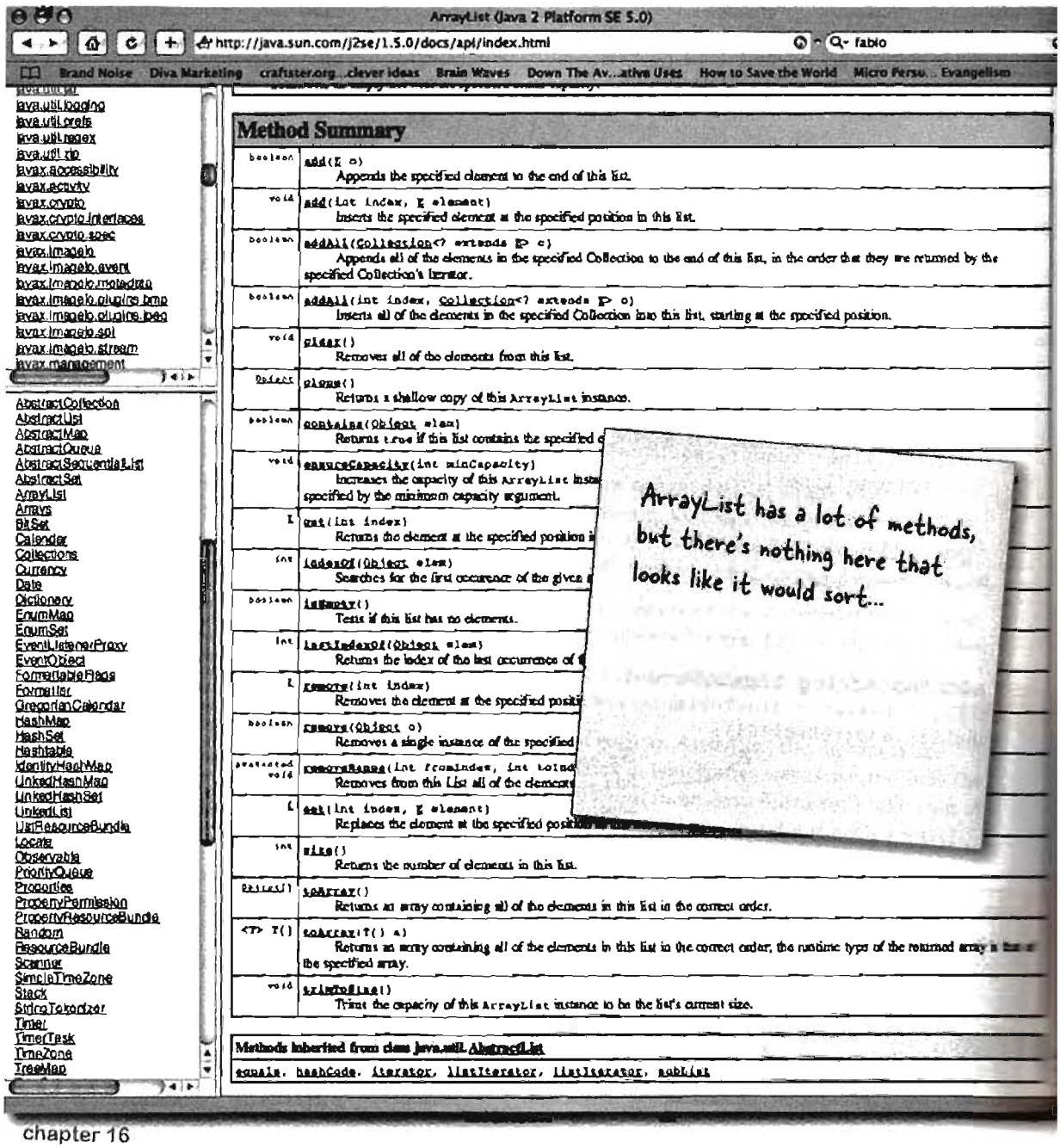
The songList prints out with the songs in the order in which they were added to the ArrayList (which is the same order the songs are in within the original text file).

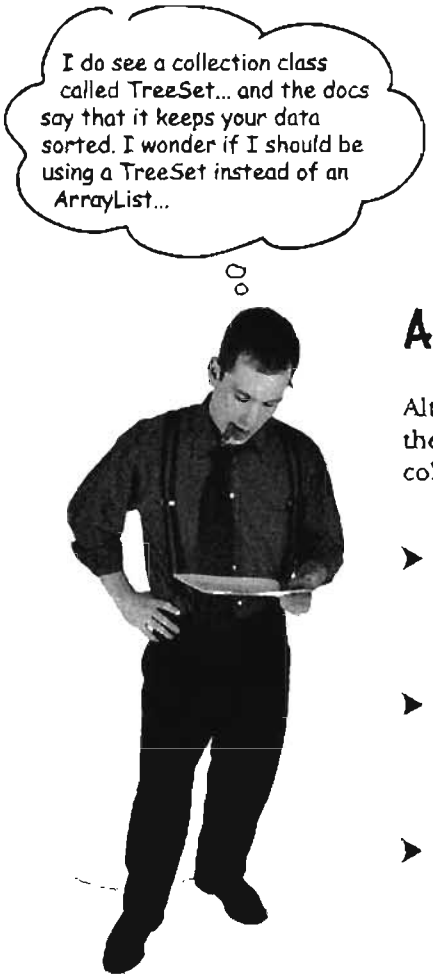
This is definitely NOT alphabetical!

ArrayList API

# But the ArrayList class does NOT have a sort() method!

When you look in ArrayList, there doesn't seem to be any method related to sorting. Walking up the inheritance hierarchy didn't help either—it's clear that *you can't call a sort method on the ArrayList.*






# ArrayList is not the only collection

Although ArrayList is the one you'll use most often, there are others for special occasions. Some of the key collection classes include:

Don't worry about trying to learn these other ones right now. We'll go into more details a little later

- **TreeSet**
  Keeps the elements sorted and prevents duplicates.

- **HashMap**
  Let's you store and access elements as name/value pairs.

- **LinkedList**
  Designed to give better performance when you insert or delete elements from the middle of the collection. (In practice, an ArrayList is still usually what you want.)

- **HashSet**
  Prevents duplicates in the collection, and given an element, can find that element in the collection quickly.

- **LinkedHashMap**
  Like a regular HashMap, except it can remember the order in which elements (name/value pairs) were inserted, or it can be configured to remember the order in which elements were last accessed.

# You *could* use a TreeSet... Or you could use the Collections.sort() method

If you put all the Strings (the song titles) into a **TreeSet** instead of an ArrayList, the Strings would automatically land in the right place, alphabetically sorted. Whenever you printed the list, the elements would always come out in alphabetical order.

And that's great when you need a *set* (we'll talk about sets in a few minutes) or when you know that the list must *always* stay sorted alphabetically.

On the other hand, if you don't need the list to stay sorted, TreeSet might be more expensive than you need—***every time you insert into a TreeSet, the TreeSet has to take the time to figure out where in the tree the new element must go.*** With ArrayList, inserts can be blindingly fast because the new element just goes in at the end.

**java.util.Collections**

```
public static void copy(List destination, List source)
public static List emptyList()
public static void fill(List listToFill, Object objToFillItWith)
public static int frequency(Collection c, Object o)
public static void reverse(List list)
public static void rotate(List list, int distance)
public static void shuffle(List list)
public static void sort(List list)
public static boole
// many more met
```

Hmmm... there IS a sort() method in the Collections class. It takes a List, and since ArrayList implements the List interface, ArrayList IS-A List. Thanks to polymorphism, you can pass an ArrayList to a method declared to take List.

**Q: But you CAN add something to an ArrayList at a specific index instead of just at the end—there's an overloaded add() method that takes an int along with the element to add. So wouldn't it be slower than inserting at the end?**

A: Yes, it's slower to insert something in an ArrayList somewhere *other* than at the end. So using the overloaded add(index, element) method doesn't work as quickly as calling the add(element)—which puts the added element at the end. But most of the time you use ArrayLists, you won't need to put something at a specific index.

**Q: I see there's a LinkedList class, so wouldn't *that* be better for doing inserts somewhere in the middle? At least if I remember my Data Structures class from college...**

A: Yes, good spot. The LinkedList *can* be quicker when you insert or remove something from the middle, but for most applications, the difference between middle inserts into a LinkedList and ArrayList is usually not enough to care about unless you're dealing with a *huge* number of elements. We'll look more at LinkedList in a few minutes.

Note: this is NOT the real Collections class API; we simplified it here by leaving out the generic type information (which you'll see in a few pages).

# Adding Collections.sort() to the Jukebox code

The Collections.sort() method sorts a list of Strings alphabetically.

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }
}
```

Call the static Collections sort() method, then print the list again. The second print out is in alphabetical order!

```
File Edit Window Help Chill
%java Jukebox1

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

Before calling sort().

After calling sort().

# But now you need Song objects, not just simple Strings.

Now your boss wants actual Song class instances in the list, not just Strings, so that each Song can have more data. The new jukebox device outputs more information, so this time the file will have *four* pieces (tokens) instead of just two.

The Song class is really simple, with only one interesting feature—the overridden toString() method. Remember, the toString() method is defined in class Object, so every class in Java inherits the method. And since the toString() method is called on an object when it's printed (System.out.println(anObject)), you should override it to print something more readable than the default unique identifier code. When you print a list, the toString() method will be called on each object.

**SongListMore.txt**

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
```

The new song file holds four attributes instead of just two. And we want ALL of them in our list, so we need to make a Song class with instance variables for all four song attributes.

```
class Song {
    String title;
    String artist;
    String rating;
    String bpm;

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;
        rating = r;
        bpm = b;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public String getRating() {
        return rating;
    }

    public String getBpm() {
        return bpm;
    }

    public String toString() {
        return title;
    }
}
```

Four instance variables for the four song attributes in the file.

The variables are all set in the constructor when the new Song is created.

The getter methods for the four attributes.

We override toString(), because when you do a System.out.println(aSongObject), we want to see the title. When you do a System.out.println(aListOfSongs), it calls the toString() method of EACH element in the list.

# Changing the Jukebox code to use Songs instead of Strings

Your code changes only a little—the file I/O code is the same, and the parsing is the same (String.split()), except this time there will be *four* tokens for each song/line, and all four will be used to create a new Song object. And of course the ArrayList will be of type <Song> instead of <String>.

```
import java.util.*;
import java.io.*;

public class Jukebox3 {

    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");

        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

Change to an ArrayList of Song objects instead of String.

Create a new Song object using the four tokens (which means the four pieces of info in the song file for this line), then add the Song to the list.

# It won't compile!

Something's wrong... the Collections class clearly shows there's a sort() method, that takes a List.

ArrayList is-a List, because ArrayList implements the List interface, so... it *should* work.

***But it doesn't!***

The compiler says it can't find a sort method that takes an ArrayList<Song>, so maybe it doesn't like an ArrayList of Song objects? It didn't mind an ArrayList<String>, so what's the important difference between Song and String? What's the difference that's making the compiler fail?

```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                           ^
1 error
```

And of course you probably already asked yourself, "What would it be sorting *on*?" How would the sort method even *know* what made one Song greater or less than another Song? Obviously if you want the song's *title* to be the value that determines how the songs are sorted, you'll need some way to tell the sort method that it needs to use the title and not, say, the beats per minute.

We'll get into all that a few pages from now, but first, let's find out why the compiler won't even let us pass a Song ArrayList to the sort() method.


## The sort() method declaration

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Google

**Method Detail**

**sort**

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

From the API docs (looking up the java.util.Collections class, and scrolling to the sort() method), it looks like the sort() method is declared... *strangely*. Or at least different from anything we've seen so far.

That's because the sort() method (along with other things in the whole collection framework in Java) makes heavy use of *generics*. Anytime you see something with angle brackets in Java source code or documentation, it means generics—a feature added to Java 5.0. So it looks like we'll have to learn how to interpret the documentation before we can figure out why we were able to sort String objects in an ArrayList, but not an ArrayList of Song objects.

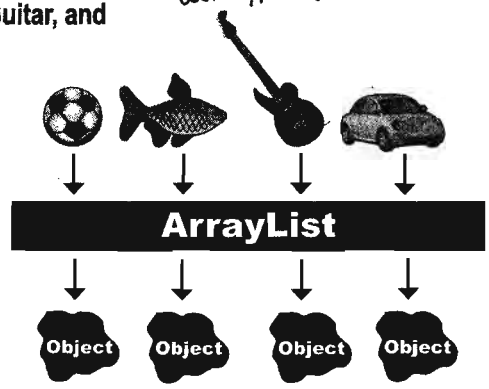
# Generics means more type-safety

We'll just say it right here—*virtually all of the code you write that deals with generics will be collection-related code.* Although generics can be used in other ways, the main point of generics is to let you write type-safe collections. In other words, code that makes the compiler stop you from putting a Dog into a list of Ducks.

Before generics (which means before Java 5.0), the compiler could not care less what you put into a collection, because all collection implementations were declared to hold type Object. You could put *anything* in any ArrayList; it was like all ArrayLists were declared as ArrayList<Object>.

**WITHOUT generics**

**Objects go IN as a reference to SoccerBall, Fish, Guitar, and Car objects**



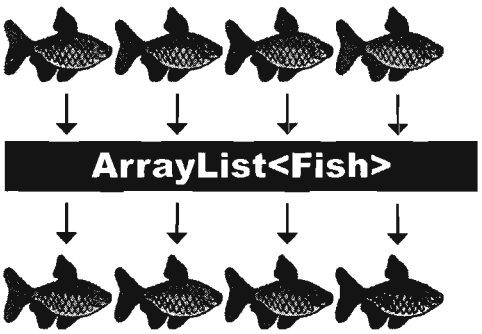

**And come OUT as a reference of type Object**

With generics, you can create type-safe collections where more problems are caught at compile-time instead of runtime.

Without generics, the compiler would happily let you put a Pumpkin into an ArrayList that was supposed to hold only Cat objects.

**WITH generics**

**Objects go IN as a reference to only Fish objects**




**And come out as a reference of type Fish**

Now with generics, you can put only Fish objects in the ArrayList<Fish>, so the objects come out as Fish references. You don't have to worry about someone sticking a Volkswagen in there, or that what you get out won't really be castable to a Fish reference.

# Learning generics

Of the dozens of things you could learn about generics, there are really only three that matter to most programmers:

- **Creating instances of generified *classes* (like ArrayList)**

  When you make an ArrayList, you have to tell it the type of objects you'll allow in the list, just as you do with plain old arrays.

```
new ArrayList<Song>()
```

- **Declaring and assigning *variables* of generic types**

  How does polymorphism really work with generic types? If you have an ArrayList<Animal> reference variable, can you assign an ArrayList<Dog> to it? What about a List<Animal> reference? Can you assign an ArrayList<Animal> to it? You'll see...

```
List<Song> songList =
      new ArrayList<Song>()
```

- **Declaring (and invoking) *methods* that take generic types**

  If you have a method that takes as a parameter, say, an ArrayList of Animal objects, what does that really mean? Can you also pass it an ArrayList of Dog objects? We'll look at some subtle and tricky polymorphism issues that are very different from the way you write methods that take plain old arrays.

  (This is actually the same point as #2, but that shows you how important we think it is.)

```
void foo(List<Song> list)

x.foo(songList)
```

**Q: But don't I also need to learn how to create my OWN generic classes? What if I want to make a class type that lets people instantiating the class decide the type of things that class will use?**

A: You probably won't do much of that. Think about it—the API designers made an entire library of collections classes covering most of the data structures you'd need, and virtually the only type of classes that really need to be generic are collection classes. In other words, classes designed to hold other elements, and you want programmers using it to specify what type those elements are when they declare and instantiate the collection class.

Yes, it is possible that you might want to *create* generic classes, but that's the exception, so we won't cover it here. (But you'll figure it out from the things we *do* cover, anyway.)

# Using generic CLASSES

Since ArrayList is our most-used generified type, we'll start by looking at its documentation. They two key areas to look at in a generified class are:

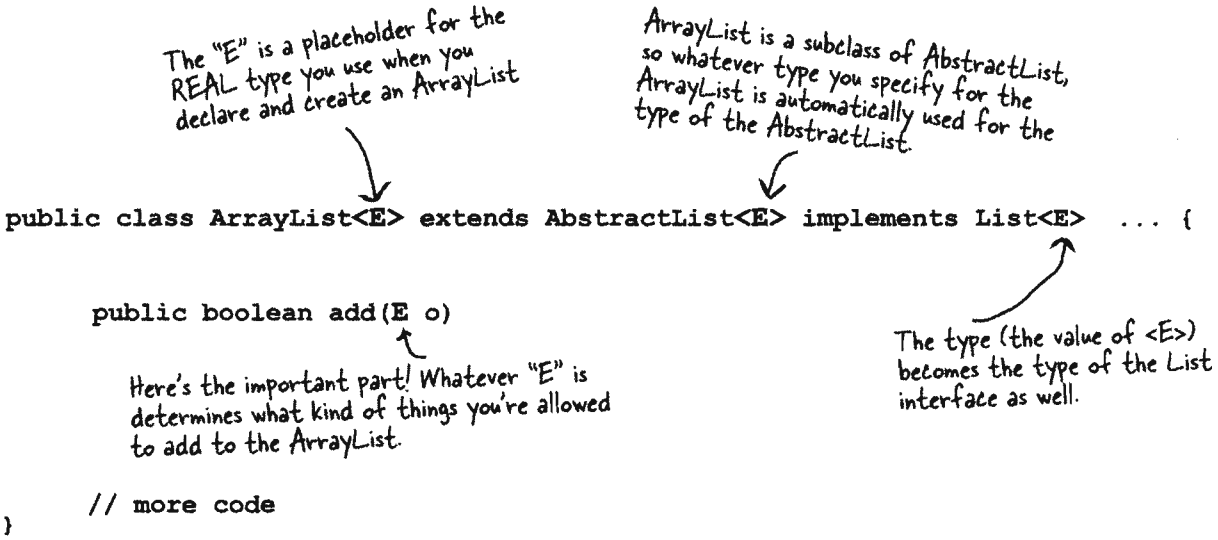
1) The *class* declaration

3) The *method* declarations that let you add elements

Think of "E" as a stand-in for "the type of element you want this collection to hold and return." (E is for Element.)

## Understanding ArrayList documentation

***(Or, what's the true meaning of "E"?)***




The "E" represents the type used to create an instance of ArrayList. When you see an "E" in the ArrayList documentation, you can do a mental find/replace to exchange it for whatever <type> you use to instantiate ArrayList.

So, new ArrayList<Song> means that "E" becomes "Song", in any method or variable declaration that uses "E".

# Using type parameters with ArrayList

**THIS code:**

```
ArrayList<String> thisList = new ArrayList<String>
```

**Means ArrayList:**

```
public class ArrayList<E> extends AbstractList<E> ... {

     public boolean add(E o)
     // more code
}
```

**Is treated by the compiler as:**

```
public class ArrayList<String> extends AbstractList<String>... {

     public boolean add(String o)
     // more code
}
```

In other words, the "E" is replaced by the *real* type (also called the *type parameter*) that you use when you create the ArrayList. And that's why the add() method for ArrayList won't let you add anything except objects of a reference type that's compatible with the type of "E". So if you make an ArrayList<**String**>, the add() method suddenly becomes **add(String o)**. If you make the ArrayList of type **Dog**, suddenly the add() method becomes **add(Dog o)**.

**Q:** **Is "E" the only thing you can put there? Because the docs for sort used "T"....**

**A:** You can use anything that's a legal Java identifier. That means anything that you could use for a method or variable name will work as a type parameter. But the convention is to use a single letter (so that's what you should use), and a further convention is to use "T" unless you're specifically writing a collection class, where you'd use "E" to represent the "type of the Element the collection will hold".

# Using generic METHODS

A generic *class* means that the *class declaration* includes a type parameter. A generic *method* means that the method declaration uses a type parameter in its signature.

You can use type parameters in a method in several different ways:

### Using a type parameter defined in the class declaration

```
public class ArrayList<E> extends AbstractList<E> ... {
    public boolean add(E o)
```

You can use the "E" here ONLY because it's already been defined as part of the class.

When you declare a type parameter for the class, you can simply use that type any place that you'd use a *real* class or interface type. The type declared in the method argument is essentially replaced with the type you use when you instantiate the class.

### Using a type parameter that was NOT defined in the class declaration

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

Here we can use <T> because we declared "T" earlier in the method declaration.

If the class itself doesn't use a type parameter, you can still specify one for a method, by declaring it in a really unusual (but available) space—*before the return type*. This method says that T can be "any type of Animal".


# Here's where it gets weird...

*This:*

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

*Is NOT the same as this:*

```
public void takeThing(ArrayList<Animal> list)
```

Both are legal, but they're *different*!

The first one, where **<T extends Animal>** is part of the method declaration, means that any ArrayList declared of a type that is Animal, or one of Animal's subtypes (like Dog or Cat), is legal. So you could invoke the top method using an ArrayList<Dog>, ArrayList<Cat>, or ArrayList<Animal>.

But... the one on the bottom, where the method argument is (ArrayList<Animal> list) means that *only* an ArrayList<Animal> is legal. In other words, while the first version takes an ArrayList of any type that is a type of Animal (Animal, Dog, Cat, etc.), the second version takes *only* an ArrayList of type Animal. Not ArrayList<Dog>, or ArrayList<Cat> but only ArrayList<Animal>.

And yes, it does appear to violate the point of polymorphism. but it will become clear when we revisit this in detail at the end of the chapter. For now, remember that we're only looking at this because we're still trying to figure out how to sort() that SongList, and that led us into looking at the API for the sort() method, which had this strange generic type declaration.

***For now, all you need to know is that the syntax of the top version is legal, and that it means you can pass in a ArrayList object instantiated as Animal or any Animal subtype.***

And now back to our sort() method...


**Remember where we were...**

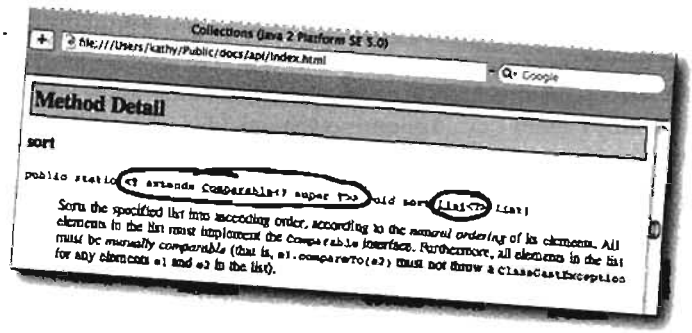
```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                           ^
1 error
```

```
import java.util.*;
import java.io.*;

public class Jukebox3 {
    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

This is where it breaks! It worked fine when passed in an ArrayList<String>, but as soon as we tried to sort an ArrayList<Song>, it failed.

# Revisiting the sort() method

So here we are, trying to read the sort() method docs to find out why it was OK to sort a list of Strings, but not a list of Song objects. And it looks like the answer is...




**The sort() method can take only lists of Comparable objects.**

**Song is NOT a subtype of Comparable, so you cannot sort() the list of Songs.**

***At least not yet...***

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

This says "Whatever 'T' is must be of type Comparable."

(Ignore this part for now. But if you can't, it just means that the type parameter for Comparable must be of type T or one of T's supertypes.)

You can pass in only a List (or subtype of list, like ArrayList) that uses a parameterized type that "extends Comparable".

Um... I just checked the docs for String, and String doesn't EXTEND Comparable--it IMPLEMENTS it. **Comparable** is an **interface**. So it's nonsense to say <T ***extends*** Comparable>.

```
public final class String extends Object implements Serializable,
                                                     Comparable<String>, CharSequence
```

# In generics, "extends" means "extends or implements"

The Java engineers had to give you a way to put a constraint on a parameterized type, so that you can restrict it to, say, only subclasses of Animal. But you also need to constrain a type to allow only classes that implement a particular interface. So here's a situation where we need one kind of syntax to work for both situations—inheritance and implementation. In other words, that works for both *extends* and *implements*.

And the winning word was... *extends*. But it really means "is-a", and works regardless of whether the type on the right is an interface or a class.

In generics, the keyword "extends" really means "is-a", and works for BOTH classes and interfaces.

Comparable is an interface, so this REALLY reads, "T must be a type that implements the Comparable interface".

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

It doesn't matter whether the thing on the right is a class or interface... you still say "extends".

**Q: Why didn't they just make a new keyword, "is"?**

A: Adding a new keyword to the language is a REALLY big deal because it risks breaking Java code you wrote in an earlier version. Think about it—you might be using a variable "is" (which we do use in this book to represent input streams). And since you're not allowed to use keywords as identifiers in your code, that means any earlier code that used the keyword *before* it was a reserved word, would break. So whenever there's a chance for the Sun engineers to reuse an existing keyword, as they did here with "extends", they'll usually choose that. But sometimes they don't have a choice...

A few (very few) new keywords *have* been added to the language, such as ***assert*** in Java 1.4 and ***enum*** in Java 5.0 (we look at enum in the appendix). And this does break people's code, however you sometimes have the option of compiling and running a *newer* version of Java so that it behaves as though it were an older one. You do this by passing a special flag to the compiler or JVM at the command-line, that says, "Yeah, yeah, I KNOW this is Java 1.4, but please pretend it's really 1.3, because I'm using a variable in my code named *assert* that I wrote back when you guys said it would OK!#$%".

(To see if you have a flag available, type *javac* (for the compiler) or *java* (for the JVM) at the command-line, without anything else after it, and you should see a list of available options. You'll learn more about these flags in the chapter on deployment.)

# Finally we know what's wrong...
## The Song class needs to implement Comparable

We can pass the ArrayList<Song> to the sort() method only if the Song class implements Comparable, since that's the way the sort() method was declared. A quick check of the API docs shows the Comparable interface is really simple, with only one method to implement:

Java.lang.Comparable

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

And the method documentation for compareTo() says

```
Returns:
a negative integer, zero, or a
positive integer as this object
is less than, equal to, or greater
than the specified object.
```

It looks like the compareTo() method will be called on one Song object, passing that Song a reference to a different Song. The Song running the compareTo() method has to figure out if the Song it was passed should be sorted higher, lower, or the same in the list.

Your big job now is to decide what makes one song greater than another, and then implement the compareTo() method to reflect that. A negative number (any negative number) means the Song you were passed is greater than the Song running the method. Returning a positive number says that the Song running the method is greater than the Song passed to the compareTo() method. Returning zero means the Songs are equal (at least for the purpose of sorting... it doesn't necessarily mean they're the same object). You might, for example, have two Songs with the same title.

(Which brings up a whole different can of worms we'll look at later...)

**The big question is: what makes *one* song less than, equal to, or greater than *another* song?**

**You can't implement the Comparable interface until you make that decision.**

### Sharpen your pencil

Write in your idea and pseudo code (or better, REAL code) for implementing the compareTo() method in a way that will sort() the Song objects by title.

Hint: If you're on the right track, it should take less than 3 lines of code!

# The new, improved, comparable Song class

We decided we want to sort by title, so we implement the compareTo() method to compare the title of the Song passed to the method against the title of the song on which the compareTo() method was invoked. In other words, the song running the method has to decide how its title compares to the title of the method parameter.

Hmmm... we know that the String class must know about alphabetical order, because the sort() method worked on a list of Strings. We know String has a compareTo() method, so why not just call it? That way, we can simply let one title String compare itself to another, and we don't have to write the comparing/alphabetizing algorithm!

Usually these match...we're specifying the type that the implementing class can be compared against.

This means that Song objects can be compared to other Song objects, for the purpose of sorting.

```
class Song implements Comparable<Song> {
   String title;
   String artist;
   String rating;
   String bpm;

   public int compareTo(Song s) {
       return title.compareTo(s.getTitle());
   }

   Song(String t, String a, String r, String b) {
      title = t;
      artist = a;
      rating = r;
      bpm = b;
   }

   public String getTitle() {
       return title;
   }

   public String getArtist() {
      return artist;
   }

   public String getRating() {
      return rating;
   }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
     return title;
  }
}
```

The sort() method sends a Song to compareTo() to see how that Song compares to the Song on which the method was invoked.

Simple! We just pass the work on to the title String objects, since we know Strings have a compareTo() method.

This time it worked. It prints the list, then calls sort which puts the Songs in alphabetical order by title.

```
File Edit Window Help Ambient
%java Jukebox3

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

# We can sort the list, but...

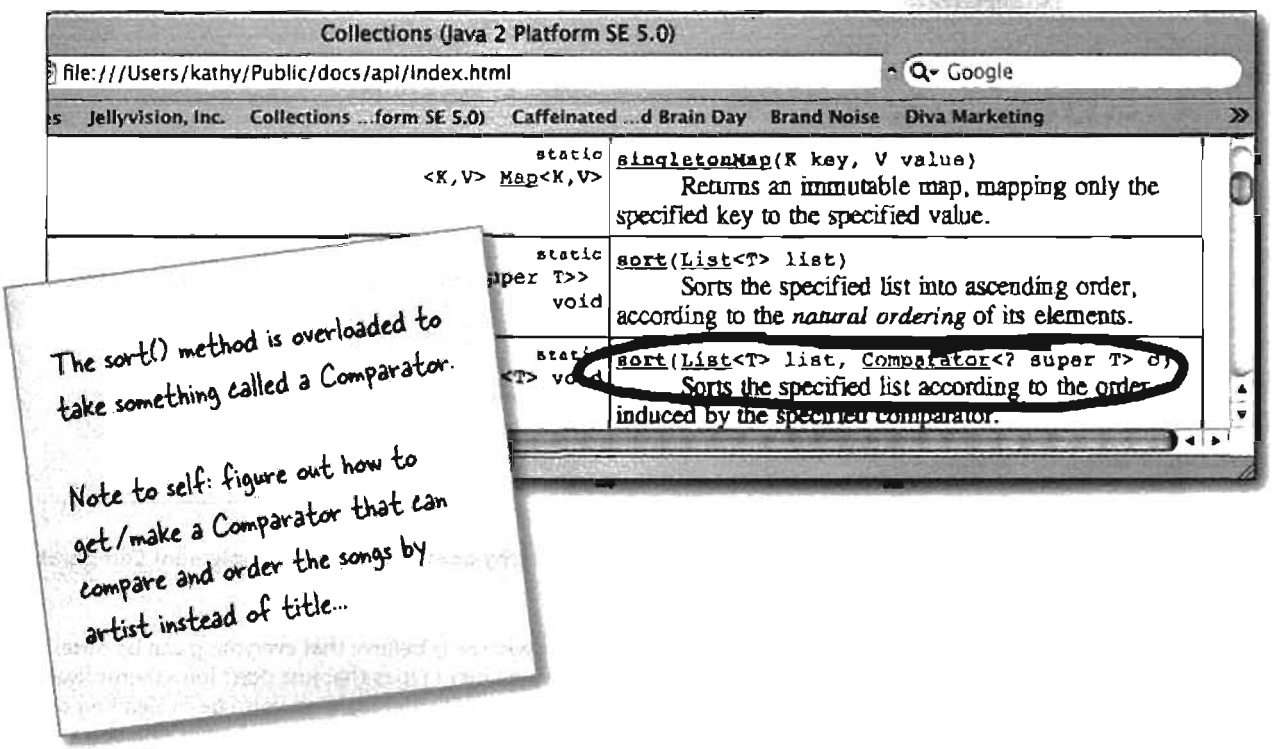
There's a new problem—Lou wants two different views of the song list, one by song title and one by artist!

But when you make a collection element comparable (by having it implement Comparable), you get only one chance to implement the compareTo() method. So what can you do?

The horrible way would be to use a flag variable in the Song class, and then do an *if* test in compareTo() and give a different result depending on whether the flag is set to use title or artist for the comparison.

But that's an awful and brittle solution, and there's something much better. Something built into the API for just this purpose—when you want to sort the same thing in more than one way.


**Look at the Collections class API again. There's a second sort() method—and it takes a Comparator.**



# Using a custom Comparator

An element in a list can compare *itself* to another of its own type in only one way, using its compareTo() method. But a Comparator is external to the element type you're comparing—it's a separate class. So you can make as many of these as you like! Want to compare songs by artist? Make an ArtistComparator. Sort by beats per minute? Make a BPMComparator.

Then all you need to do is call the overloaded sort() method that takes the List and the Comparator that will help the sort() method put things in order.

The sort() method that takes a Comparator will use the Comparator instead of the element's own compareTo() method, when it puts the elements in order. In other words, if your sort() method gets a Comparator, it won't even *call* the compareTo() method of the elements in the list. The sort() method will instead invoke the **compare**() method on the Comparator.

So, the rules are:

- **Invoking the one-argument sort(List o) method means the list element's compareTo() method determines the order. So the elements in the list MUST implement the Comparable interface.**

- **Invoking sort(List o, Comparator c) means the list element's compareTo() method will NOT be called, and the Comparator's compare() method will be used instead. That means the elements in the list do NOT need to implement the Comparable interface.**

java.util.Comparator

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

If you pass a Comparator to the sort() method, the sort order is determined by the Comparator rather than the element's own compareTo() method.

---

**Q: So does this mean that if you have a class that doesn't implement Comparable, and you don't have the source code, you could still put the things in order by creating a Comparator?**

A: That's right. The other option (if it's possible) would be to subclass the element and make the subclass implement Comparable.

**Q: But why doesn't *every* class implement Comparable?**

A: Do you really believe that *everything* can be ordered? If you have element types that just don't lend themselves to any kind of natural ordering, then you'd be misleading other programmers if you implement Comparable. And you aren't taking a huge risk by not implementing Comparable, since a programmer can compare anything in any way that he chooses using his own custom Comparator.

# Updating the Jukebox to use a Comparator

We did three new things in this code:

1) Created an inner class that implements Comparator (and thus the *compare()* method that does the work previously done by *compareTo()*).

2) Made an instance of the Comparator inner class.

3) Called the overloaded sort() method, giving it both the song list and the instance of the Comparator inner class.

Note: we also updated the Song class toString() method to print both the song title and the artist. (It prints *title: artist* regardless of how the list is sorted.)

```
import java.util.*;
import java.io.*;

public class Jukebox5 {
   ArrayList<Song> songList = new ArrayList<Song>();
   public static void main(String[] args) {
      new Jukebox5().go();
   }

   class ArtistCompare implements Comparator<Song> {
     public int compare(Song one, Song two) {
         return one.getArtist().compareTo(two.getArtist());
     }
   }

   public void go() {
     getSongs();
     System.out.println(songList);
     Collections.sort(songList);
     System.out.println(songList);

     ArtistCompare  artistCompare = new ArtistCompare();
     Collections.sort(songList, artistCompare);

     System.out.println(songList);
   }

   void getSongs() {
       // I/O code here
   }

   void addSong(String lineToParse) {
       // parse line and add to song list
   }
}
```

Create a new inner class that implements Comparator (note that its type parameter matches the type we're going to compare—in this case Song objects.)

This becomes a String (the artist)

We're letting the String variables (for artist) do the actual comparison, since Strings already know how to alphabetize themselves.

Make an instance of the Comparator inner class.

Invoke sort(), passing it the list and a reference to the new custom Comparator object

Note: we've made sort-by-title the default sort, by keeping the compareTo() method in Song use the titles. But another way to design this would be to implement both the title sorting and artist sorting as inner Comparator classes, and not have Song implement Comparable at all. That means we'd always use the two-arg version of Collections.sort().

## collections exercise


```
import _____________________;

public class SortMountains {

  LinkedList_______________ mtn = new LinkedList_____________();

  class NameCompare _____________________________________ {
    public int compare(Mountain one, Mountain two) {

      return _____________________________;
    }
  }
  class HeightCompare _____________________________ {
    public int compare(Mountain one, Mountain two) {

      return (_____________________________);
    }
  }
  public static void main(String [] args) {
    new SortMountain().go();
  }
  public void go() {
    mtn.add(new Mountain("Longs", 14255));
    mtn.add(new Mountain("Elbert", 14433));
    mtn.add(new Mountain("Maroon", 14156));
    mtn.add(new Mountain("Castle", 14265));

    System.out.println("as entered:\n" + mtn);
    NameCompare nc = new NameCompare();

    _____________________________;
    System.out.println("by name:\n" + mtn);
    HeightCompare hc = new HeightCompare();

    _____________________________;
    System.out.println("by height:\n" + mtn);
  }
}

class Mountain {

  _____________________;
  _____________________;

  _____________________ {
    _____________________;
    _____________________;
  }
  _____________________ {
    _____________________________;
  }
}
```

## Reverse Engineer

Assume this code exists in a single file. Your job is to fill in the blanks so the the program will create the output shown.

Note: answers are at the end of the chapter.

**Output:**

```
File  Edit  Window  Help  ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```


## Fill-in-the-blanks

For each of the questions below, fill in the blank with one of the words from the "possible answers" list, to correctly answer the question. Answers are at the end of the chapter.

**Possible Answers:**

**Comparator,**

**Comparable,**

**compareTo( ),**

**compare( ),**

**yes,**

**no**

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

1. What must the class of the objects stored in `myArrayList` implement? _____

2. What method must the class of the objects stored in `myArrayList` implement? _____

3. Can the class of the objects stored in `myArrayList` implement both `Comparator` AND `Comparable`? _____

Given the following compilable statement:

```
Collections.sort(myArrayList, myCompare);
```

4. Can the class of the objects stored in `myArrayList` implement `Comparable`? _____

5. Can the class of the objects stored in `myArrayList` implement `Comparator`? _____

6. Must the class of the objects stored in `myArrayList` implement `Comparable`? _____

7. Must the class of the objects stored in `myArrayList` implement `Comparator`? _____

8. What must the class of the `myCompare` object implement? _____

9. What method must the class of the `myCompare` object implement? _____

# Uh-oh. The sorting all works, but now we have duplicates...

The sorting works great, now we know how to sort on both *title* (using the Song object's compareTo() method) and *artist* (using the Comparator's compare() method). But there's a new problem we didn't notice with a test sample of the jukebox text file—***the sorted list contains duplicates.***

It appears that the diner jukebox just keeps writing to the file regardless of whether the same song has already been played (and thus written) to the text file. The SongListMore.txt jukebox text file is a complete record of every song that was played, and might contain the same song multiple times.

```
File Edit Window Help TooManyNotes
%java Jukebox4

[Pink Moon: Nick Drake, Somersault: Zero 7, Shiva Moon: Prem
Joshua, Circles: BT, Deep Channel: Afro Celts, Passenger:
Headmix, Listen: Tahiti 80, Listen: Tahiti 80, Listen: Tahiti
80, Circles: BT]

[Circles: BT, Circles: BT, Deep Channel: Afro Celts, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua,
Somersault: Zero 7]

[Deep Channel: Afro Celts, Circles: BT, Circles: BT, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Somersault:
Zero 7]
```

Before sorting.

After sorting using the Song's own compareTo() method (sort by title).

After sorting using the ArtistCompare Comparator (sort by artist name).

**SongListMore.txt**

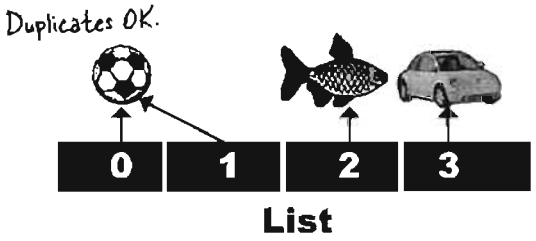
```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
```

The SongListMore text file now has duplicates in it, because the jukebox machine is writing every song played, in order. Somebody decided to play "Listen" three times in a row, followed by "Circles", a song that had been played earlier.

We can't change the way the text file is written because sometimes we're going to need all that information. We have to change the java code.

# We need a Set instead of a List

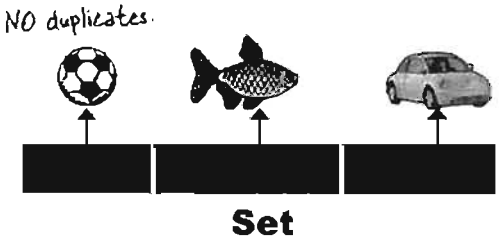
From the Collection API, we find three main interfaces, **List**, **Set**, and **Map**. ArrayList is a **List**, but it looks like *Set* is exactly what we need.




➤ **LIST** - when *sequence* matters

Collections that know about ***index position.***

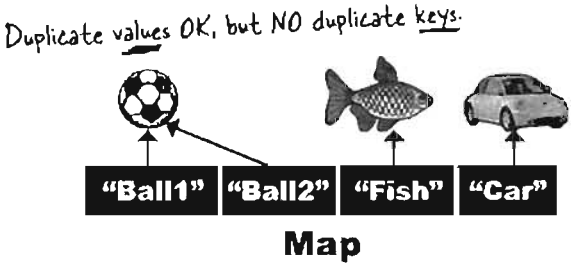
Lists know where something is in the list. You can have more than one element referencing the same object.




➤ **SET** - when *uniqueness* matters

Collections that ***do not allow duplicates.***

Sets know whether something is already in the collection. You can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal—we'll look at what object equality means in a moment).




➤ **MAP** - when *finding something by key* matters

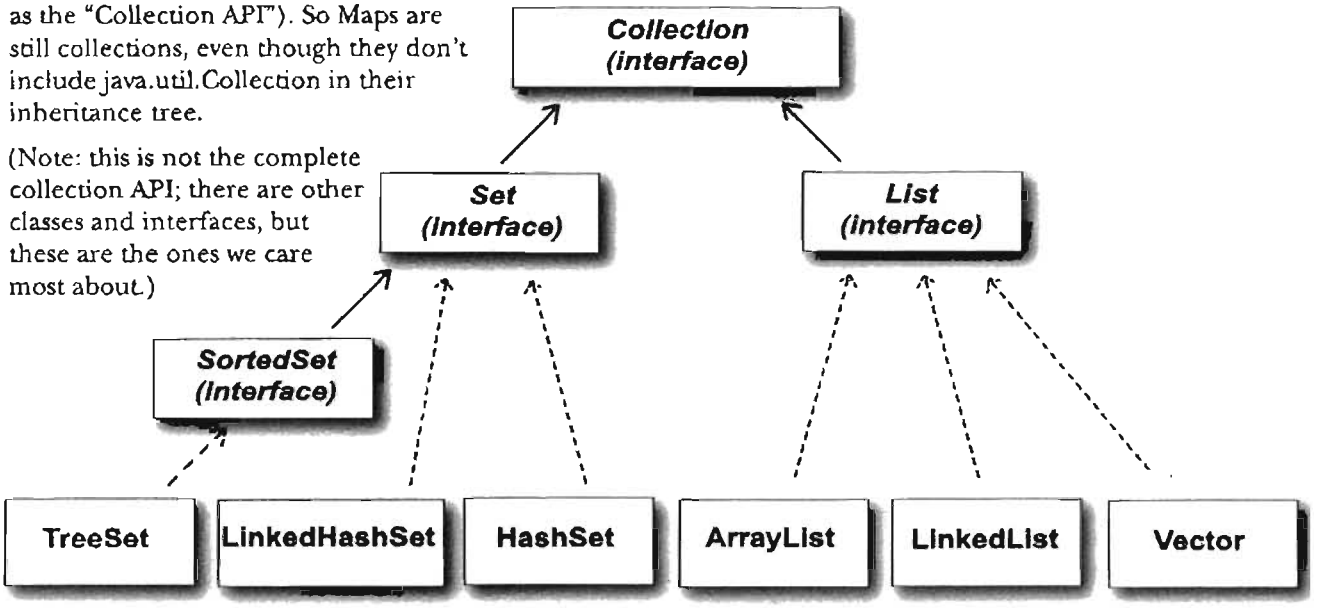
Collections that use ***key-value pairs.***

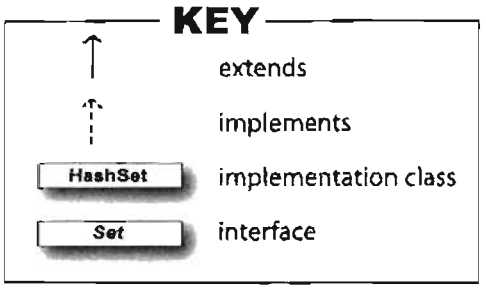
Maps know the value associated with a given key. You can have two keys that reference the same value, but you cannot have duplicate keys. Although keys are typically String names (so that you can make name/value property lists, for example), a key can be any object.

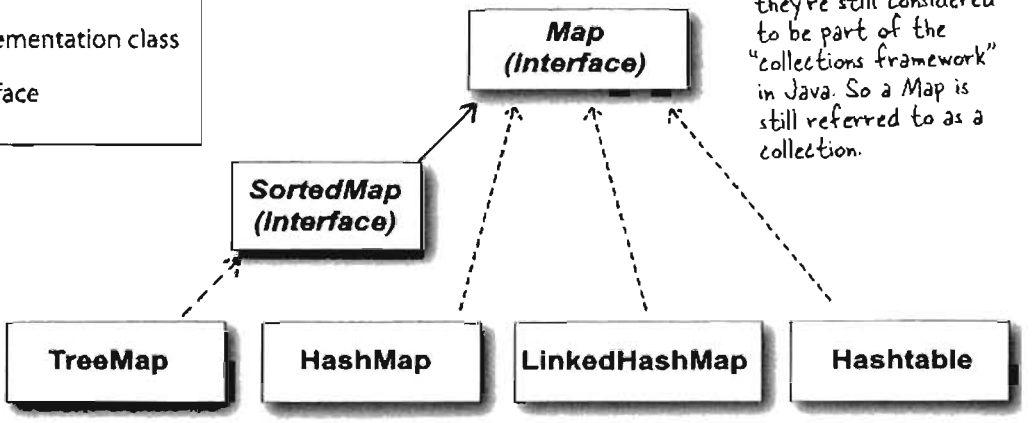
# The Collection API (part of it)

Notice that the Map interface doesn't actually extend the Collection interface, but Map is still considered part of the "Collection Framework" (also known as the "Collection API"). So Maps are still collections, even though they don't include java.util.Collection in their inheritance tree.

(Note: this is not the complete collection API; there are other classes and interfaces, but these are the ones we care most about.)




Maps don't extend from java.util.Collection, but they're still considered to be part of the "collections framework" in Java. So a Map is still referred to as a collection.

# Using a HashSet instead of ArrayList

We added on to the Jukebox to put the songs in a HashSet. (Note: we left out some of the Jukebox code, but you can copy it from earlier versions. And to make it easier to read the output, we went back to the earlier version of the Song's toString() method, so that it prints only the title instead of title *and* artist.)

```
import java.util.*;
import java.io.*;

public class Jukebox6 {
   ArrayList<Song> songList = new ArrayList<Song>();
   // main method etc.

   public void go() {
      getSongs();
      System.out.println(songList);
      Collections.sort(songList);
      System.out.println(songList);
      HashSet<Song> songSet = new HashSet<Song>();
      songSet.addAll(songList);
      System.out.println(songSet);
   }
   // getSongs() and addSong() methods
}
```

We didn't change getSongs(), so it still puts the songs in an ArrayList

Here we create a new HashSet parameterized to hold Songs

HashSet has a simple addAll() method that can take another collection and use it to populate the HashSet. It's the same as if we added each song one at a time (except much simpler).

```
File Edit Window Help GetBetterMusic
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles, Deep Channel,
Passenger, Listen, Listen, Listen, Circles]

[Circles, Circles, Deep Channel, Listen, Listen, Listen,
Passenger, Pink Moon, Shiva Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles, Listen, Deep Channel,
Passenger, Circles, Listen, Somersault]
```

Before sorting the ArrayList

After sorting the ArrayList (by title).

After putting it into a HashSet, and printing the HashSet (we didn't call sort() again).

The Set didn't help!!
We still have all the duplicates!

(And it lost its sort order when we put the list into a HashSet, but we'll worry about that one later...)

# What makes two objects equal?

**If two objects *foo* and *bar* are equal, *foo.equals(bar)* must be *true*, and both *foo* and *bar* must return the same value from *hashCode()*. For a Set to treat two objects as duplicates, you must override the hashCode() and equals() methods inherited from class Object, so that you can make two different objects be viewed as equal.**

First, we have to ask—what makes two Song references duplicates? They must be considered *equal*. Is it simply two references to the very same object, or is it two separate objects that both have the same *title*?

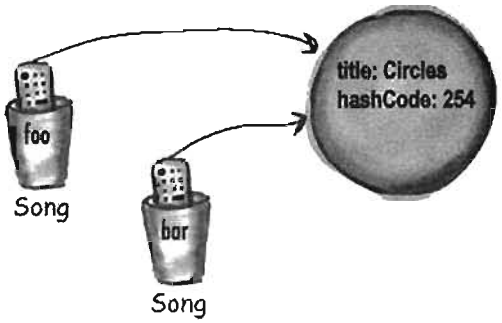
This brings up a key issue: *reference* equality vs. *object* equality.

## ➤ Reference equality

**Two references, one object on the heap.**

Two references that refer to the same object on the heap are equal. Period. If you call the **hashCode()** method on both references, you'll get the same result. If you don't override the hashCode() method, the default behavior (remember, you inherited this from class Object) is that each object will get a unique number (most versions of Java assign a hashcode based on the object's memory address on the heap, so no two objects will have the same hashcode).

If you want to know if two *references* are really referring to the same object, use the == operator, which (remember) compares the bits in the variables. If both references point to the same object, the bits will be identical.



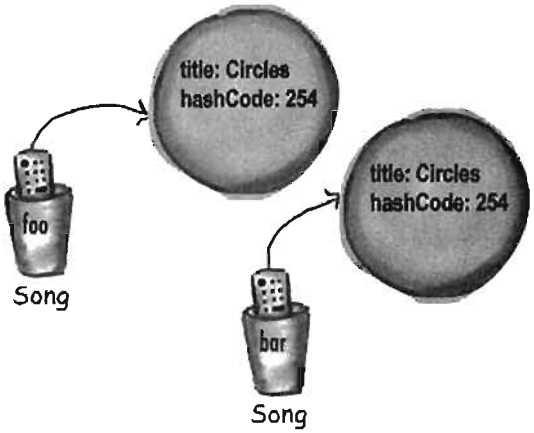

```
if (foo == bar) {
   // both references are referring
   // to the same object on the heap
}
```

## ➤ Object equality

**Two references, two objects on the heap, but the objects are considered *meaningfully equivalent*.**

If you want to treat two different Song objects as equal (for example if you decided that two Songs are the same if they have matching *title* variables), you must override *both* the **hashCode()** and **equals()** methods inherited from class Object.

As we said above, if you *don't* override hashCode(), the default behavior (from Object) is to give each object a unique hashcode value. So you must override hashCode() to be sure that two equivalent objects return the same hashcode. But you must also override equals() so that if you call it on *either* object, passing in the other object, always returns ***true***.




```
if (foo.equals(bar) && foo.hashCode() == bar.hashCode()) {
   // both references are referring to either a
   // a single object, or to two objects that are equal
}
```

# How a HashSet checks for duplicates: hashCode() and equals()

When you put an object into a Hashset, it uses the object's hashcode value to determine where to put the object in the Set. But it also compares the object's hashcode to the hashcode of all the other objects in the HashSet, and if there's no matching hashcode, the HashSet assumes that this new object is not a duplicate.

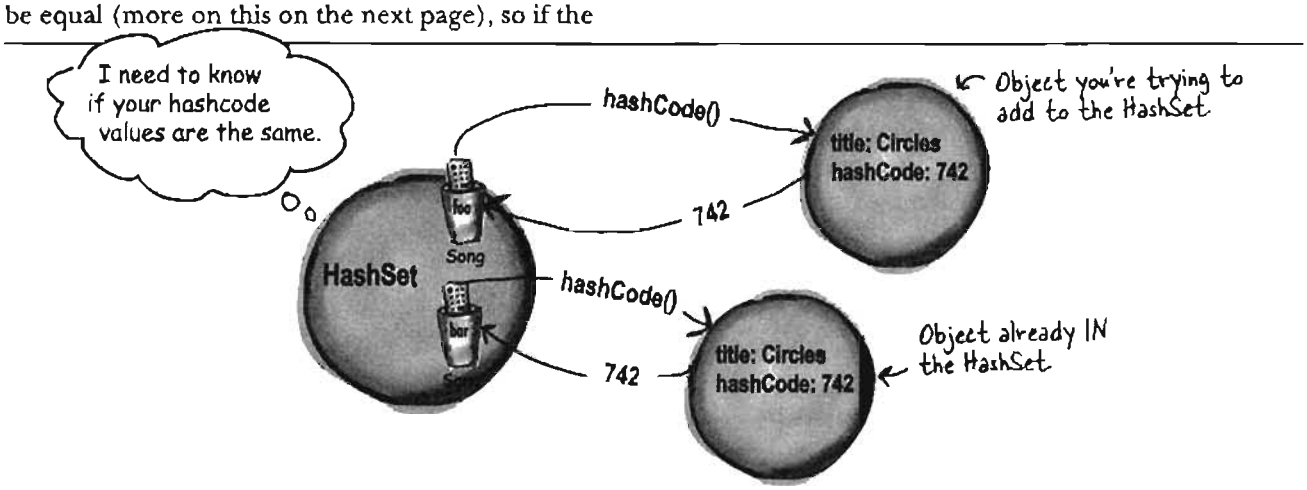
In other words, if the hashcodes are different, the HashSet assumes there's no way the objects can be equal!

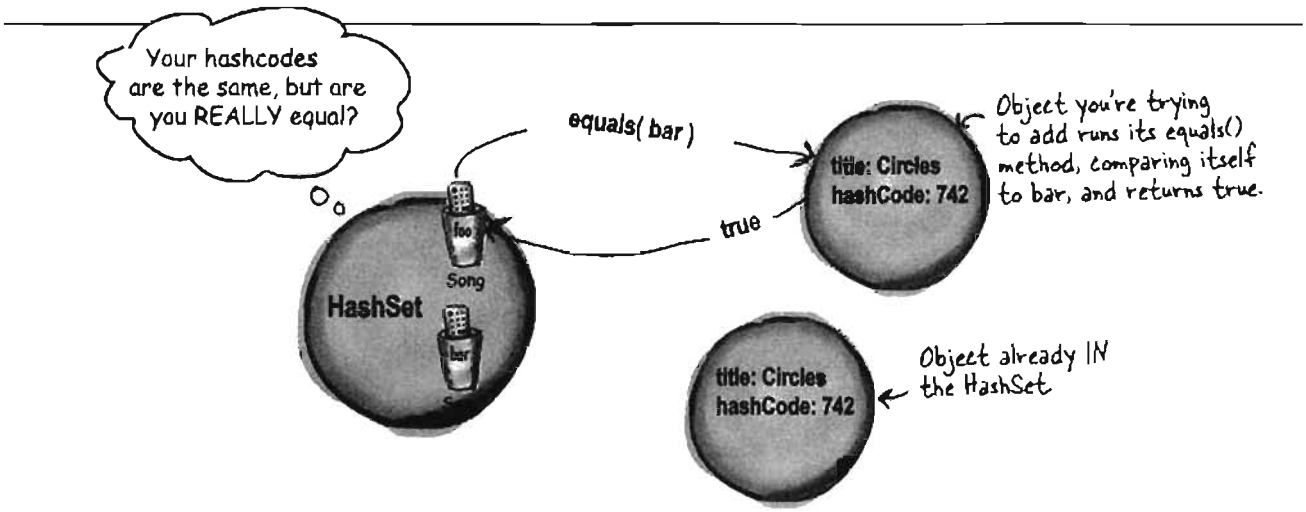
So you must override hashCode() to make sure the objects have the same value.

But two objects with the same hashCode() might *not* be equal (more on this on the next page), so if the HashSet finds a matching hashcode for two objects—one you're inserting and one already in the set—the HashSet will then call one of the object's equals() methods to see if these hashcode-matched objects really *are* equal.

And if they're equal, the HashSet knows that the object you're attempting to add is a duplicate of something in the Set, so the add doesn't happen.

You don't get an exception, but the HashSet's add() method returns a boolean to tell you (if you care) whether the new object was added. So if the add() method returns *false*, you know the new object was a duplicate of something already in the set.



# The Song class with overridden hashCode() and equals()

```
class Song implements Comparable<Song>{
   String title;
   String artist;
   String rating;
   String bpm;

  public boolean equals(Object aSong) {
     Song s = (Song) aSong;
     return getTitle().equals(s.getTitle());
  }

  public int hashCode() {
      return title.hashCode();
  }

   public int compareTo(Song s) {
      return title.compareTo(s.getTitle());
   }

   Song(String t, String a, String r, String b) {
      title = t;
      artist = a;
      rating = r;
      bpm = b;
   }

   public String getTitle() {
       return title;
   }

   public String getArtist() {
      return artist;
   }

   public String getRating() {
      return rating;
   }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
     return title;
  }
}
```

The HashSet (or anyone else calling this method) sends it another Song.

The GREAT news is that title is a String, and Strings have an overridden equals() method. So all we have to do is ask one title if it's equal to the other song's title.

Same deal here... the String class has an overridden hashCode() method, so you can just return the result of calling hashCode() on the title. Notice how hashCode() and equals() are using the SAME instance variable.

Now it works! No duplicates when we print out the HashSet. But we didn't call sort() again, and when we put the ArrayList into the HashSet, the HashSet didn't preserve the sort order.

```
File Edit Window Help RebootWindows
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles,
Deep Channel, Passenger, Listen, Listen,
Listen, Circles]

[Circles, Circles, Deep Channel, Listen,
Listen, Listen, Passenger, Pink Moon, Shiva
Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles,
Deep Channel, Passenger, Somersault]
```

## Java Object Law For HashCode() and equals()

**The API docs for class Object state the rules you MUST follow:**

**If two objects are equal, they MUST have matching hashcodes.**

**If two objects are equal, calling equals() on either object MUST return true. In other words, if (a.equals(b)) then (b.equals(a)).**

**If two objects have the same hashcode value, they are NOT required to be equal. But if they're equal, they MUST have the same hashcode value.**

**So, if you override equals(), you MUST override hashCode().**

**The default behavior of hashCode() is to generate a unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects of that type can EVER be considered equal.**

**The default behavior of equals() is to do an == comparison. In other words, to test whether the two references refer to a single object on the heap. So if you don't override equals() in a class, no two objects can EVER be considered equal since references to two different objects will always contain a different bit pattern.**

***a.equals(b)* must also mean that *a.hashCode() == b.hashCode()***

**But *a.hashCode() == b.hashCode()* does NOT have to mean *a.equals(b)***

## there are no Dumb Questions

**Q: How come hashcodes can be the same even if objects aren't equal?**

A: HashSets use hashcodes to store the elements in a way that makes it much faster to access. If you try to find an object in an ArrayList by giving the ArrayList a copy of the object (as opposed to an index value), the ArrayList has to start searching from the beginning, looking at each element in the list to see if it matches. But a HashSet can find an object much more quickly, because it uses the hashcode as a kind of label on the "bucket" where it stored the element. So if you say, "I want you to find an object in the set that's exactly like this one..." the HashSet gets the hashcode value from the copy of the Song you give it (say, 742), and then the HashSet says, "Oh, I know exactly where the object with hashcode #742 is stored...", and it goes right to the #742 bucket.

This isn't the whole story you get in a computer science class, but it's enough for you to use HashSets effectively. In reality, developing a good hashcode algorithm is the subject of many a PhD thesis, and more than we want to cover in this book.

The point is that hashcodes can be the same without necessarily guaranteeing that the objects are equal, because the "hashing algorithm" used in the hashCode() method might happen to return the same value for multiple objects. And yes, that means that multiple objects would all land in the same bucket in the HashSet (because each bucket represents a single hashcode value), but that's not the end of the world. It might mean that the HashSet is just a little less efficient (or that it's filled with an extremely large number of elements), but if the HashSet finds more than one object in the same hashcode bucket, the HashSet will simply use the equals() method to see if there's a perfect match. In other words, hashcode values are sometimes used to narrow down the search, but to find the one exact match, the HashSet still has to take all the objects in that one bucket (the bucket for all objects with the same hashcode) and then call equals() on them to see if the object it's looking for is in that bucket.

# And if we want the set to stay sorted, we've got TreeSet

TreeSet is similar to HashSet in that it prevents duplicates. But it also *keeps* the list sorted. It works just like the sort() method in that if you make a TreeSet using the set's no-arg constructor, the TreeSet uses each object's compareTo() method for the sort. But you have the option of passing a Comparator to the TreeSet constructor, to have the TreeSet use that instead. The downside to TreeSet is that if you don't *need* sorting, you're still paying for it with a small performance hit. But you'll probably find that the hit is almost impossible to notice for most apps.

```
import java.util.*;
import java.io.*;
public class Jukebox8 {
   ArrayList<Song> songList = new ArrayList<Song>();
   int val;

   public static void main(String[] args) {
      new Jukebox8().go();
   }

   public void go() {
      getSongs();
      System.out.println(songList);
      Collections.sort(songList);
      System.out.println(songList);
      TreeSet<Song> songSet = new TreeSet<Song>();
      songSet.addAll(songList);
      System.out.println(songSet);
   }

   void getSongs() {
      try {
         File file = new File("SongListMore.txt");
         BufferedReader reader = new BufferedReader(new FileReader(file));
         String line =  null;
         while ((line= reader.readLine()) != null) {
              addSong(line);
         }

      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }

   void addSong(String lineToParse) {
      String[] tokens = lineToParse.split("/");
      Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
      songList.add(nextSong);
   }
}
```

Instantiate a TreeSet instead of HashSet. Calling the no-arg TreeSet constructor means the set will use the Song object's compareTo() method for the sort.

(We could have passed in a Comparator.)

We can add all the songs from the HashSet using addAll(). (Or we could have added the songs individually using songSet.add() just the way we added songs to the ArrayList.)

# What you MUST know about TreeSet...

TreeSet looks easy, but make sure you really understand what you need to do to use it. We thought it was so important that we made it an exercise so you'd *have* to think about it. Do NOT turn the page until you've done this. *We mean it.*


Look at this code. Read it carefully, then answer the questions below. (Note: there are no syntax errors in this code.)

```
import java.util.*;

public class TestTree {
   public static void main (String[] args) {
      new TestTree().go();
   }

   public void go() {
     Book b1 = new Book("How Cats Work");
     Book b2 = new Book("Remix your Body");
     Book b3 = new Book("Finding Emo");

     TreeSet<Book> tree = new TreeSet<Book>();
     tree.add(b1);
     tree.add(b2);
     tree.add(b3);
     System.out.println(tree);
   }
}

class Book {
   String title;
   public Book(String t) {
     title = t;
   }
}
```

1). What is the result when you compile this code?

2). If it compiles, what is the result when you run the TestTree class?

3). If there is a problem (either compile-time or runtime) with this code, how would you fix it?

# TreeSet elements MUST be comparable

TreeSet can't read the programmer's mind to figure out how the object's should be sorted. You have to tell the TreeSet *how*.

## To use a TreeSet, one of these things must be true:

### ➤ The elements in the list must be of a type that implements *Comparable*

The Book class on the previous page didn't implement Comparable, so it wouldn't work at runtime. Think about it, the poor TreeSet's sole purpose in life is to keep your elements sorted, and once again—it had no idea how to sort Book objects! It doesn't fail at compile-time, because the TreeSet add() method doesn't take a Comparable type,The TreeSet add() method takes whatever type you used when you created the TreeSet. In other words, if you say new TreeSet<Book>() the add() method is essentially add(Book). And there's no requirement that the Book class implement Comparable! But it fails at runtime when you add the second element to the set. That's the first time the set tries to call one of the object's compareTo() methods and... can't.

```
class Book implements Comparable {
   String title;
   public Book(String t) {
     title = t;
   }
   public int compareTo(Object b) {
     Book book = (Book) b;
     return (title.compareTo(book.title));
   }
}
```

**OR**

### ➤ You use the TreeSet's overloaded constructor that takes a *Comparator*

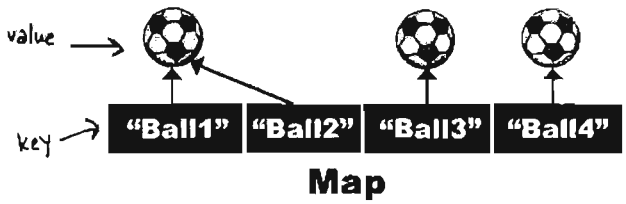
TreeSet works a lot like the sort() method—you have a choice of using the element's compareTo() method, assuming the element type implemented the Comparable interface, OR you can use a custom Comparator that knows how to sort the elements in the set. To use a custom Comparator, you call the TreeSet constructor that takes a Comparator.

```
public class BookCompare implements Comparator<Book> {
   public int compare(Book one, Book two) {
      return (one.title.compareTo(two.title));
   }
}

class Test {
     public void go() {
     Book b1 = new Book("How Cats Work");
     Book b2 = new Book("Remix your Body");
     Book b3 = new Book("Finding Emo");
     BookCompare bCompare = new BookCompare();
     TreeSet<Book> tree = new TreeSet<Book>(bCompare);
     tree.add(new Book("How Cats Work");
     tree.add(new Book("Finding Emo");
     tree.add(new Book("Remix your Body");
     System.out.println(tree);
   }
}
```

# We've seen Lists and Sets, now we'll use a Map

Lists and Sets are great, but sometimes a Map is the best collection (not Collection with a capital "C"—remember that Maps are part of Java collections but they don't implement the Collection interface).

Imagine you want a collection that acts like a property list, where you give it a name and it gives you back the value associated with that name. Although keys will often be Strings, they can be any Java object (or, through autoboxing, a primitive).




**Each element in a Map is actually TWO objects—a *key* and a *value*. You can have duplicate *values*, but NOT duplicate *keys*.**

## Map example

```
import java.util.*;

public class TestMap {

   public static void main(String[] args) {

     HashMap<String, Integer> scores = new HashMap<String, Integer>();

     scores.put("Kathy", 42);
     scores.put("Bert", 343);
     scores.put("Skyler", 420);

      System.out.println(scores);
      System.out.println(scores.get("Bert"));
   }
}
```

HashMap needs TWO type parameters—one for the key and one for the value.

Use put() instead of add(), and now of course it takes two arguments (key, value).

The get() method takes a key, and returns the value (in this case, an Integer).

```
File Edit Window Help WhereAmI
%java TestMap

{Skyler=420, Bert=343, Kathy=42}
343
```

When you print a Map, it gives you the key=value, in braces { } instead of the brackets [ ] you see when you print lists and sets.

# Finally, back to generics

Remember earlier in the chapter we talked about how methods that take arguments with generic types can be... *weird*. And we mean weird in the polymorphic sense. If things start to feel strange here, just keep going—it takes a few pages to really tell the whole story.

We'll start with a reminder on how *array* arguments work, polymorphically, and then look at doing the same thing with generic lists. The code below compiles and runs without errors:

**If a method argument is an *array* of Animals, it will also take an array of any Animal subtype.**

**In other words, if a method is declared as:**

***void foo(Animal[] a) { }***

**Assuming Dog extends Animal, you are free to call both:**

***foo(anAnimalArray);***
***foo(aDogArray);***

### Here's how it works with regular *arrays*:

```
import java.util.*;

public class TestGenerics1 {
   public static void main(String[] args) {
      new TestGenerics1().go();
   }

   public void go() {
      Animal[] animals = {new Dog(), new Cat(), new Dog()};
      Dog[] dogs = {new Dog(), new Dog(), new Dog()};
      takeAnimals(animals);
      takeAnimals(dogs);
   }

  public void takeAnimals(Animal[] animals) {
      for(Animal a: animals) {
        a.eat();
      }
   }
}
```

Declare and create an Animal array, that holds both dogs and cats.

Declare and create a Dog array, that holds only Dogs (the compiler won't let you put a Cat in).

Call takeAnimals(), using both array types as arguments...

The crucial point is that the takeAnimals() method can take an Animal[] or a Dog[], since Dog IS-A Animal. Polymorphism in action.

Remember, we can call ONLY the methods declared in type animal, since the animals parameter is of type Animal array, and we didn't do any casting. (What would we cast it to? That array might hold both Dogs and Cats.)

---

```
abstract class Animal {
   void eat() {
     System.out.println("animal eating");
   }
}
class Dog extends Animal {
   void bark() { }
}
class Cat extends Animal {
   void meow() { }
}
```

The simplified Animal class hierarchy.

# Using polymorphic arguments and generics

So we saw how the whole thing worked with arrays, but will it work the same way when we switch from an array to an ArrayList? Sounds reasonable, doesn't it?

First, let's try it with only the Animal ArrayList. We made just a few changes to the go() method:

### Passing in just ArrayList<Animal>

A simple change from Animal[] to ArrayList<Animal>.

```
public void go() {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog());
    animals.add(new Cat());
    animals.add(new Dog());

    takeAnimals(animals);
}
```

We have to add one at a time since there's no shortcut syntax like there is for array creation.

This is the same code, except now the "animals" variable refers to an ArrayList instead of array.

```
public void takeAnimals(ArrayList<Animal> animals) {
    for(Animal a: animals) {
    a.eat();
    }
}
```

The method now takes an ArrayList instead of an array, but everything else is the same. Remember, that for loop syntax works for both arrays and collections.

### Compiles and runs just fine

```
File Edit Window Help CatFoodIsBetter
%java TestGenerics2

animal eating
animal eating
animal eating
animal eating
animal eating
animal eating
```

# But will it work with ArrayList<Dog> ?

Because of polymorphism, the compiler let us pass a Dog array to a method with an Animal array argument. No problem. And an ArrayList<Animal> can be passed to a method with an ArrayList<Animal> argument. So the big question is, will the ArrayList<Animal> argument accept an ArrayList<Dog>? If it works with arrays, shouldn't it work here too?

### Passing in just ArrayList<Dog>

```
public void go() {
   ArrayList<Animal> animals = new ArrayList<Animal>();
   animals.add(new Dog());
   animals.add(new Cat());
   animals.add(new Dog());
   takeAnimals(animals);
   ArrayList<Dog> dogs = new ArrayList<Dog>();
   dogs.add(new Dog());
   dogs.add(new Dog());
   takeAnimals(dogs);
}

public void takeAnimals(ArrayList<Animal> animals) {
     for(Animal a: animals) {
     a.eat();
    }
 }
```

We know this line worked fine.

Make a Dog ArrayList and put a couple dogs in.

Will this work now that we changed from an array to an ArrayList?

### When we compile it:

```
File Edit Window Help CatsAreSmarter
%java TestGenerics3

TestGenerics3.java:21: takeAnimals(java.util.
ArrayList<Animal>) in TestGenerics3 cannot be applied to
(java.util.ArrayList<Dog>)
     takeAnimals(dogs);
     ^
1 error
```

It looked so right, but went so wrong...


## What could happen if it *were* allowed...

Imagine the compiler let you get away with that. It let you pass an ArrayList<Dog> to a method declared as:

```
public void takeAnimals(ArrayList<Animal> animals) {
   for(Animal a: animals) {
      a.eat();
   }
}
```

There's nothing in that method that *looks* harmful, right? After all, the whole point of polymorphism is that anything an Animal can do (in this case, the eat() method), a Dog can do as well. So what's the problem with having the method call eat() on each of the Dog references?

*Nothing.* Nothing at all.

There's nothing wrong with *that* code. But imagine *this* code instead:

```
public void takeAnimals(ArrayList<Animal> animals) {
   animals.add(new Cat());
}
```

← Yikes!! We just stuck a Cat in what might be a Dogs-only ArrayList

So that's the problem. There's certainly nothing wrong with adding a Cat to an ArrayList<Animal>, and that's the whole point of having an ArrayList of a supertype like Animal—so that you can put all types of animals in a single Animal ArrayList.

But if you passed a Dog ArrayList—one meant to hold ONLY Dogs—to this method that takes an Animal ArrayList, then suddenly you'd end up with a Cat in the Dog list. The compiler knows that if it lets you pass a Dog ArrayList into the method like that, someone could, at runtime, add a Cat to your Dog list. So instead, the compiler just won't let you take the risk.

***If you declare a method to take ArrayList<Animal> it can take ONLY an ArrayList<Animal>, not ArrayList<Dog> or ArrayList<Cat>.***


## Array types are checked again at runtime, but collection type checks happen only when you compile

Let's say you *do* add a Cat to an array declared as Dog[] (an array that was passed into a method argument declared as Animal[], which is a perfectly legal assignment for arrays).

```
public void go() {
   Dog[] dogs = {new Dog(), new Dog(), new Dog()};
   takeAnimals(dogs);
}

public void takeAnimals(Animal[] animals) {
   animals[0] = new Cat();
}
```

We put a new Cat into a Dog array. The compiler allowed it, because it knows that you might have passed a Cat array or Animal array to the method, so to the compiler it was possible that this was OK.

**It compiles, but when we run it:**

Whew! At least the JVM stopped it.

```
File Edit Window Help CatsAreSmarter
%java TestGenerics1
Exception in thread "main" java.lang.ArrayStoreException:
Cat
        at TestGenerics1.takeAnimals(TestGenerics1.java:16)
        at TestGenerics1.go(TestGenerics1.java:12)
        at TestGenerics1.main(TestGenerics1.java:5)
```

Wouldn't it be dreamy if there were a way to still use polymorphic collection types as method arguments, so that my veterinary program could take Dog lists and Cat lists? That way I could loop through the lists and call their immunize() method, but it would still have to be safe so that you couldn't add a Cat in to the Dog list. But I guess that's just a fantasy...

# Wildcards to the rescue

It looks unusual, but there *is* a way to create a method argument that can accept an ArrayList of any Animal subtype. The simplest way is to use a **wildcard**—added to the Java language explicitly for this reason.

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
    for(Animal a: animals) {
        a.eat();
    }
}
```

Remember, the keyword "extends" here means either extends OR implements depending on the type. So if you want to take an ArrayList of types that implement the Pet interface, you'd declare it as:

ArrayList<? extends Pet>

So now you're wondering, "What's the *difference*? Don't you have the same problem as before? The method above isn't doing anything dangerous—calling a method any Animal subtype is guaranteed to have—but can't someone still change this to add a Cat to the *animals* list, even though it's really an ArrayList<Dog>? And since it's not checked again at runtime, how is this any different from declaring it without the wildcard?"

And you'd be right for wondering. The answer is NO. When you use the wildcard <?> in your declaration, the compiler won't let you do anything that adds to the list!

**When you use a wildcard in your method argument, the compiler will STOP you from doing anything that could hurt the list referenced by the method parameter.**

**You can still invoke methods on the elements in the list, but you cannot add elements to the list.**

**In other words, you can do things *with* the list elements, but you can't put *new* things in the list. So you're safe at runtime, because the compiler won't let you do anything that might be horrible at runtime.**

**So, this is OK inside takeAnimals():**

```
for(Animal a: animals)
    a.eat();
}
```

**But THIS would not compile:**

```
animals.add(new Cat());
```

# Alternate syntax for doing the same thing

You probably remember that when we looked at the sort() method, it used a generic type, but with an unusual format where the type parameter was declared before the return type. It's just a different way of declaring the type parameter, but the results are the same:

**This:**

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

**Does the same thing as this:**

```
public void takeThing(ArrayList<? extends Animal> list)
```

## there are no Dumb Questions

**Q: If they both do the same thing, why would you use one over the other?**

A: It all depends on whether you want to use "T" somewhere else. For example, what if you want the method to have two arguments—both of which are lists of a type that extend Animal? In that case, it's more efficient to just declare the type parameter once:

```
public <T extends Animal> void takeThing(ArrayList<T> one, ArrayList<T> two)
```

Instead of typing:

```
public void takeThing(ArrayList<? extends Animal> one,
                      ArrayList<? extends Animal> two)
```


# BE the compiler, advanced

Your job is to play compiler and determine which of these statements would compile. But some of this code wasn't covered in the chapter, so you need to work out the answers based on what you DID learn, applying the "rules" to these new situations. In some cases, you might have to guess, but the point is to come up with a reasonable answer based on what you know so far.

(Note: assume that this code is within a legal class and method.)

**Compiles?**

- ☐ `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- ☐ `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- ☐ `List<Animal> list = new ArrayList<Animal>();`
- ☐ `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- ☐ `ArrayList<Animal> animals = dogs;`
- ☐ `List<Dog> dogList = dogs;`
- ☐ `ArrayList<Object> objects = new ArrayList<Object>();`
- ☐ `List<Object> objList = objects;`
- ☐ `ArrayList<Object> objs = new ArrayList<Dog>();`

## Solution to the "Reverse Engineer" sharpen exercise

```
import java.util.*;

public class SortMountains {

  LinkedList<Mountain> mtn = new LinkedList<Mountain>();

  class NameCompare implements Comparator <Mountain> {
    public int compare(Mountain one, Mountain two) {
      return one.name.compareTo(two.name);
    }
  }
  class HeightCompare implements Comparator <Mountain> {
    public int compare(Mountain one, Mountain two) {
      return (two.height - one.height);
    }
  }
  public static void main(String [] args) {
    new SortMountain().go();
  }
  public void go() {
    mtn.add(new Mountain("Longs", 14255));
    mtn.add(new Mountain("Elbert", 14433));
    mtn.add(new Mountain("Maroon", 14156));
    mtn.add(new Mountain("Castle", 14265));
    System.out.println("as entered:\n" + mtn);
    NameCompare nc = new NameCompare();
    Collections.sort(mtn, nc);
    System.out.println("by name:\n" + mtn);
    HeightCompare hc = new HeightCompare();
    Collections.sort(mtn, hc);
    System.out.println("by height:\n" + mtn);
  }
}

class Mountain {
  String name;
  int height;

  Mountain(String n, int h) {
    name = n;
    height = h;
  }
  public String toString( ) {
    return name + " " + height;
  }
```

Did you notice that the height list is in DESCENDING sequence? :)

**Output:**

```
File Edit Window Help ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```

**fill-in-the-blank** solution

## Exercise Solution

**Possible Answers:**

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

1. What must the class of the objects stored in `myArrayList` implement? **Comparable**
2. What method must the class of the objects stored in `myArrayList` implement? **compareTo( )**
3. Can the class of the objects stored in `myArrayList` implement both `Comparator` AND `Comparable`? **yes**

Given the following compilable statement:

```
Collections.sort(myArrayList, myCompare);
```

4. Can the class of the objects stored in `myArrayList` implement `Comparable`? **yes**
5. Can the class of the objects stored in `myArrayList` implement `Comparator`? **yes**
6. Must the class of the objects stored in `myArrayList` implement `Comparable`? **no**
7. Must the class of the objects stored in `myArrayList` implement `Comparator`? **no**
8. What must the class of the `myCompare` object implement? **Comparator**
9. What method must the class of the `myCompare` object implement? **compare( )**

# BE the compiler solution

**Compiles?**

- [ ] `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`
- [ ] `ArrayList<Animal> animals1 = new ArrayList<Dog>();`
- [x] `List<Animal> list = new ArrayList<Animal>();`
- [ ] `ArrayList<Dog> dogs = new ArrayList<Dog>();`
- [ ] `ArrayList<Animal> animals = dogs;`
- [x] `List<Dog> dogList = dogs;`
- [x] `ArrayList<Object> objects = new ArrayList<Object>();`
- [x] `List<Object> objList = objects;`
- [ ] `ArrayList<Object> objs = new ArrayList<Dog>();`

# Release Your Code

**It's time to let go.** You wrote your code. You tested your code. You refined your code. You told everyone you know that if you never saw a line of code again, that'd be fine. But in the end, you've created a work of art. The thing actually runs! But now what? *How* do you give it to end users? *What* exactly do you give to end users? What if you don't even know who your end users are? In these final two chapters, we'll explore how to organize, package, and deploy your Java code. We'll look at local, semi-local, and remote deployment options including executable jars, Java Web Start, RMI, and Servlets. In this chapter, we'll spend most of our time on organizing and packaging your code—things you'll need to know regardless of your ultimate deployment choice. In the final chapter, we'll finish with one of the coolest things you can do in Java. Relax. Releasing your code is not saying goodbye. There's always maintenance...

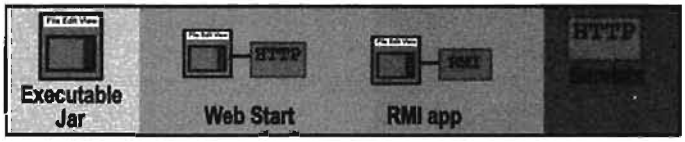
# Deploying your application

What exactly *is* a Java application? In other words, once you're done with development, what is it that you deliver? Chances are, your end-users don't have a system identical to yours. More importantly, they don't have your application. So now it's time to get your program in shape for deployment into The Outside World. In this chapter, we'll look at local deployments, including Executable Jars and the part-local/part-remote technology called Java Web Start. In the next chapter, we'll look at the more remote deployment options, including RMI and Servlets.

A Java program is a bunch of classes. That's the output of your development.

The real question is what to do with those classes when you're done.

### Deployment options




100% Local  Combination  100% Remote

1. **Local**
   The entire application runs on the end-user's computer, as a stand-alone, probably GUI, program, deployed as an executable JAR (we'll look at JAR in a few pages.)

2. **Combination of local and remote**
   The application is distributed with a client portion running on the user's local system, connected to a server where other parts of the application are running.

3. **Remote**
   The entire Java application runs on a server system, with the client accessing the system through some non-Java means, probably a web browser.

But before we really get into the whole deployment thing, let's take a step back and look at what happens when you've finished programming your app and you simply want to pull out the class files to give them to an end-user. What's really *in* that working directory?

## Brain Barbell

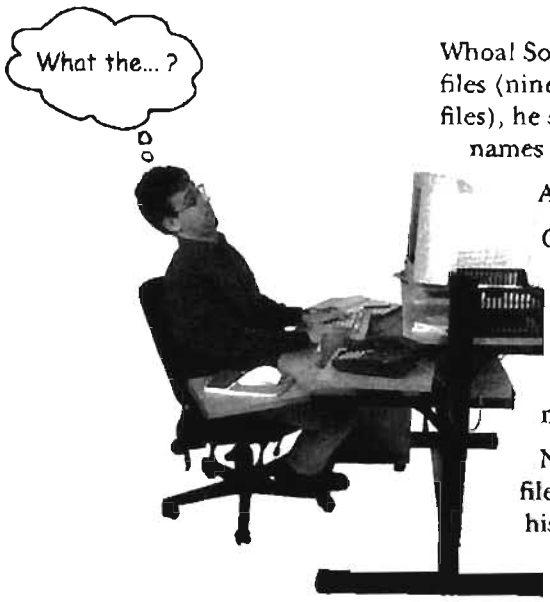
What are the advantages and disadvantages of delivering your Java program as a local, stand-alone application running on the end-user's computer?

What are the advantages and disadvantages of delivering your Java program as web-based system where the user interacts with a web browser, and the Java code runs as servlets on the server?


## Imagine this scenario...

Bob's happily at work on the final pieces of his cool new Java program. After weeks of being in the "I'm-just-one-compile-away" mode, this time he's really done. The program is a fairly sophisticated GUI app, but since the bulk of it is Swing code, he's made only nine classes of his own.

At last, it's time to deliver the program to the client. He figures all he has to do is copy the nine class files, since the client already has the Java API installed. He starts by doing an `ls` on the directory where all his files are...




Whoa! Something strange has happened. Instead of 18 files (nine source code files and nine compiled class files), he sees 31 files, many of which have very strange names like:

Account$FileListener.class

Chart$SaveListener.class

and on it goes. He had completely forgotten that the compiler has to generate class files for all those inner class GUI event listeners he made, and that's what all the strangely-named classes are.

Now he has to carefully extract all the class files he needs. If he leaves even one of them out, his program won't work. But it's tricky since he doesn't want to accidentally send the client one of his *source* code files, yet everything is in the same directory in one big mess.

# Separate source code and class files

A single directory with a pile of source code and class files is a mess. It turns out, Bob should have been organizing his files from the beginning, keeping the source code and compiled code separate. In other words, making sure his compiled class files didn't land in the same directory as his source code.

***The key is a combination of directory structure organization and the -d compiler option.***


There are dozens of ways you can organize your files, and your company might have a specific way they want you to do it. We recommend an organizational scheme that's become almost standard, though.

With this scheme, you create a project directory, and inside that you create a directory called **source** and a directory called **classes**. You start by saving your source code (.java files) into the **source** directory. Then the trick is to compile your code in such a way that the output (the .class files) ends up in the *classes* directory.

And there's a nice compiler flag, **-d**, that lets you do that.

## Compiling with the -d (directory) flag

```
%cd MyProject/source
%javac -d ../classes MyApp.java
```

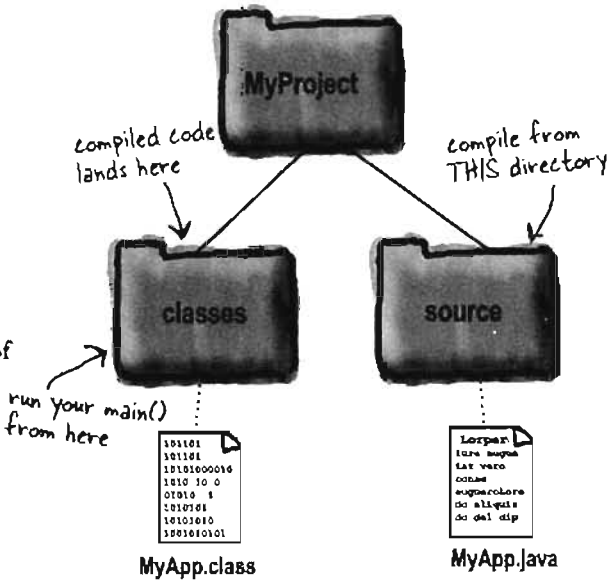
tells the compiler to put the compiled code (class files) into the "classes" directory that's one directory up and back down again from the current working directory.

the last thing is still the name of the java file to compile

By using the **-d** flag, *you* get to decide which *directory* the compiled code lands in, rather than accepting the default of class files landing in the same directory as the source code. To compile all the .java files in the source directory, use:

```
%javac -d ../classes *.java
```

*.java compiles ALL source files in the current directory




## Running your code

```
%cd MyProject/classes
%java Mini
```

run your program from the 'classes' directory.

(troubleshooting note: everything in this chapter assumes that the current working directory (i.e. the ".") is in your classpath. If you have explicitly set a classpath environment variable, be certain that it contains the ".")

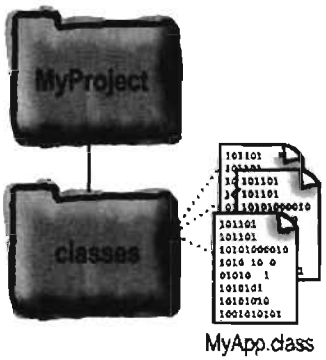
# Put your Java in a JAR


A **JAR** file is a **J**ava **AR**chive. It's based on the pkzip file format, and it lets you bundle all your classes so that instead of presenting your client with 28 class files, you hand over just a single JAR file. If you're familiar with the tar command on UNIX, you'll recognize the jar tool commands. (Note: when we say JAR in all caps, we're referring to the archive *file*. When we use lowercase, we're referring to the *jar tool* you use to create JAR files.)

The question is, what does the client *do* with the JAR? How do you get it to *run*?

You make the JAR ***executable***.

An executable JAR means the end-user doesn't have to pull the class files out before running the program. The user can run the app while the class files are still in the JAR. The trick is to create a ***manifest*** file, that goes in the JAR and holds information about the files in the JAR. To make a JAR executable, the manifest must tell the JVM *which class has the main() method!*




## Making an executable JAR

- **Make sure all of your class files are in the classes directory**

  We're going to refine this in a few pages, but for now, keep all your class files sitting in the directory named 'classes'.

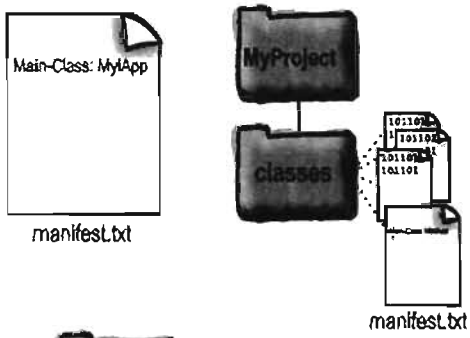
- **Create a manifest.txt file that states which class has the main() method**

  Make a text file named manifest.txt that has a one line:

  ```
  Main-Class: MyApp
  ```

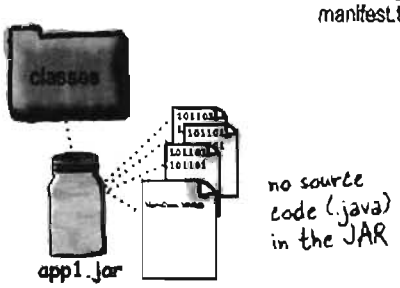
  don't put the .class on the end

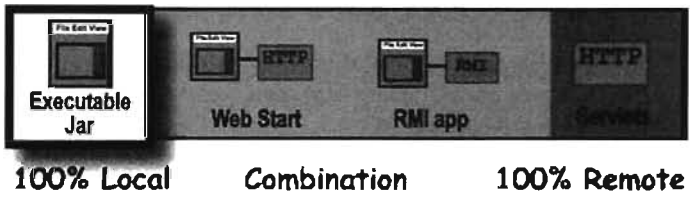
  Press the return key after typing the Main-Class line, or your manifest may not work correctly. Put the manifest file into the "classes" directory.




- **Run the jar tool to create a JAR file that contains everything in the classes directory, plus the manifest.**

```
%cd MiniProject/classes
%jar -cvmf manifest.txt app1.jar *.class
OR
%jar -cvmf manifest.txt app1.jar MyApp.class
```



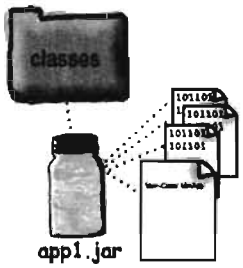



Most 100% local Java apps are deployed as executable JAR files.

## Running (executing) the JAR

Java (the JVM) is capable of loading a class from a JAR, and calling the main() method of that class. In fact, the entire application can *stay* in the JAR. Once the ball is rolling (i.e., the main() method starts running), the JVM doesn't care *where* your classes come from, as long as it can find them. And one of the places the JVM looks is within any JAR files in the classpath. If it can *see* a JAR, the JVM will *look* in that JAR when it needs to find and load a class.

The JVM has to 'see' the JAR, so it must be in your classpath. The easiest way to make the JAR visible is to make your working directory the place where the JAR is.




```
%cd MyProject/classes

%java -jar app1.jar
```

The -jar flag tells the JVM you're giving it a JAR instead of a class.

The JVM looks inside this JAR for a manifest with an entry for Main-Class. If it doesn't find one, you get a runtime exception.

Depending on how your operating system is configured, you might even be able to simply double-click the JAR file to launch it. This works on most flavors of Windows, and Mac OS X. You can usually make this happen by selecting the JAR and telling the OS to "Open with..." (or whatever the equivalent is on your operating system).

## there are no Dumb Questions

**Q:** **Why can't I just JAR up an entire directory?**

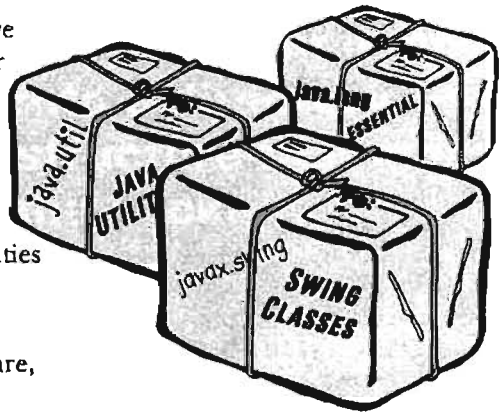
**A:** The JVM looks inside the JAR and expects to find what it needs *right there*. It won't go digging into other directories, unless the class is part of a package, and even *then* the JVM looks only in the directories that match the package statement?

**Q:** **What did you just say?**

**A:** You can't put your class files into some arbitrary directory and JAR them up that way. But if your classes belong to packages, you can JAR up the entire package directory structure. In fact, you *must*. We'll explain all this on the next page, so you can relax.

# Put your classes in packages!

So you've written some nicely reusable class files, and you've posted them in your internal development library for other programmers to use. While basking in the glow of having just delivered some of the (in your humble opinion) best examples of OO ever conceived, you get a phone call. A frantic one. Two of your classes have the same name as the classes Fred just delivered to the library. And all hell is breaking loose out there, as naming collisions and ambiguities bring development to its knees.




And all because you didn't use packages! Well, you did use packages, in the sense of using classes in the Java API that are, of course, in packages. But you didn't put your own classes into packages, and in the Real World, that's Really Bad.

We're going to modify the organizational structure from the previous pages, just a little, to put classes into a package, and to JAR the entire package. Pay very close attention to the subtle and picky details. Even the tiniest deviation can stop your code from compiling and/or running.

## Packages prevent class name conflicts

Although packages aren't just for preventing name collisions, that's a key feature. You might write a class named Customer and a class named Account and a class named ShoppingCart. And what do you know, half of all developers working in enterprise e-commerce have probably written classes with those names. In an OO world, that's just dangerous. If part of the point of OO is to write reusable components, developers need to be able to piece together components from a variety of sources, and build something new out of them. Your components have to be able to 'play well with others', including those you didn't write or even know about.

Remember way back in chapter 6 when we discussed how a package name is like the full name of a class, technically known as the *fully-qualified name*. Class ArrayList is really *java.util.ArrayList*, JButton is really *javax.swing.JButton*, and Socket is really *java.net.Socket*. Notice that two of those classes, ArrayList and Socket, both have *java* as their "first name". In other words, the first part of their fully-qualified names is "java". Think of a hierarchy when you think of package structures, and organize your classes accordingly.

**Package structure of the Java API for:**

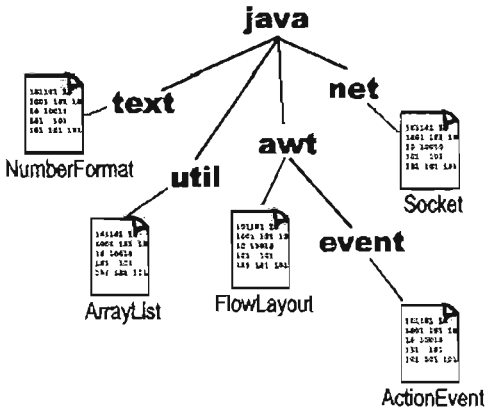
java.text.NumberFormat

java.util.ArrayList

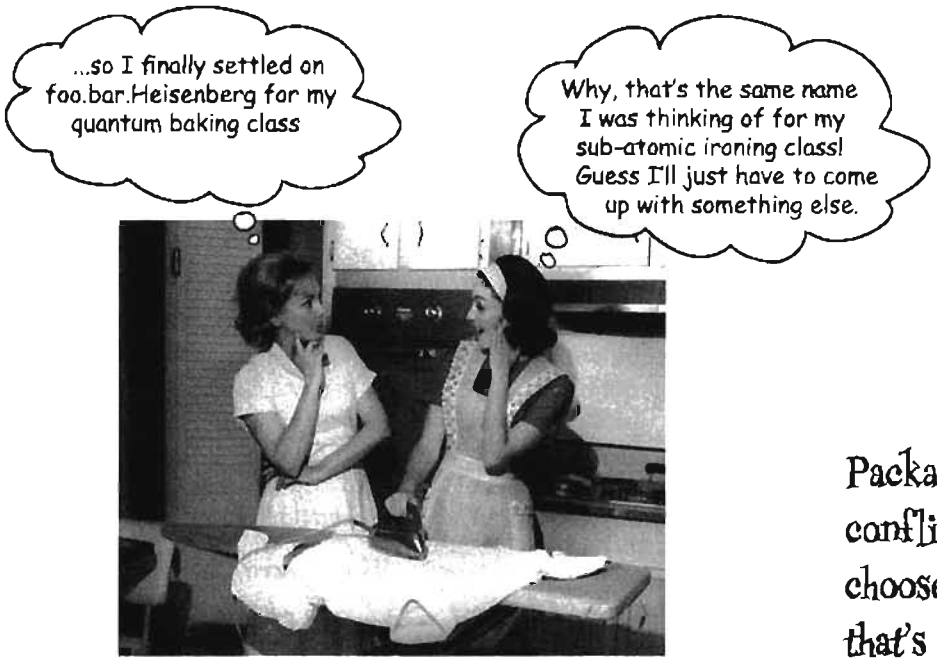
java.awt.FlowLayout

java.awt.event.ActionEvent

java.net.Socket




**What does this picture look like to you? Doesn't it look a whole lot like a directory hierarchy?**




Packages can prevent name conflicts, but only if you choose a package name that's guaranteed to be unique. The best way to do that is to preface your packages with your reverse domain name.

com.headfirstbooks.Book

package name

class name

## Preventing package name conflicts

Putting your class in a package reduces the chances of naming conflicts with other classes, but what's to stop two programmers from coming up with identical *package* names? In other words, what's to stop two programmers, each with a class named Account, from putting the class in a package named shopping.customers? Both classes, in that case, would *still* have the same name:

***shopping.customers.Account***

Sun strongly suggests a package naming convention that greatly reduces that risk—prepend every class with your reverse domain name. Remember, domain names are guaranteed to be unique. Two different guys can be named Bartholomew Simpson, but two different domains cannot be named doh.com.

### Reverse domain package names

com.headfirstjava.projects.Chart

the class name is always capitalized

start the package with your reverse domain, separated by a dot (.), then add your own organizational structure after that

projects.Chart might be a common name, but adding com.headfirstjava means we have to worry about only our own in-house developers.

## To put your class in a package:

### 1 Choose a package name

We're using com.headfirstjava as our example. The class name is PackageExercise, so the fully-qualified name of the class is now: com.headfirstjava.PackageExercise.

### 2 Put a package statement in your class

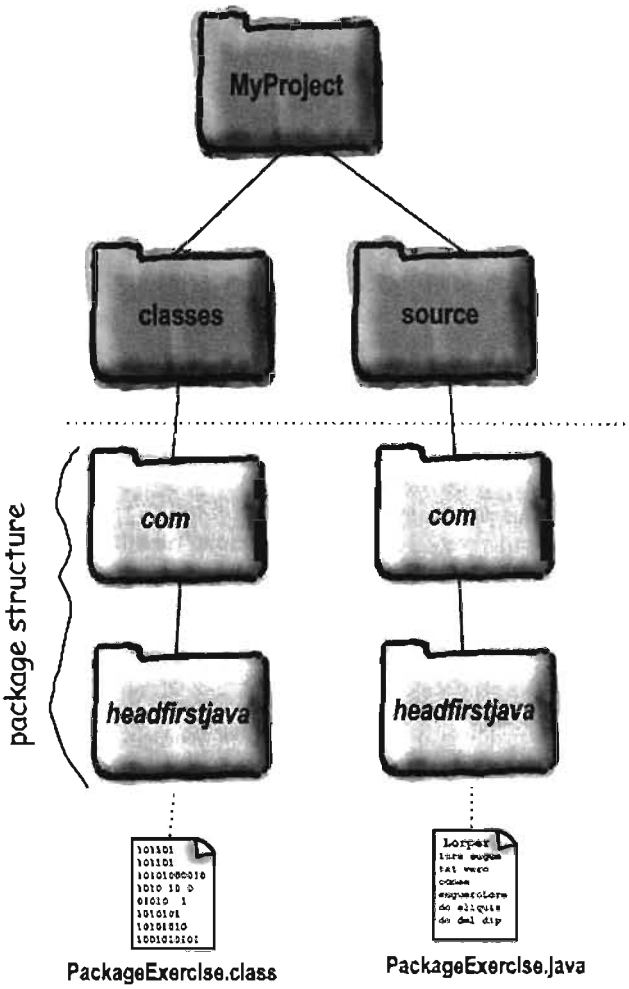
It must be the first statement in the source code file, above any import statements. There can be only one package statement per source code file, so **all classes in a source file must be in the same package**. That includes inner classes, of course.

```
package com.headfirstjava;

import javax.swing.*;

public class PackageExercise {
   // life-altering code here
}
```

### 3 Set up a matching directory structure

It's not enough to *say* your class is in a package, by merely putting a package statement in the code. Your class isn't *truly* in a package until you put the class in a matching directory structure. So, if the fully-qualified class name is com.headfirstjava.PackageExercise, you *must* put the PackageExercise source code in a directory named **headfirstjava**, which *must* be in a directory named **com**.
It is *possible* to compile without doing that, but trust us—it's not worth the other problems you'll have. Keep your source code in a directory structure that matches the package structure, and you'll avoid a ton of painful headaches down the road.

**You must put a class into a directory structure that matches the package hierarchy.**




Set up a matching directory structure for both the source and classes trees.

# Compiling and running with packages

When your class is in a package, it's a little trickier to compile and run. The main issue is that both the compiler and JVM have to be capable of finding your class and all of the other classes it uses. For the classes in the core API, that's never a problem. Java always knows where its own stuff is. But for your classes, the solution of compiling from the same directory where the source files are simply won't work (or at least not *reliably*). We guarantee, though, that if you follow the structure we describe on this page, you'll be successful. There are other ways to do it, but this is the one we've found the most reliable and the easiest to stick to.

## Compiling with the -d (directory) flag

```
%cd MyProject/source
```

stay in the source directory! Do NOT cd down into the directory where the .java file is!

```
%javac -d ../classes com/headfirstjava/PackageExercise.java
```

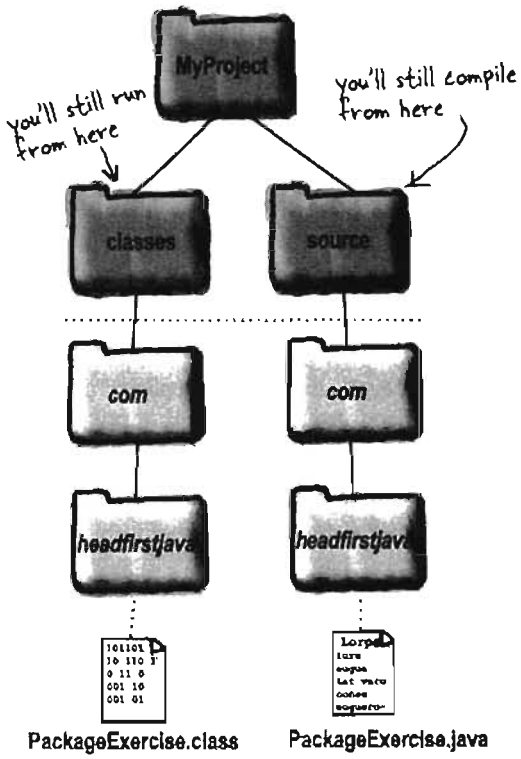
tells the compiler to put the compiled code (class files) into the classes directory, within the right package structure!! Yes, it knows.

Now you have to specify the PATH to get to the actual source file.

To compile all the .java files in the com.headfirstjava package, use:

```
%javac -d ../classes com/headfirstjava/*.java
```

compiles every source (.java) file in this directory




## Running your code

```
%cd MyProject/classes
```

run your program from the 'classes' directory.

```
%java com.headfirstjava.PackageExercise
```

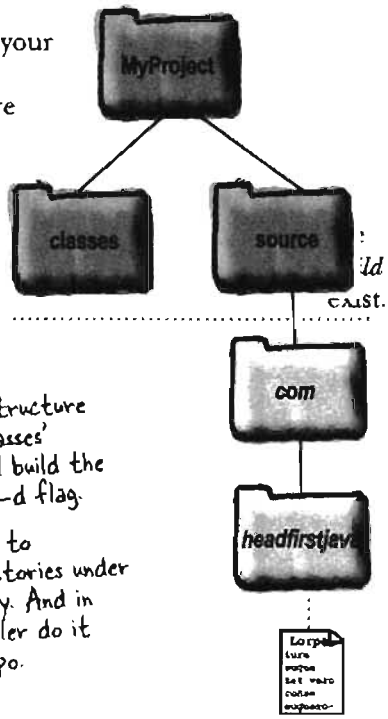
You MUST give the fully-qualified class name! The JVM will see that, and immediately look inside its current directory (classes) and expect to find a directory named com, where it expects to find a directory named headfirstjava, and in there it expects to find the class. If the class is in the "com" directory, or even in "classes", it won't work!

## The -d flag is even cooler than we said

Compiling with the **-d** flag is wonderful because not only does it let you send your compiled class files into a directory other than the one where the source file is, but it also knows to put the class into the correct directory structure for the package the class is in.

But it gets even better!

Let's say that you have a nice directory structure all set up for your source code. But you haven't set up a matching directory structure for your classes directory. Not a problem! Compiling with **-d** tells the compiler to not just *put* your classes into correct directory tree, but to the directories if they don't




If the package directory structure doesn't exist under the 'classes' directory, the compiler will build the directories if you use the -d flag.

So you don't actually have to physically create the directories under the 'classes' root directory. And in fact, if you let the compiler do it there's no chance of a typo.

**The -d flag tells the compiler, "Put the class into its package directory structure, using the class specified after the -d as the root directory. But... if the directories aren't there, create them first and *then* put the class in the right place!"**

### there are no Dumb Questions

**Q: I tried to cd into the directory where my main class was, but now the JVM says it can't find my class! But it's right THERE in the current directory!**

A: Once your class is in a package, you can't call it by its 'short' name. You MUST specify, at the command-line, the fully-qualified name of the class whose main() method you want to run. But since the fully-qualified name includes the *package* structure, Java insists that the class be in a matching *directory* structure. So if at the command-line you say:

```
%java com.foo.Book
```

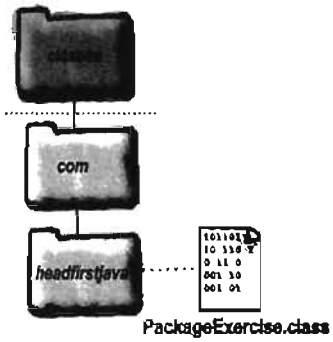
the JVM will look in its current directory (and the rest of its classpath), for a directory named "com". ***It will not look for a class named Book, until it has found a directory named "com" with a directory inside named "foo".*** Only then will the JVM accept that its found the correct Book class. If it finds a Book class anywhere else, it assumes the class isn't in the right structure, even if it is! The JVM won't for example, look back up the directory tree to say, "Oh, I can see that above us is a directory named com, so this must be the right package..."

# Making an executable JAR with packages


When your class is in a package, the package directory structure must be inside the JAR! You can't just pop your classes in the JAR the way we did pre-packages. And you must be sure that you don't include any other directories above your package. The first directory of your package (usually com) must be the first directory within the JAR! If you were to accidentally include the directory *above* the package (e.g. the "classes" directory), the JAR wouldn't work correctly.

## Making an executable JAR

**Make sure all of your class files are within the correct package structure, under the classes directory.**

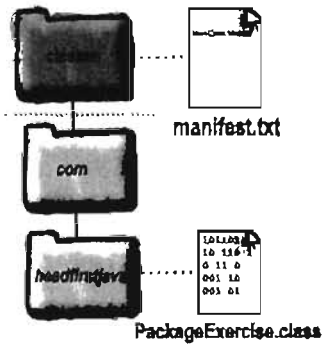



**Create a manifest.txt file that states which class has the main() method, and be sure to use the fully-qualified class name!**

Make a text file named manifest.txt that has a single line:

```
Main-Class: com.headfirstjava.PackageExercise
```

Put the manifest file into the classes directory

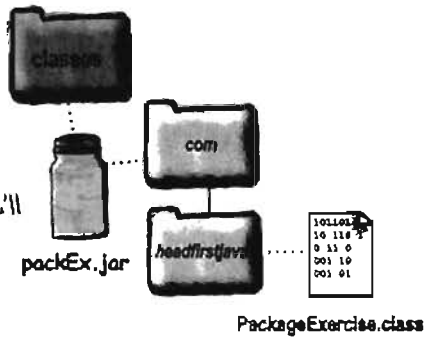



**Run the jar tool to create a JAR file that contains the package directories plus the manifest**

The only thing you need to include is the 'com' directory, and the entire package (and all classes) will go into the JAR.

```
%cd MyProject/classes

%jar -cvmf  manifest.txt  packEx.jar  com
```

All you specify is the com directory! And you'll get everything in it!



# So where did the manifest file go?

Why don't we look inside the JAR and find out? From the command-line, the jar tool can do more than just create and run a JAR. You can extract the contents of a JAR (just like 'unzipping' or 'untarring').

Imagine you've put the packEx.jar into a directory named Skyler.

## jar commands for listing and extracting

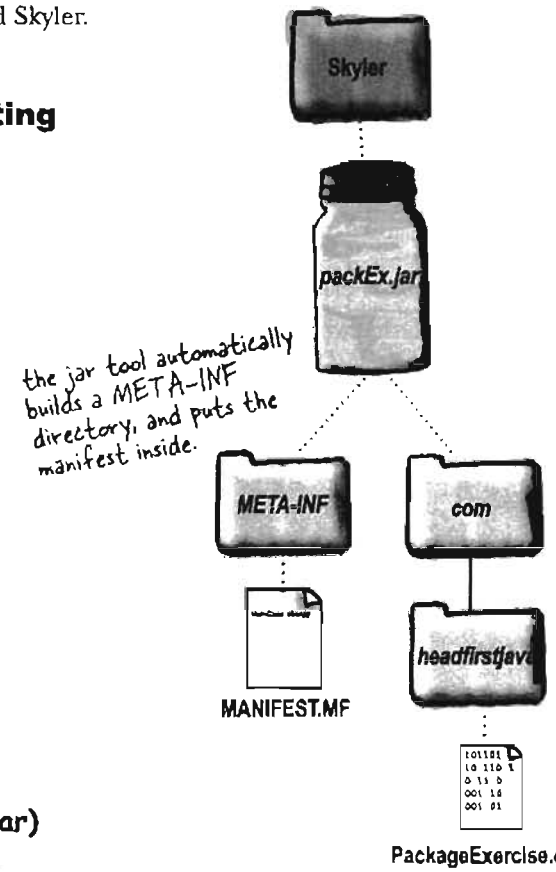
### 1 List the contents of a JAR

```
% jar -tf packEx.jar
```

-tf stands for 'Table File' as in "show me a table of the JAR file"

```
File Edit Window Help Pickle
% cd Skyler
% jar -tf packEx.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/headfirstjava/
com/headfirstjava/
PackageExercise.class
```

we put the JAR file into a directory named Skyler

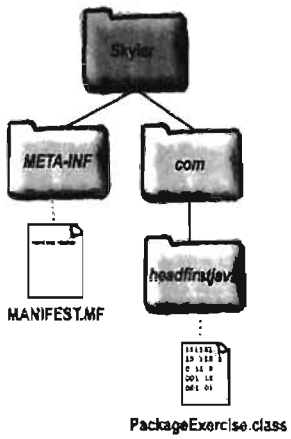



the jar tool automatically builds a META-INF directory, and puts the manifest inside.

### 2 Extract the contents of a JAR (i.e. unjar)

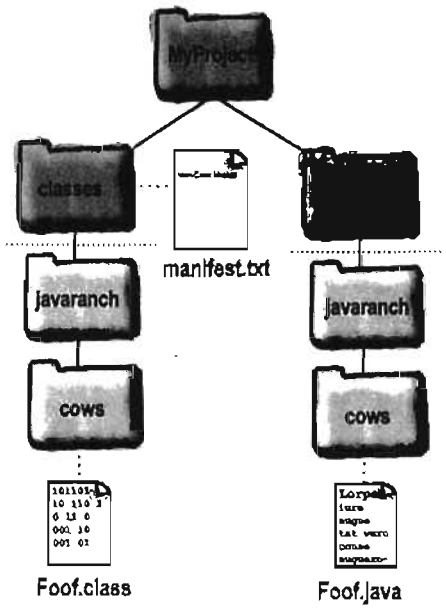
```
% cd Skyler
% jar -xf packEx.jar
```

-xf stands for 'Extract File' and it works just like unzipping or untarring. If you extract the packEx.jar, you'll see the META-INF directory and the com directory directory in your current directory




META-INF stands for 'meta information'. The jar tool creates the META-INF directory as well as the MANIFEST.MF file. It also takes the contents of *your* manifest file, and puts it into the MANIFEST.MF file. So, *your* manifest *file* doesn't go into the JAR, but the *contents* of it are put into the 'real' manifest (MANIFEST.MF).

## Sharpen your pencil




Given the package/directory structure in this picture, figure out what you should type at the command-line to compile, run, create a JAR, and execute a JAR. Assume we're using the standard where the package directory structure starts just below *source* and *classes*. In other words, the *source* and *classes* directories are not part of the package.

**Compile:**

```
%cd source
%javac _____
```

**Run:**

```
%cd _____

%java _____
```

**Create a JAR**

```
%cd _____

% _____
```

**Execute a JAR**

```
%cd _____

% _____
```

Bonus question: What's wrong with the package name?

## there are no Dumb Questions

**Q: What happens if you try to run an executable JAR, and the end-user doesn't have Java installed?**

A: Nothing will run, since without a JVM, Java code can't run. The end-user must have Java installed.

**Q: How can I get Java installed on the end-user's machine?**

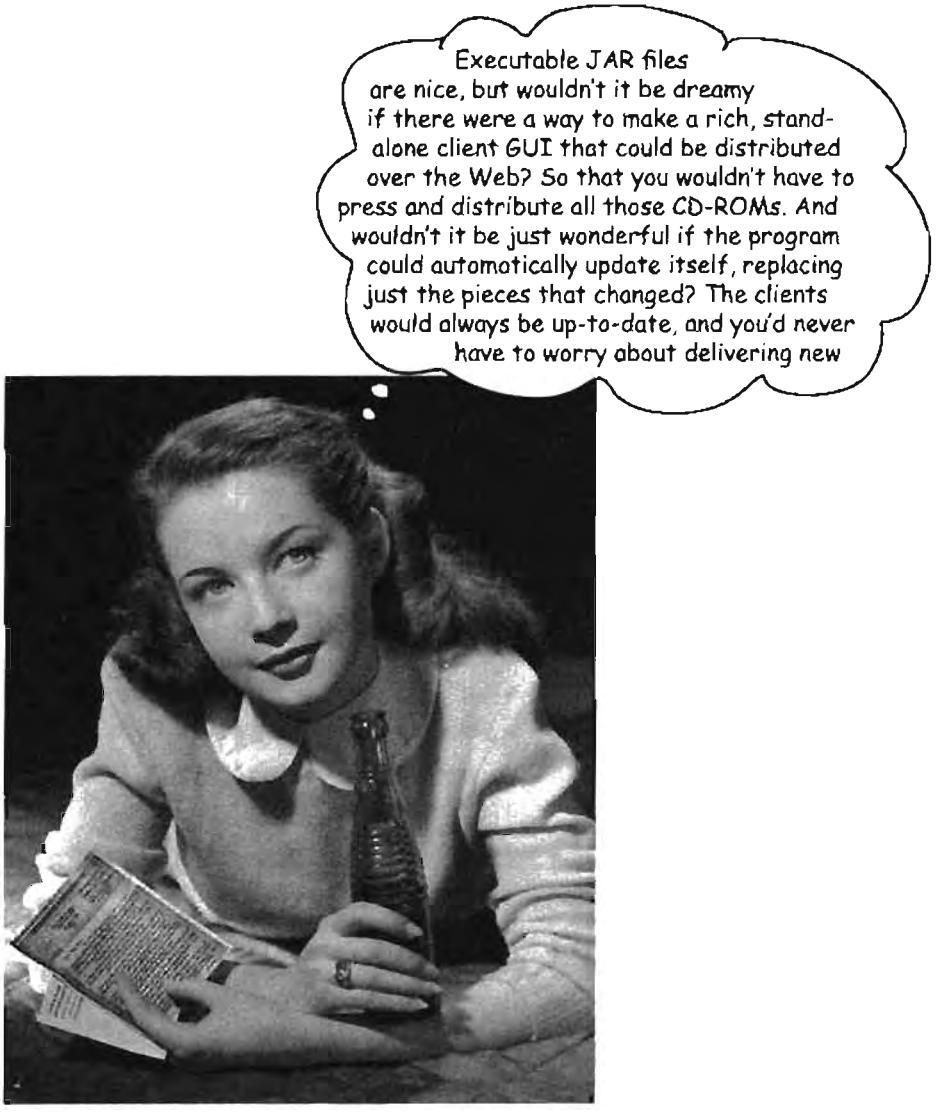
Ideally, you can create a custom installer and distribute it along with your application. Several companies offer installer programs ranging from simple to extremely powerful. An installer program could, for example, detect whether or not the end-user has an appropropriate version of Java installed, and if not, install and configure Java before installing your application. InstallShield, InstallAnywhere, and DeployDirector all offer Java installer solutions.

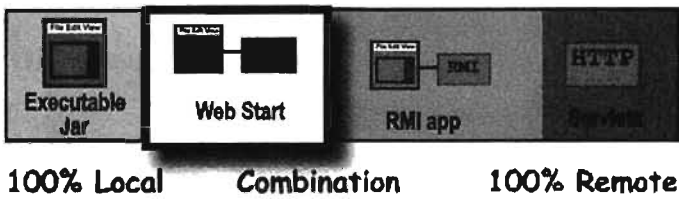
Another cool thing about some of the installer programs is that you can even make a deployment CD-ROM that includes installers for all major Java platforms, so... one CD to rule them all. If the user's running on Solaris, for example, the Solaris version of Java is installed. On Windows, the Windows, version, etc. If you have the budget, this is by far the easiest way for your end-users to get the right version of Java installed and configured.

## BULLET POINTS

- Organize your project so that your source code and class files are not in the same directory.
- A standard organization structure is to create a *project* directory, and then put a *source* directory and a *classes* directory inside the project directory.
- Organizing your classes into packages prevents naming collisions with other classes, if you prepend your reverse domain name on to the front of a class name.
- To put a class in a package, put a package statement at the top of the source code file, before any import statements:
  ```
  package com.wickedlysmart;
  ```
- To be in a package, a class must be in a *directory structure that exactly matches the package structure*. For a class, com.wickedlysmart.Foo, the Foo class must be in a directory named *wickedlysmart*, which is in a directory named *com*.
- To make your compiled class land in the correct package directory structure under the *classes* directory, use the `-d` compiler flag:
  ```
  % cd source
  % javac -d ../classes com/wickedlysmart/Foo.java
  ```
- To run your code, cd to the classes directory, and give the fully-qualified name of your class:
  ```
  % cd classes
  % java com.wickedlysmart.Foo
  ```
- You can bundle your classes into JAR (Java ARchive) files. JAR is based on the pkzip format.
- You can make an executable JAR file by putting a manifest into the JAR that states which class has the main() method. To create a manifest file, make a text file with an entry like the following (for example):
  ```
  Main-Class: com.wickedlysmart.Foo
  ```
- Be sure you hit the return key after typing the Main-Class line, or your manifest file may not work.
- To create a JAR file, type:
  ```
  jar -cvfm manifest.txt MyJar.jar com
  ```
- The entire package directory structure (and *only* the directories matching the package) must be immediately inside the JAR file.
- To run an executable JAR file, type:
  ```
  java -jar MyJar.jar
  ```

**wouldn't it be** dreamy...






# Java Web Start

With Java Web Start (JWS), your application is launched for the first time from a Web browser (get it? *Web Start*?) but it runs as a stand-alone application (well, *almost*), without the constraints of the browser. And once it's downloaded to the end-user's machine (which happens the first time the user accesses the browser link that starts the download), it *stays* there.

Java Web Start is, among other things, a small Java program that lives on the client machine and works much like a browser plug-in (the way, say, Adobe Acrobat Reader opens when your browser gets a .pdf file). This Java program is called the **Java Web Start 'helper app'**, and its key purpose is to manage the downloading, updating, and launching (executing) of *your* JWS apps.

When JWS downloads your application (an executable JAR), it invokes the main() method for your app. After that, the end-user can launch your application directory from the JWS helper app *without* having to go back through the Web page link.

But that's not the best part. The amazing thing about JWS is its ability to detect when even a small part of application (say, a single class file) has changed on the server, and—without any end-user intervention—download and integrate the updated code.

There's still an issue, of course, like how does the end-user *get* Java and Java Web Start? They need both—Java to run the app, and Java Web Start (a small Java application itself) to handle retrieving and launching the app. But even *that* has been solved. You can set things up so that if your end-users don't have JWS, they can download it from Sun. And if they *do* have JWS, but their version of Java is out-of-date (because you've specified in your JWS app that you need a specific version of Java), the Java 2 Standard Edition can be downloaded to the end-user machine.

Best of all, it's simple to use. You can serve up a JWS app much like any other type of Web resource such as a plain old HTML page or a JPEG image. You set up a Web (HTML) page with a link to your JWS application, and you're in business.

In the end, your JWS application isn't much more than an executable JAR that end-users can download from the Web.

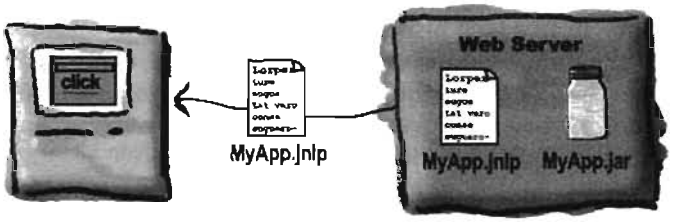
End-users launch a Java Web Start app by clicking on a link in a Web page. But once the app downloads, it runs outside the browser, just like any other stand-alone Java application. In fact, a Java Web Start app is just an executable JAR that's distributed over the Web.

## How Java Web Start works


① **The client clicks on a Web page link to your JWS application (a .jnlp file).**

The Web page link

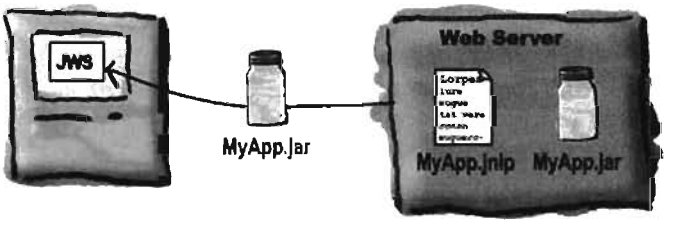
```
<a href="MyApp.jnlp">Click</a>
```



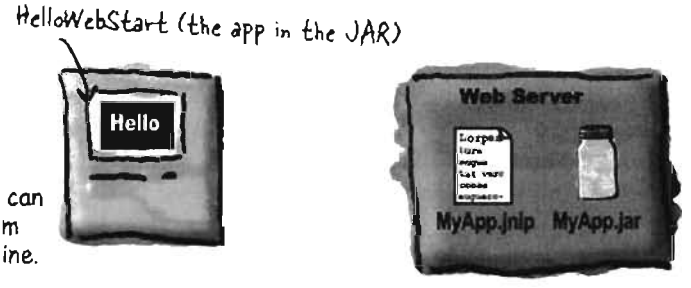

② **The Web server (HTTP) gets the request and sends back a .jnlp file (this is NOT the JAR).**

The .jnlp file is an XML document that states the name of the application's executable JAR file.


③ **Java Web Start (a small 'helper app' on the client) is started up by the browser. The JWS helper app reads the .jnlp file, and asks the server for the MyApp.jar file.**




④ **The Web server 'serves' up the requested .jar file.**




⑤ **Java Web Start gets the JAR and starts the application by calling the specified main( ) method (just like an executable JAR).**

Next time the user wants to run this app, he can open the Java Web Start application and from there launch your app, without even being online.

# The .jnlp file

To make a Java Web Start app, you need to .jnlp (Java Network Launch Protocol) file that describes your application. This is the file the JWS app reads and uses to find your JAR and launch the app (by calling the JAR's main() method). A .jnlp file is a simple XML document that has several different things you can put in, but as a minimum, it should look like this:

```
<?xml version="1.0" encoding="utf-8"?>

<jnlp spec="0.2 1.0"
      codebase="http://127.0.0.1/~kathy"
      href="MyApp.jnlp">

   <information>
      <title>kathy App</title>
      <vendor>Wickedly Smart</vendor>
      <homepage href="index.html"/>
      <description>Head First WebStart demo</description>
      <icon href="kathys.gif"/>
      <offline-allowed/>
   </information>

   <resources>
      <j2se version="1.3+"/>
      <jar href="MyApp.jar"/>
   </resources>

   <application-desc main-class="HelloWebStart"/>
</jnlp>
```

The 'codebase' tag is where you specify the 'root' of where your web start stuff is on the server. We're testing this on our localhost, so we're using the local loopback address "127.0.0.1". For web start apps on our internet web server, this would say, "http://www.wickedlysmart.com"

This is the location of the .jnlp file relative to the codebase. This example shows that MyApp.jnlp is available in the root directory of the web server, not nested in some other directory.

Be sure to include all of these tags, or your app might not work correctly! The 'information' tags are used by the JWS helper app, mostly for displaying when the user wants to relaunch a previously-downloaded application.

This means the user can run your program without being connected to the internet. If the user is offline, it means the automatic-updating feature won't work.

This says that your app needs version 1.3 of Java, or greater.

The name of your executable JAR! You might have other JAR files as well, that hold other classes or even sounds and images used by your app.

This is like the mainfest Main-Class entry... it says which class in the JAR has the main() method.

# Steps for making and deploying a Java Web Start app

1. Make an executable JAR for your application.

MyApp.jar

2. Write a .jnlp file.

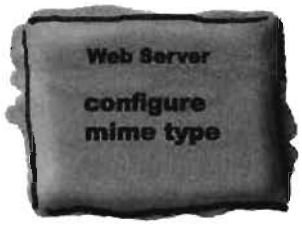
MyApp.jnlp

3. Place your JAR and .jnlp files on your Web server.


4. Add a new mime type to your Web server.

```
application/x-java-jnlp-file
```

This causes the server to send the .jnlp file with the correct header, so that when the browser receives the .jnlp file it knows what it is and knows to start the JWS helper app.



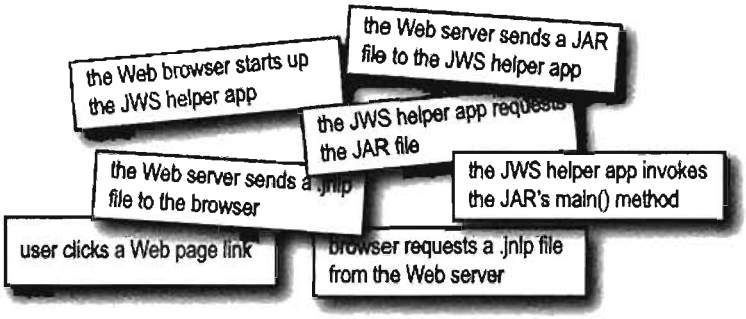

5. Create a Web page with a link to your .jnlp file

MyJWSApp.html

```
<HTML>
  <BODY>
    <a href="MyApp2.jnlp">Launch My Application</a>
  </BODY>
</HTML>
```


## What's First?

Look at the sequence of events below, and place them in the order in which they occur in a JWS application.




1. 
2. 
3. 
4. 
5. 
6. 
7. 

## there are no Dumb Questions

**Q: How is Java Web Start different from an applet?**

A: Applets can't live outside of a Web browser. An applet is downloaded from the Web as part of a Web page rather than simply from a Web page. In other words, to the browser, the applet is just like a JPEG or any other resource. The browser uses either a Java plug-in or the browser's own built-in Java (far less common today) to run the applet. Applets don't have the same level of functionality for things such as automatic updating, and they must always be launched from the browser. With JWS applications, once they're downloaded from the Web, the user doesn't even have to be using a browser to relaunch the application locally. Instead, the user can start up the JWS helper app, and use it to launch the already-downloaded application again.

**Q: What are the security restrictions of JWS?**

A: JWS apps have several limitations including being restricted from reading and writing to the user's hard drive. But... JWS has its own API with a special open and save dialog box so that, with the user's permission, your app can save and read its own files in a special, restricted area of the user's drive.

### BULLET POINTS

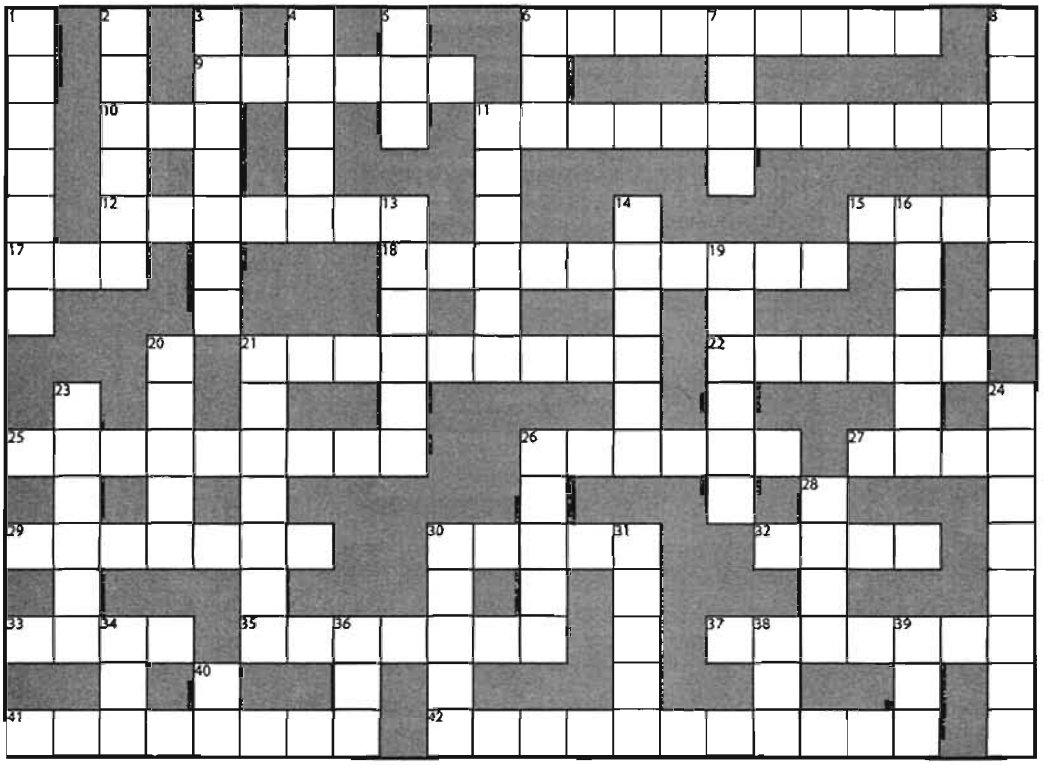
- Java Web Start technology lets you deploy a stand-alone client application from the Web.
- Java Web Start includes a 'helper app' that must be installed on the client (along with Java).
- A Java Web Start (JWS) app has two pieces: an executable JAR and a .jnlp file.
- A .jnlp file is a simple XML document that describes your JWS application. It includes tags for specifying the name and location of the JAR, and the name of the class with the main() method.
- When a browser gets a .jnlp file from the server (because the user clicked on a link to the .jnlp file), the browser starts up the JWS helper app.
- The JWS helper app reads the .jnlp file and requests the executable JAR from the Web server.
- When the JWS gets the JAR, it invokes the main() method (specified in the .jnlp file).

**exercise:** True or False


We explored packaging, deployment, and JWS in this chapter. Your job is to decide whether each of the following statements is true or false.


1. The Java compiler has a flag, -d, that lets you decide where your .class files should go.
2. A JAR is a standard directory where your .class files should reside.
3. When creating a Java Archive you must create a file called jar.mf.
4. The supporting file in a Java Archive declares which class has the main() method.
5. JAR files must be unzipped before the JVM can use the classes inside.
6. At the command line, Java Archives are invoked using the -arch flag.
7. Package structures are meaningfully represented using hierarchies.
8. Using your company's domain name is not recommended when naming packages.
9. Different classes within a source file can belong to different packages.
10. When compiling classes in a package, the -p flag is highly recommended.
11. When compiling classes in a package, the full name must mirror the directory tree.
12. Judicious use of the -d flag can help to assure that there are no typos in your class tree.
13. Extracting a JAR with packages will create a directory called meta-inf.
14. Extracting a JAR with packages will create a file called manifest.mf.
15. The JWS helper app always runs in conjunction with a browser.
16. JWS applications require a .nlp (Network Launch Protocol) file to work properly.
17. A JWS's main method is specified in its JAR file.


# Summary-Cross 7.0



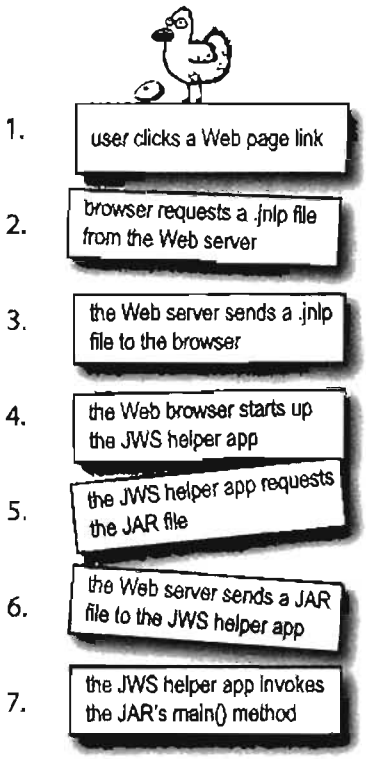
Anything in the book is fair game for this one!

## Across

6. Won't travel
9. Don't split me
10. Release-able
11. Got the key
12. I/O gang
15. Flatten
17. Encapsulated returner
18. Ship this one
21. Make it so
22. I/O sieve
25. Disk leaf
26. Mine is unique
27. GUI's target
29. Java team
30. Factory
32. For a while
33. Atomic * 8
35. Good as new
37. Pairs event
41. Where do I start
42. A little firewall

## Down

1. Pushy widgets
2. ____ of my desire
3. 'Abandoned' moniker
4. A chunk
5. Math not trig
6. Be brave
7. Arrange well
8. Swing slang
11. I/O canals
13. Organized release
14. Not for an instance
16. Who's allowed
19. Efficiency expert
20. Early exit
21. Common wrapper
23. Yes or no
24. Java jackets
26. Not behavior
28. Socket's suite
30. I/O cleanup
31. Milli-nap
34. Trig method
36. Encaps method
38. JNLP format
39. VB's final
40. Java branch


**True** 1. The Java compiler has a flag, -d, that lets you decide where your .class files should go.

**False** 2. A JAR is a standard directory where your .class files should reside.

**False** 3. When creating a Java Archive you must create a file called jar.mf.

**True** 4. The supporting file in a Java Archive declares which class has the main() method.

**False** 5. JAR files must be unzipped before the JVM can use the classes inside.

**False** 6. At the command line, Java Archives are invoked using the -arch flag.

**True** 7. Package structures are meaningfully represented using hierarchies.

**False** 8. Using your company's domain name is not recommended when naming packages.

**False** 9. Different classes within a source file can belong to different packages.

**False** 10. When compiling classes in a package, the -p flag is highly recommended.

**True** 11. When compiling classes in a package, the full name must mirror the directory tree.

**True** 12. Judicious use of the -d flag can help to assure that there are no typos in your tree.

**True** 13. Extracting a JAR with packages will create a directory called meta-inf.

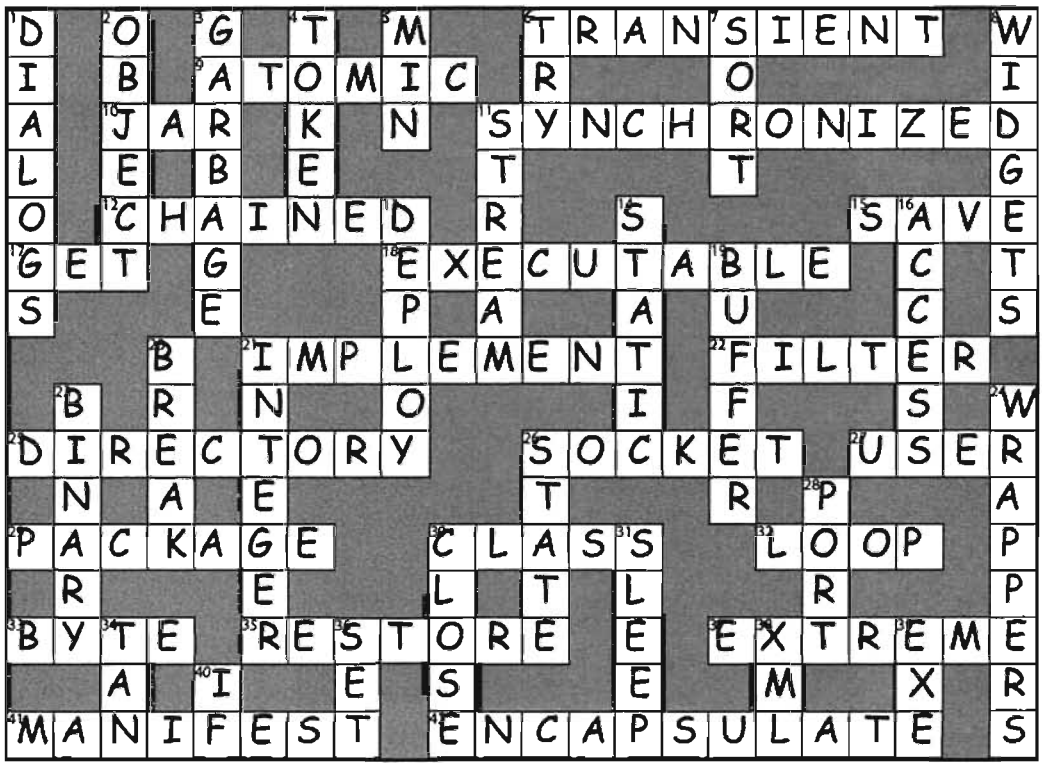
**True** 14. Extracting a JAR with packages will create a file called manifest.mf.

**False** 15. The JWS helper app always runs in conjunction with a browser.

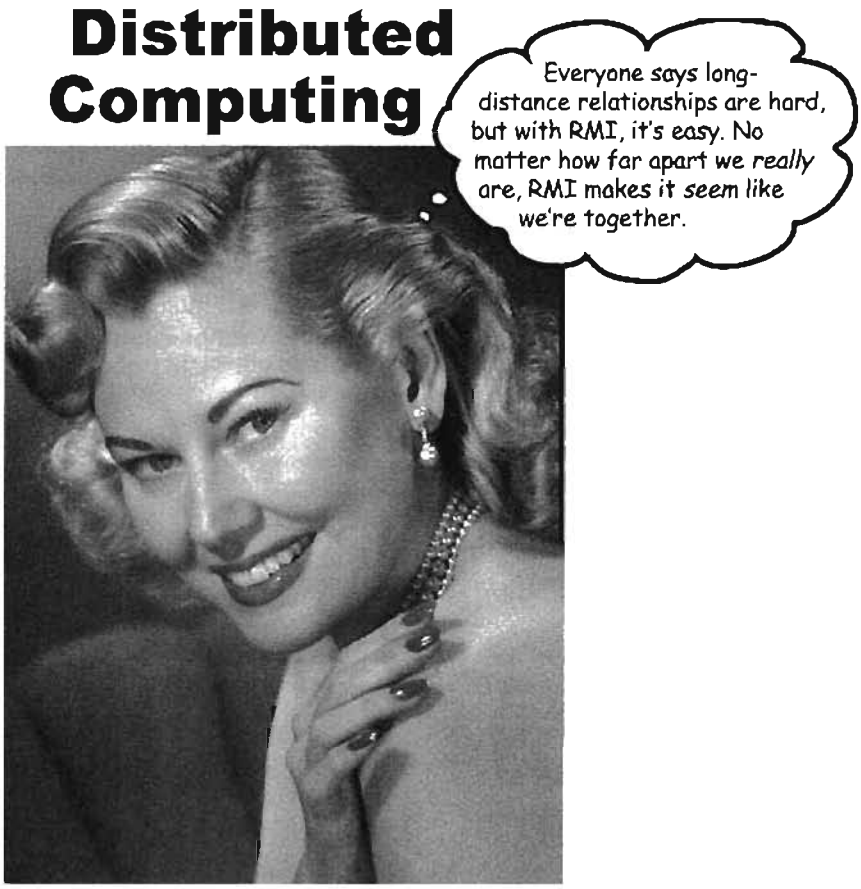
**False** 16. JWS applications require a .nlp (Network Launch Protocol) file to work properly.

**False** 17. A JWS's main method is specified in its JAR file.

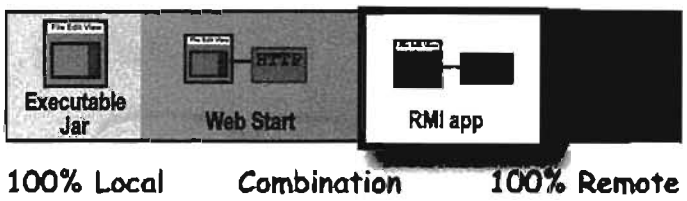
# Summary-Cross 7.0



# 18 remote deployment with RMI




**Being remote doesn't have to be a bad thing.** Sure, things *are* easier when all the parts of your application are in one place, in one heap, with one JVM to rule them all. But that's not always possible. Or desirable. What if your application handles powerful computations, but the end-users are on a wimpy little Java-enabled device? What if your app needs data from a database, but for security reasons, only code on your server can access the database? Imagine a big e-commerce back-end, that has to run within a transaction-management system? Sometimes, part of your app *must* run on a server, while another part (usually a client) must run on a *different* machine. In this chapter, we'll learn to use Java's amazingly simple Remote Method Invocation (RMI) technology. We'll also take a quick peek at Servlets, Enterprise Java Beans (EJB) , and Jini, and look at the ways in which EJB and Jini *depend* on RMI. We'll end the book by writing one of the coolest things you can make in Java, a *universal service browser*.

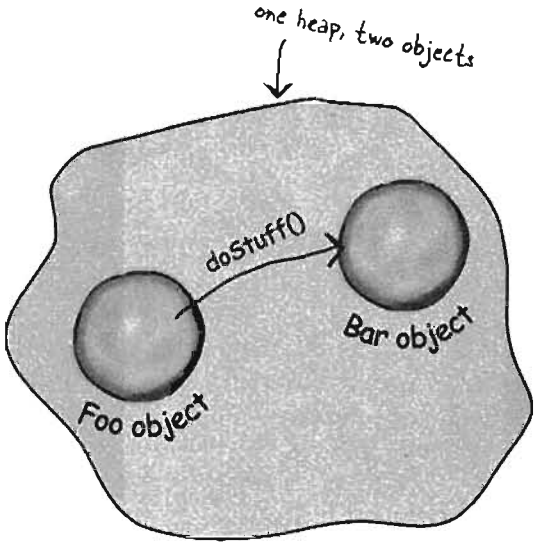



## Method calls are always between two objects on the same heap.

So far in this book, every method we've invoked has been on an object running in the same virtual machine as the caller. In other words, the calling object and the callee (the object we're invoking the method on) live on the same heap.

```
class Foo {
    void go() {
        Bar b = new Bar();
        b.doStuff();
    }
    public static void main (String[] args) {
       Foo f = new Foo();
       f.go();
    }
}
```

In the code above, we know that the Foo instance referenced by *f* and the Bar object referenced by *b* are both on the same heap, run by the same JVM. Remember, the JVM is responsible for stuffing bits into the reference variable that represent *how to get to an object on the heap*. The JVM always knows where each object is, and how to get to it. But the JVM can know about references on only its *own* heap! You can't, for example, have a JVM running on one machine knowing about the heap space of a JVM running on a *different* machine. In fact, a JVM running on one machine can't know anything about a different JVM running on the *same* machine. It makes no difference if the JVMs are on the same or different physical machines; it matters only that the two JVMs are, well, two different invocations of the JVM.



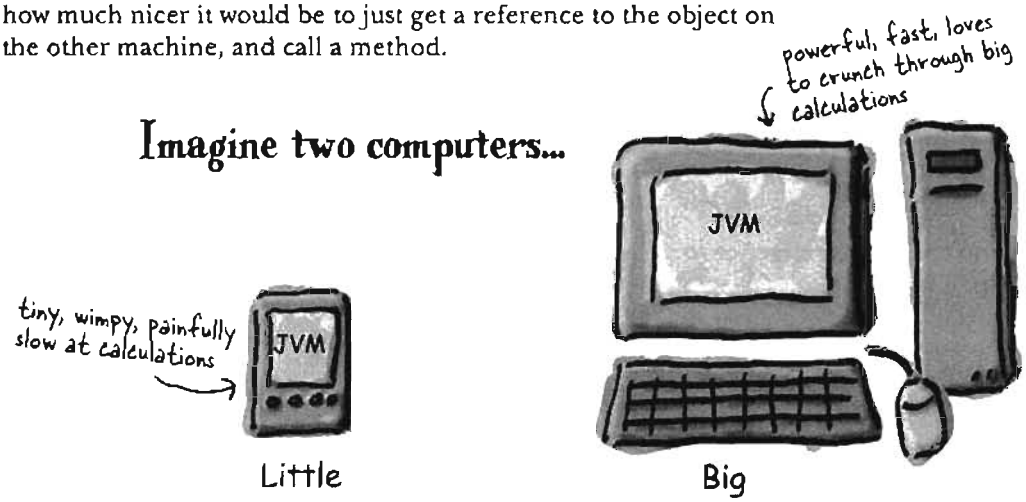

In most applications, when one object calls a method on another, both objects are on the same heap. In other words, both are running within the same JVM.

# What if you want to invoke a method on an object running on another machine?

We know how to get information from one machine to another—with Sockets and I/O. We open a Socket connection to another machine, and get an OutputStream and write some data to it.

But what if we actually want to *call a method* on something running in another machine... another JVM? Of course we could always build our own protocol, and when you send data to a ServerSocket the server could parse it, figure out what you meant, do the work, and send back the result on another stream. What a pain, though. Think how much nicer it would be to just get a reference to the object on the other machine, and call a method.

Imagine two computers...




Big has something Little wants.

Compute power.

Little wants to send some data to Big, so that Big can do the heavy computing.

Little wants simply to call a method...

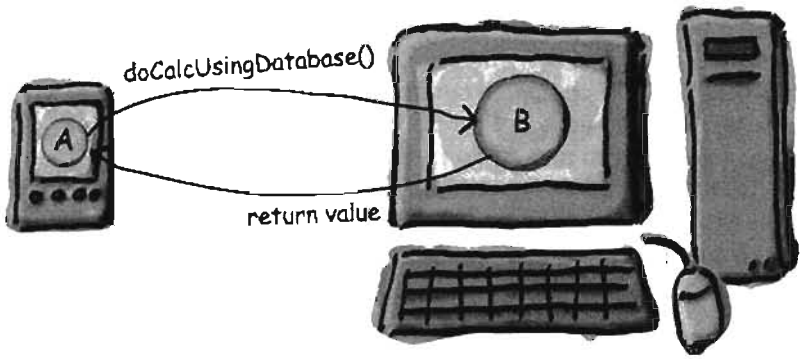
```
double doCalcUsingDatabase(CalcNumbers numbers)
```

and get back the result.

But how can Little get a reference to an object on Big?

## Object A, running on Little, wants to call a method on Object B, running on Big.

The question is, how do we get an object on one machine (which means a different heap/JVM) to call a method on another machine?




## But you can't do that.

Well, not directly anyway. You can't get a reference to something on another heap. If you say:

Dog d = ???

Whatever *d* is referencing must be in the same heap space as the code running the statement.

But imagine you want to design something that will use Sockets and I/O to communicate your intention (a method invocation on an object running on another machine), yet still *feel* as though you were making a local method call.

In other words, you want to cause a method invocation on a *remote* object (i.e., an object in a heap somewhere else), but with code that lets you *pretend* that you're invoking a method on a local object. The ease of a plain old everyday method call, but the power of remote method invocation. That's our goal.

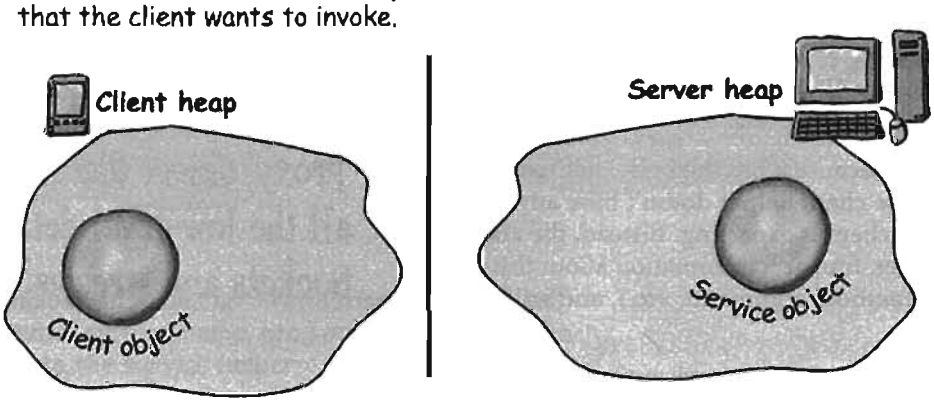
That's what RMI (Remote Method Invocation) gives you!

But let's step back and imagine how you would design RMI if you were doing it yourself. Understanding what you'd have to build yourself will help you learn how RMI works.

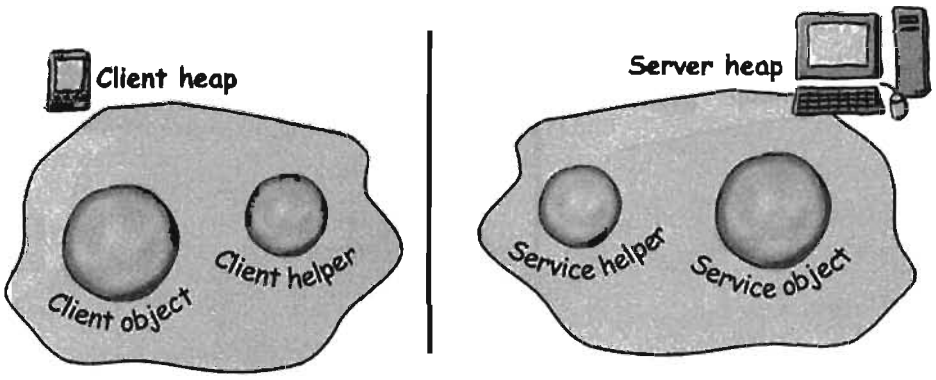
# A design for remote method calls

## Create four things: server, client, server helper, client helper

1. Create client and server apps. The server app is the **remote service** that has an object with the method that the client wants to invoke.




2. Create client and server 'helpers'. They'll handle all the low-level networking and I/O details so your client and service can pretend like they're in the same heap.



# The role of the 'helpers'

The 'helpers' are the objects that actually do the communicating. They make it possible for the client to *act* as though its calling a method on a local object. In fact, it *is*. The client calls a method on the client helper, *as if the client helper were the actual service. The client helper is a proxy for the Real Thing.*

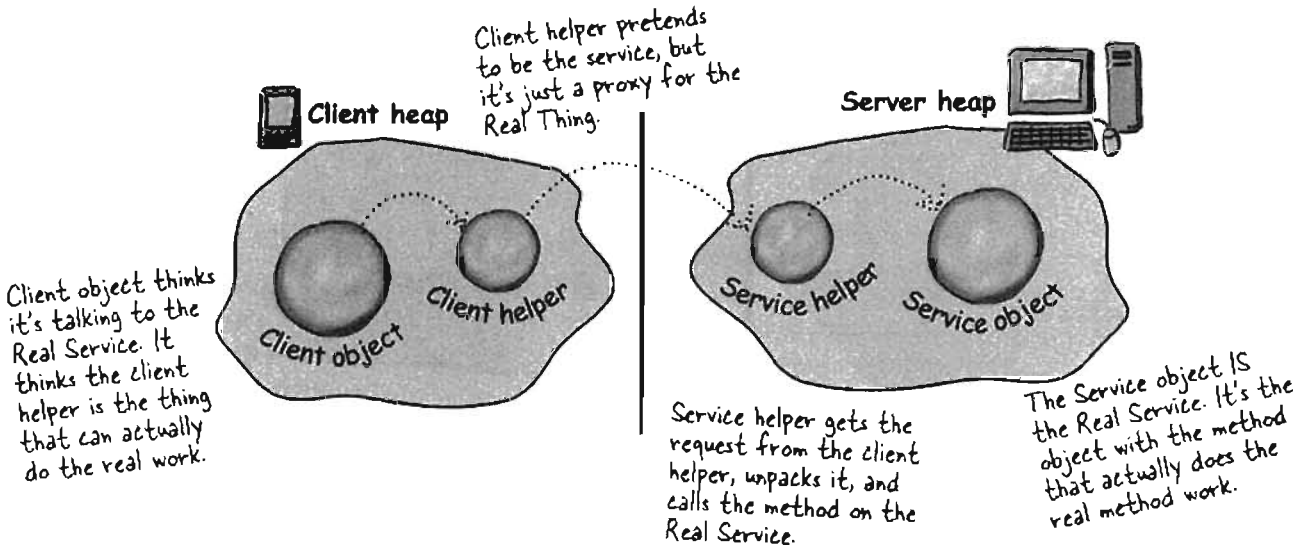
In other words, the client object *thinks* it's calling a method on the remote service, because the client helper is *pretending* to be the service object. ***Pretending to be the thing with the method the client wants to call!***

But the client helper isn't really the remote service. Although the client helper *acts* like it (because it has the same method that the service is advertising), the client helper doesn't have any of the actual method logic the client is expecting. Instead, the client helper contacts the server, transfers information about the method call (e.g., name of the method, arguments, etc.), and waits for a return from the server.

On the server side, the service helper receives the request from the client helper (through a Socket connection), unpacks the information about the call, and then invokes the *real* method on the *real* service object. So to the service object, the call is local. It's coming from the service helper, not a remote client.

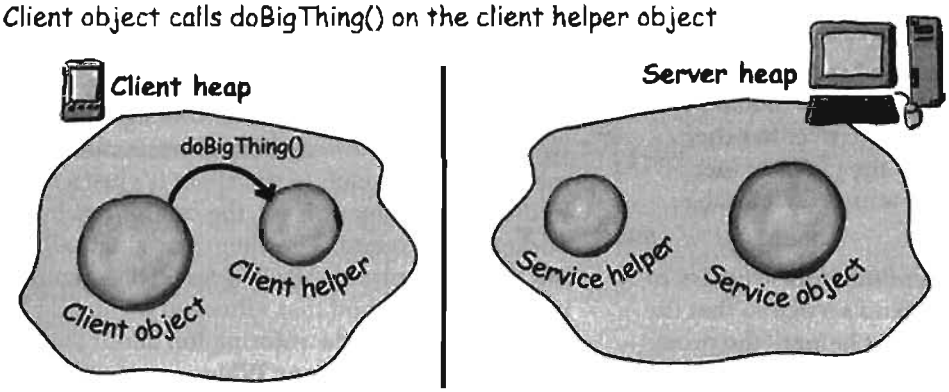
The service helper gets the return value from the service, packs it up, and ships it back (over a Socket's output stream) to the client helper. The client helper unpacks the information and returns the value to the client object.

Your client object gets to act like it's making remote method calls. But what it's really doing is calling methods on a heap-local 'proxy' object that handles all the low-level details of Sockets and streams.

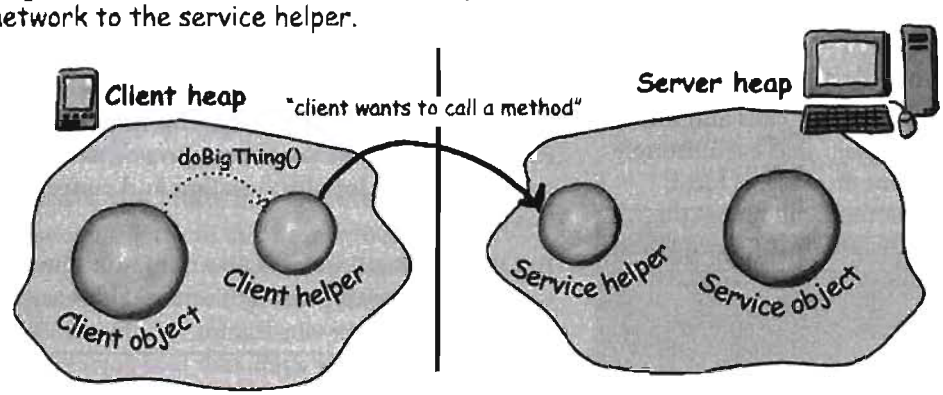


# How the method call happens

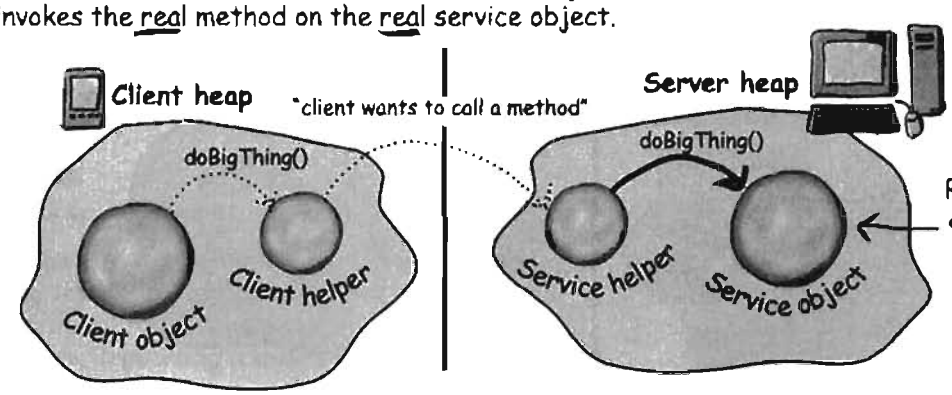
1. Client object calls doBigThing() on the client helper object




2. Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.




3. Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the real method on the real service object.



# Java RMI gives you the client and service helper objects!

In Java, RMI builds the client and service helper objects for you, and it even knows how to make the client helper look like the Real Service. In other words, RMI knows how to give the client helper object the same methods you want to call on the remote service.

Plus, RMI provides all the runtime infrastructure to make it work, including a lookup service so that the client can find and get the client helper (the proxy for the Real Service).

With RMI, you don't write *any* of the networking or I/O code yourself. The client gets to call remote methods (i.e. the ones the Real Service has) just like normal method calls on objects running in the client's own local JVM.

Almost.

There is one difference between RMI calls and local (normal) method calls. Remember that even though to the client it looks like the method call is local, the client helper sends the method call across the network. So there is networking and I/O. And what do we know about networking and I/O methods?

They're risky!

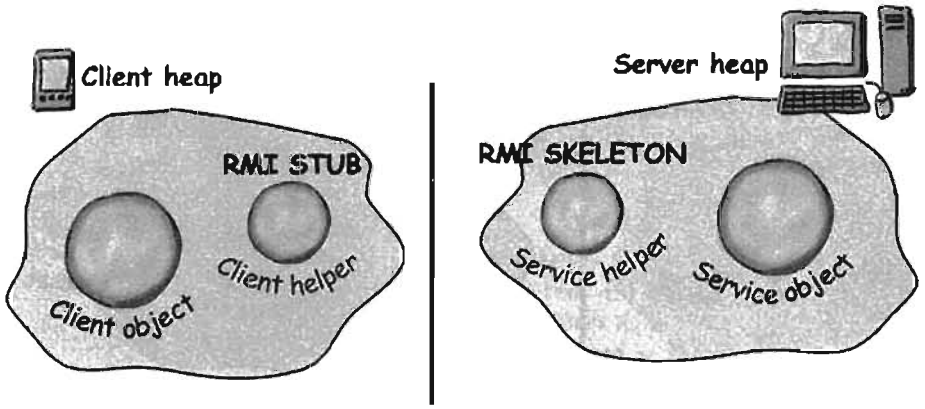
They throw exceptions all over the place.

So, the client does have to acknowledge the risk. The client has to acknowledge that when it calls a remote method, even though to the client it's just a local call to the proxy/helper object, the call *ultimately* involves Sockets and streams. The client's original call is *local*, but the proxy turns it into a *remote* call. A remote call just means a method that's invoked on an object on another JVM. *How* the information about that call gets transferred from one JVM to another depends on the protocol used by the helper objects.

With RMI, you have a choice of protocols: JRMP or IIOP. JRMP is RMI's 'native' protocol, the one made just for Java-to-Java remote calls. IIOP, on the other hand, is the protocol for CORBA (Common Object Request Broker Architecture), and lets you make remote calls on things which aren't necessarily Java objects. CORBA is usually *much* more painful than RMI, because if you don't have Java on both ends, there's an awful lot of translation and conversion that has to happen.

But thankfully, all we care about is Java-to-Java, so we're sticking with plain old, remarkably easy RMI.

## In RMI, the client helper is a 'stub' and the server helper is a 'skeleton'.



# Making the Remote Service

This is an overview of the five steps for making the remote service (that runs on the server). Don't worry, each step is explained in detail over the next few pages.


## Step one:

### Make a Remote Interface

The remote interface defines the methods that a client can call remotely. It's what the client will use as the polymorphic class type for your service. Both the Stub and actual service will implement this!


This interface defines the remote methods that you want clients to call

## Step two:

### Make a Remote Implementation

This is the class that does the Real Work. It has the real implementation of the remote methods defined in the remote interface. It's the object that the client ~~wants to call methods on~~.


The Real Service. The class with the methods that do the real work. It implements the remote interface.

## Step three:

### Generate the stubs and skeletons using rmic

These are the client and server 'helpers'. You don't have to create these classes or ever look at the source code that generates them. It's all handled automatically when you run the rmic tool that ships with your Java development kit.

Running rmic against the actual service implementation class...

```
File Edit Window Help Eat
%rmic MyRemoteImpl
```

spits out two new classes for the helper objects



## Step four:

### Start the RMI registry (rmiregistry)

The *rmiregistry* is like the white pages of a phone book. It's where the user goes to get the proxy (the client stub/helper object).

```
File Edit Window Help Drink
%rmiregistry
```

run this in a separate terminal

## Step five:

### Start the remote service

You have to get the service object up and running. Your service implementation class instantiates an instance of the service and registers it with the RMI registry. Registering it makes the service available for clients.

```
File Edit Window Help BeMerry
%java MyRemoteImpl
```

# Step one: Make a Remote Interface


MyRemote.java

### 1 Extend java.rmi.Remote

Remote is a 'marker' interface, which means it has no methods. It has special meaning for RMI, though, so you must follow this rule. Notice that we say 'extends' here. One interface is allowed to *extend* another interface.

```
public interface MyRemote extends Remote {
```

Your interface has to announce that it's for remote method calls. An interface can't implement anything, but it can extend other interfaces.

### 2 Declare that all methods throw a RemoteException

The remote interface is the one the client uses as the polymorphic type for the service. In other words, the client invokes methods on something that implements the remote interface. That something is the stub, of course, and since the stub is doing networking and I/O, all kinds of Bad Things can happen. The client has to acknowledge the risks by handling or declaring the remote exceptions. If the methods in an interface declare exceptions, any code calling methods on a reference of that type (the interface type) must handle or declare the exceptions.

```
import java.rmi.*;
```

the Remote interface is in java.rmi

```
public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```

Every remote method call is considered 'risky'. Declaring RemoteException on every method forces the client to pay attention and acknowledge that things might not work.

### 3 Be sure arguments and return values are primitives or Serializable

Arguments and return values of a remote method must be either primitive or Serializable. Think about it. Any argument to a remote method has to be packaged up and shipped across the network, and that's done through Serialization. Same thing with return values. If you use primitives, Strings, and the majority of types in the API (including arrays and collections), you'll be fine. If you are passing around your own types, just be sure that you make your classes implement Serializable.

```
public String sayHello() throws RemoteException;
```

This return value is gonna be shipped over the wire from the server back to the client, so it must be Serializable. That's how args and return values get packaged up and sent.

# Step two: Make a Remote Implementation

MyRemoteImpl.java

### Implement the Remote interface

Your service has to implement the remote interface—the one with the methods your client is going to call.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    // more code in class
}
```

The compiler will make sure that you've implemented all the methods from the interface you implement. In this case, there's only one.

### Extend UnicastRemoteObject

In order to work as a remote service object, your object needs some functionality related to 'being remote'. The simplest way is to extend UnicastRemoteObject (from the java.rmi.server package) and let that class (your superclass) do the work for you.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
```

### Write a no-arg constructor that declares a RemoteException

Your new superclass, UnicastRemoteObject, has one little problem—its constructor throws a RemoteException. The only way to deal with this is to declare a constructor for your remote implementation, just so that you have a place to declare the RemoteException. Remember, when a class is instantiated, its superclass constructor is always called. If your superclass constructor throws an exception, you have no choice but to declare that your constructor also throws an exception.

```
public MyRemoteImpl() throws RemoteException { }
```

You don't have to put anything in the constructor. You just need a way to declare that your superclass constructor throws an exception.

### Register the service with the RMI registry

Now that you've got a remote service, you have to make it available to remote clients. You do this by instantiating it and putting it into the RMI registry (which must be running or this line of code fails). When you register the implementation object, the RMI system actually puts the *stub* in the registry, since that's what the client really needs. Register your service using the static rebind() method of the java.rmi.Naming class.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("Remote Hello", service);
} catch(Exception ex) {...}
```

Give your service a name (that clients can use to look it up in the registry) and register it with the RMI registry. When you bind the service object, RMI swaps the service for the stub and puts the stub in the registry.

# Step three: generate stubs and skeletons

**1 Run rmic on the remote implementation class (not the remote interface)**

The rmic tool, that comes with the Java software development kit, takes a service implementation and creates two new classes, the stub and the skeleton. It uses a naming convention that is the name of your remote implementation, with either _Stub or _Skeleton added to the end. There are other options with rmic, including not generating skeletons, seeing what the source code for these classes looked like, and even using IIOP as the protocol. The way we're doing it here is the way you'll usually do it. The classes will land in the current directory (i.e. whatever you did a cd to). Remember, *rmic* must be able to see your implementation class, so you'll probably run rmic from the directory where your remote implementation is. (We're deliberately not using packages here, to make it simpler. In the Real World, you'll need to account for package directory structures and fully-qualified names).

Notice that you don't say ".class" on the end. Just the class name.

```
File Edit Window Help Whuffle
%rmic MyRemoteImpl
```

spits out two new classes for the helper objects

MyRemoteImpl_Stub.class

MyRemoteImpl_Skel.class

# Step four: run rmiregistry

**1 Bring up a terminal and start the rmiregistry.**

Be sure you start it from a directory that has access to your classes. The simplest way is to start it from your 'classes' directory.

```
File Edit Window Help Huh?
%rmiregistry
```

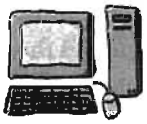
# Step five: start the service

**1 Bring up another terminal and start your service**

This might be from a main() method in your remote implementation class, or from a separate launcher class. In this simple example, we put the starter code in the implementation class, in a main method that instantiates the object and registers it with RMI registry.

```
File Edit Window Help Huh?
%java MyRemoteImpl
```

# Complete code for the server side




## The Remote interface:

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

RemoteException and Remote interface are in java.rmi package

Your interface MUST extend java.rmi.Remote

All of your remote methods must declare a RemoteException

---

## The Remote service (the implementation):

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

   public String sayHello() {
       return "Server says, 'Hey'";
   }

   public MyRemoteImpl() throws RemoteException { }

   public static void main (String[] args) {

       try {
           MyRemote service = new MyRemoteImpl();
           Naming.rebind("Remote Hello", service);
       } catch(Exception ex) {
           ex.printStackTrace();
       }
   }
}
```

UnicastRemoteObject is in the java.rmi.server package

extending UnicastRemoteObject is the easiest way to make a remote object

You have to implement all the interface methods, of course. But notice that you do NOT have to declare the RemoteException.

you MUST implement your remote interface!!

your superclass constructor (for UnicastRemoteObject) declares an exception, so YOU must write a constructor, because it means that your constructor is calling risky code (its super constructor)

Make the remote object, then 'bind' it to the rmiregistry using the static Naming.rebind(). The name you register it under is the name clients will need to look it up in the rmi registry.

# How does the client get the stub object?

The client has to get the stub object, since that's the thing the client will call methods on. And that's where the RMI registry comes in. The client does a 'lookup', like going to the white pages of a phone book, and essentially says, "Here's a name, and I'd like the stub that goes with that name."

lookup() is a static method of the Naming class

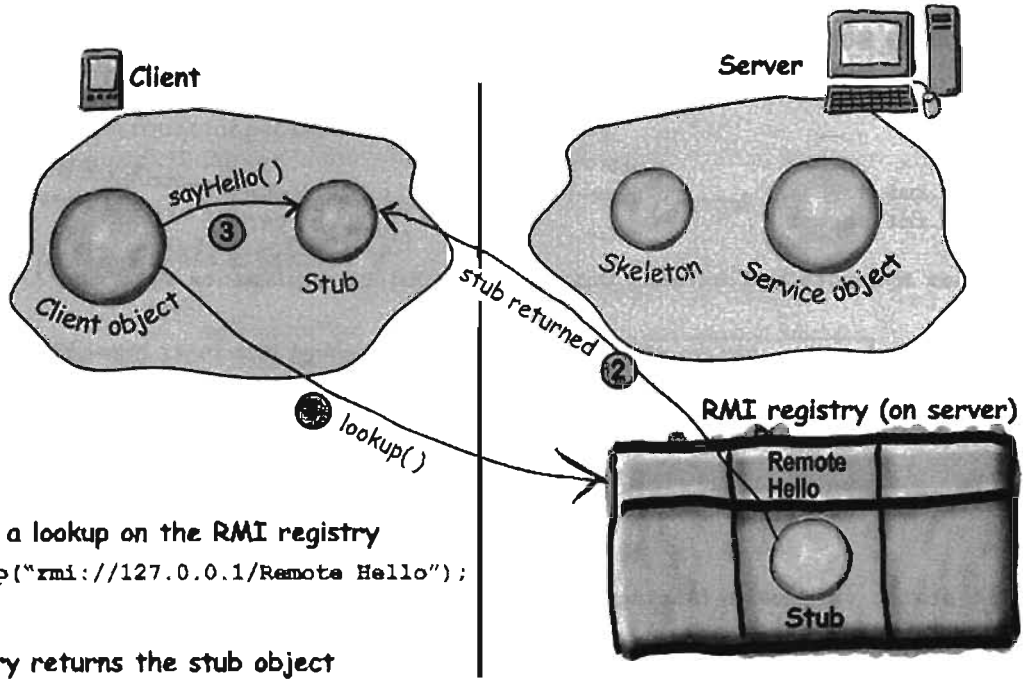
This must be the name that the service was registered under

```
MyRemote service = (MyRemote)  Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

The client always uses the remote implementation as the type of the service. In fact, the client never needs to know the actual class name of your remote service.

You have to cast it to the interface, since the lookup method returns type Object

your host name or IP address goes here




**1 Client does a lookup on the RMI registry**

```
Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

**2 RMI registry returns the stub object**
(as the return value of the lookup method) and RMI deserializes the stub automatically. You MUST have the stub class (that rmic generated for you) on the client or the stub won't be deserialized.

**3 Client invokes a method on the stub, as though the stub IS the real service**

# How does the client get the stub class?

Now we get to the interesting question. Somehow, someway, the client must have the stub class (that you generated earlier using rmic) at the time the client does the lookup, or else the stub won't be deserialized on the client and the whole thing blows up. In a simple system, you can simply hand-deliver the stub class to the client.

There's a much cooler way, though, although it's beyond the scope of this book. But just in case you're interested, the cooler way is called "dynamic class downloading". With dynamic class downloading, a stub object (or really any Serialized object) is 'stamped' with a URL that tells the RMI system on the client where to find the class file for that object. Then, in the process of deserializing an object, if RMI can't find the class locally, it uses that URL to do an HTTP Get to retrieve the class file. So you'd need a simple Web server to serve up class files, and you'd also need to change some security parameters on the client. There are a few other tricky issues with dynamic class downloading, but that's the overview.

## Complete client code

```
import java.rmi.*;

public class MyRemoteClient {
   public static void main (String[] args) {
        new MyRemoteClient().go();
   }

  public void go() {

     try {
        MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/Remote Hello");

        String s = service.sayHello();

        System.out.println(s);
      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

The Naming class (for doing the remiregistry lookup) is in the java.rmi package

It comes out of the registry as type Object, so don't forget the cast

you need the IP address or hostname

and the name used to bind/rebind the service

It looks just like a regular old method call! (Except it must acknowledge the RemoteException)

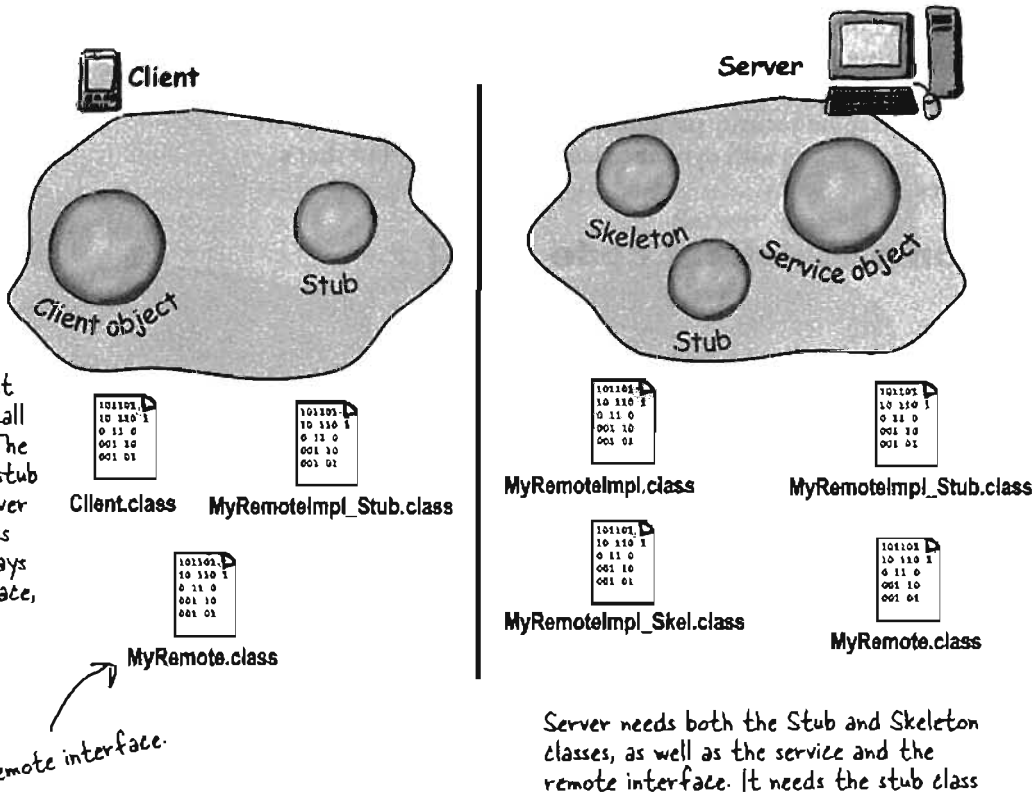
# Be sure each machine has the class files it needs.

The top three things programmers do wrong with RMI are:

1) Forget to start rmiregistry before starting remote service (when you register the service using Naming.rebind(), the rmiregistry must be running!)

2) Forget to make arguments and return types serializable (you won't know until runtime; this is not something the compiler will detect.)

3) Forget to give the stub class to the client.




Don't forget, the client uses the interface to call methods on the stub. The client JVM needs the stub class, but the client never refers to the stub class in code. The client always uses the remote interface, as though the remote interface WERE the actual remote object.

The remote interface.

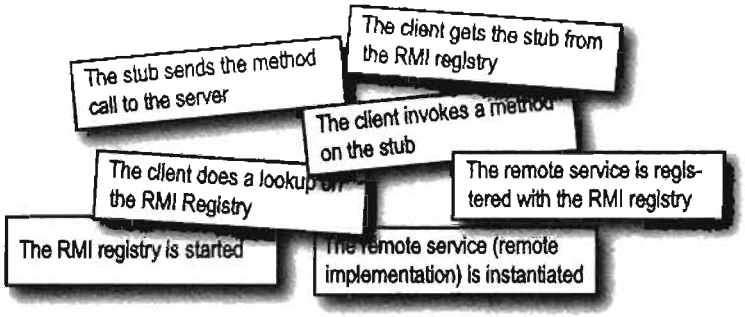
Server needs both the Stub and Skeleton classes, as well as the service and the remote interface. It needs the stub class because remember, the stub is substituted for the real service, when the real service is bound to the RMI registry.

Sharpen your pencil

## What's First?

Look at the sequence of events below, and place them in the order in which they occur in a Java RMI application.

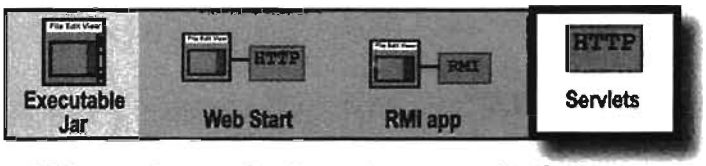
1. 
2. 
3. 
4. 
5. 
6. 
7. 




### BULLET POINTS

- An object on one heap cannot get a normal Java reference to an object on a different heap (which means running on a different JVM)
- Java Remote Method Invocation (RMI) makes it *seem* like you're calling a method on a remote object (i.e. an object in a different JVM), but you aren't.
- When a client calls a method on a remote object, the client is really calling a method on a *proxy* of the remote object. The proxy is called a 'stub'.
- A stub is a client helper object that takes care of the low-level networking details (sockets, streams, serialization, etc.) by packaging and sending method calls to the server.
- To build a remote service (in other words, an object that a remote client can ultimately call methods on), you must start with a remote interface.
- A remote interface must extend the java.rmi.Remote interface, and all methods must declare RemoteException.
- Your remote service implements your remote interface.
- Your remote service should extend UnicastRemoteObject. (Technically there are other ways to create a remote object, but extending UnicastRemoteObject is the simplest).
- Your remote service class must have a constructor, and the constructor must declare a RemoteException (because the superclass constructor declares one).
- Your remote service must be instantiated, and the object registered with the RMI registry.
- To register a remote service, use the static Naming.rebind("Service Name", serviceInstance);
- The RMI registry must be running on the same machine as the remote service, before you try to register a remote object with the RMI registry.
- The client looks up your remote service using the static Naming.lookup("rmi://MyHostName/ServiceName");
- Almost everything related to RMI can throw a RemoteException (checked by the compiler). This includes registering or looking up a service in the reigstry, and *all* remote method calls from the client to the stub.

# Yeah, but who really *uses* RMI?



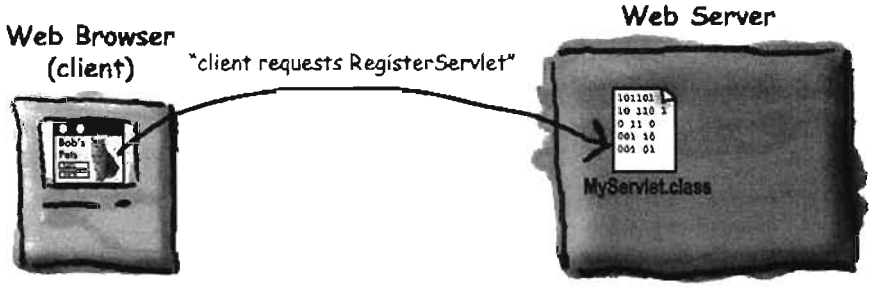

# What about Servlets?

Servlets are Java programs that run on (and with) an HTTP web server. When a client uses a web browser to interact with a web page, a request is sent back to the web server. If the request needs the help of a Java servlet, the web server runs (or calls, if the servlet is already running) the servlet code. Servlet code is simply code that runs on the server, to do work as a result of whatever the client requests (for example, save information to a text file or database on the server). If you're familiar with CGI scripts written in Perl, you know exactly what we're talking about. Web developers use CGI scripts or servlets to do everything from sending user-submitted info to a database, to running a web-site's discussion board.

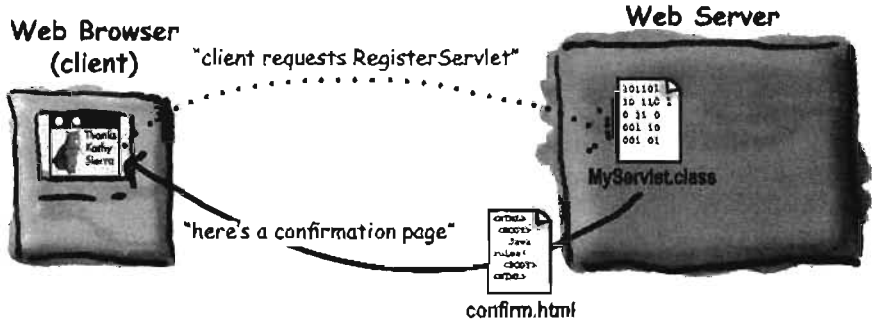
*And even servlets can use RMI!*

By far, the most common use of J2EE technology is to mix servlets and EJBs together, where servlets are the client of the EJB. And in that case, *the servlet is using RMI to talk to the EJBs.* (Although the way you use RMI with EJB is a *little* different from the process we just looked at.)

1. Client fills out a registration form and clicks 'submit'. The HTTP server (i.e. web server) gets the request, sees that it's for a servlet, and sends the request to the servlet.



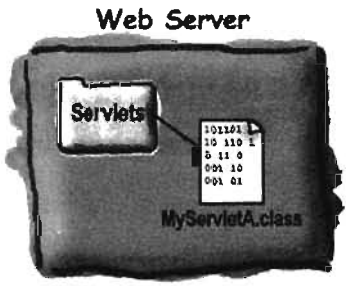

2. Servlet (Java code) runs, adds data to the database, composes a web page (with custom info) and sends it back to the client where it displays in the browser.



# Step for making and running a servlet

### ① Find out where your servlets need to be placed.

For these examples, we'll assume that you already have a web server up and running, and that it's already configured to support servlets. The most important thing is to find out exactly where your servlet class files have to be placed in order for your server to 'see' them. If you have a web site hosted by an ISP, the hosting service can tell you where to put your servlets, just as they'll tell you where to place your CGI scripts.




### ② Get the servlets.jar and add it to your classpath

Servlets aren't part of the standard Java libraries; you need the servlets classes packaged into the servlets.jar file. You can download the servlets classes from java.sun.com, or you can get them from your Java-enabled web server (like Apache Tomcat, at the apache.org site). Without these classes, you won't be able to compile your servlets.


### ③ Write a servlet class by extending HttpServlet

A servlet is just a Java class that extends HttpServlet (from the javax.servlet.http package). There are other types of servlets you can make, but most of the time we care only about HttpServlet.


```
public class MyServletA extends HttpServlet { ... }
```

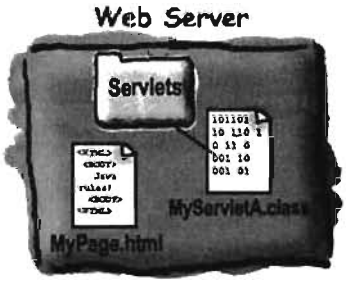
### ④ Write an HTML page that invokes your servlet

When the user clicks a link that references your servlet, the web server will find the servlet and invoke the appropriate method depending on the HTTP command (GET, POST, etc.)


```
<a href="servlets/MyServletA">This is the most amazing servlet.</a>
```

### ⑤ Make your servlet and HTML page available to your server

This is completely dependent on your web server (and more specifically, on which *version* of Java Servlets that you're using). Your ISP may simply tell you to drop it into a "Servlets" directory on your web site. But if you're using, say, the latest version of Tomcat, you'll have a lot more work to do to get the servlet (and web page) into the right location. (We just happen to have a book on this too .)



## BULLET POINTS

- Servlets are Java classes that run entirely on (and/or within) an HTTP (web) server.
- Servlets are useful for running code on the server as a result of client interaction with a web page. For example, if a client submits information in a web page form, the servlet can process the information, add it to a database, and send back a customized, confirmation response page.
- To compile a servlet, you need the servlet packages which are in the servlets.jar file. The servlet classes are not part of the Java standard libraries, so you need to download the servlets.jar from java.sun.com or get them from a servlet-capable web server. (Note: the Servlet library is included with the Java 2 Enterprise Edition (J2EE))
- To run a servlet, you must have a web server capable of running servlets, such as the Tomcat server from apache.org.
- Your servlet must be placed in a location that's specific to your particular web server, so you'll need to find that out before you try to run your servlets. If you have a web site hosted by an ISP that supports servlets, the ISP will tell you which directory to place your servlets in.
- A typical servlet extends HttpServlet and overrides one or more servlet methods, such as doGet() or doPost().
- The web server starts the servlet and calls the appropriate method (doGet(), etc.) based on the client's request.
- The servlet can send back a response by getting a PrintWriter output stream from the response parameter of the doGet() method.
- The servlet 'writes' out an HTML page, complete with tags).

## there are no Dumb Questions

**Q: What's a JSP, and how does it relate to servlets?**

A: JSP stands for Java Server Pages. In the end, the web server turns a JSP into a servlet, but the difference between a servlet and a JSP is what YOU (the developer) actually create. With a servlet, you write a Java *class* that contains *HTML* in the output statements (if you're sending back an HTML page to the client). But with a JSP, it's the opposite—you write an *HTML* page that contains *Java* code!

This gives you the ability to have dynamic web pages where you write the page as a normal HTML page, except you embed Java code (and other tags that "trigger" Java code at runtime) that gets processed at runtime. In other words, part of the page is customized at runtime when the Java code runs.

The main benefit of JSP over regular servlets is that it's just a lot easier to write the HTML part of a servlet as a JSP page than to write HTML in the torturous print out statements in the servlet's response. Imagine a reasonably complex HTML page, and now imagine formatting it within println statements. Yikes!

But for many applications, it isn't necessary to use JSPs because the servlet doesn't need to send a dynamic response, or the HTML is simple enough not to be such a big pain. And, there are still many web servers out there that support servlets but do not support JSPs, so you're stuck.

Another benefit of JSPs is that you can separate the work by having the Java developers write the servlets and the web page developers write the JSPs. That's the promised benefit, anyway. In reality, there's still a Java learning curve (and a tag learning curve) for anyone writing a JSP, so to think that an HTML web page designer can bang out JSPs is not realistic. Well, not without tools. But that's the good news—authoring tools are starting to appear, that help web page designers create JSPs without writing the code from scratch.

**Q: Is this all you're gonna say about servlets? After such a *huge* thing on RMI?**

A: Yes. RMI is part of the Java language, and all the classes for RMI are in the standard libraries. Servlets and JSPs are *not* part of the Java language; they're considered *standard extensions*. You can run RMI on any modern JVM, but Servlets and JSPs require a properly configured web server with a servlet "container". This is our way of saying, "it's beyond the scope of this book." But you can read much more in the lovely *Head First Servlets & JSP*.

# Just for fun, let's make the Phrase-O-Matic work as a servlet

Now that we told you that we won't say any more about servlets, we can't resist servletizing (yes, we *can* verbify it) the Phrase-O-Matic from chapter 1. A servlet is still just Java. And Java code can call Java code from other classes. So a servlet is free to call a method on the Phrase-O-Matic. All you have to do is drop the Phrase-O-Matic class into the same directory as your servlet, and you're in business. (The Phrase-O-Matic code is on the next page).


```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class KathyServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException  {

        String title = "PhraseOMatic has generated the following phrase.";

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println("PhraseOmatic");
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>" + PhraseOMatic.makePhrase());
        out.println("<P><a href=\"KathyServlet\">make another phrase</a></p>");
        out.println("</BODY></HTML>");

        out.close();
    }
}
```

See? Your servlet can call methods on another class. In this case, we're calling the static makePhrase() method of the PhraseOMatic class (on the next page)

# Phrase-O-Matic code, servlet-friendly

This is a slightly different version from the code in chapter one. In the original, we ran the entire thing in a main() method, and we had to rerun the program each time to generate a new phrase at the command-line. In this version, the code simply returns a String (with the phrase) when you invoke the static makePhrase() method. That way, you can call the method from any other code and get back a String with the randomly-composed phrase.

Please note that these long String[] array assignments are a victim of word-processing here—don't type in the hyphens! Just keep on typing and let your code editor do the wrapping. And whatever you do, don't hit the return key in the middle of a String (i.e. something between double quotes).

```
public class PhraseOMatic {
   public static String makePhrase() {

      // make three sets of words to choose from
      String[] wordListOne = {"24/7","multi-Tier","30,000 foot","B-to-B","win-win","front-
end", "web-based","pervasive", "smart", "six-sigma","critical-path", "dynamic"};

      String[] wordListTwo = {"empowered", "sticky", "valued-added", "oriented", "centric",
"distributed", "clustered", "branded","outside-the-box", "positioned", "networked", "fo-
cused", "leveraged", "aligned", "targeted", "shared", "cooperative", "accelerated"};

      String[] wordListThree = {"process", "tipping point", "solution", "architecture",
"core competency", "strategy", "mindshare", "portal", "space", "vision", "paradigm", "mis-
sion"};

     // find out how many words are in each list
     int oneLength = wordListOne.length;
     int twoLength = wordListTwo.length;
     int threeLength = wordListThree.length;

     // generate three random numbers, to pull random words from each list
     int rand1 = (int) (Math.random() * oneLength);
     int rand2 = (int) (Math.random() * twoLength);
     int rand3 = (int) (Math.random() * threeLength);

     // now build a phrase
     String phrase = wordListOne[rand1] + " " + wordListTwo[rand2] + " " +
wordListThree[rand3];

     // now return it
     return ("What we need is a " + phrase);
   }
}
```

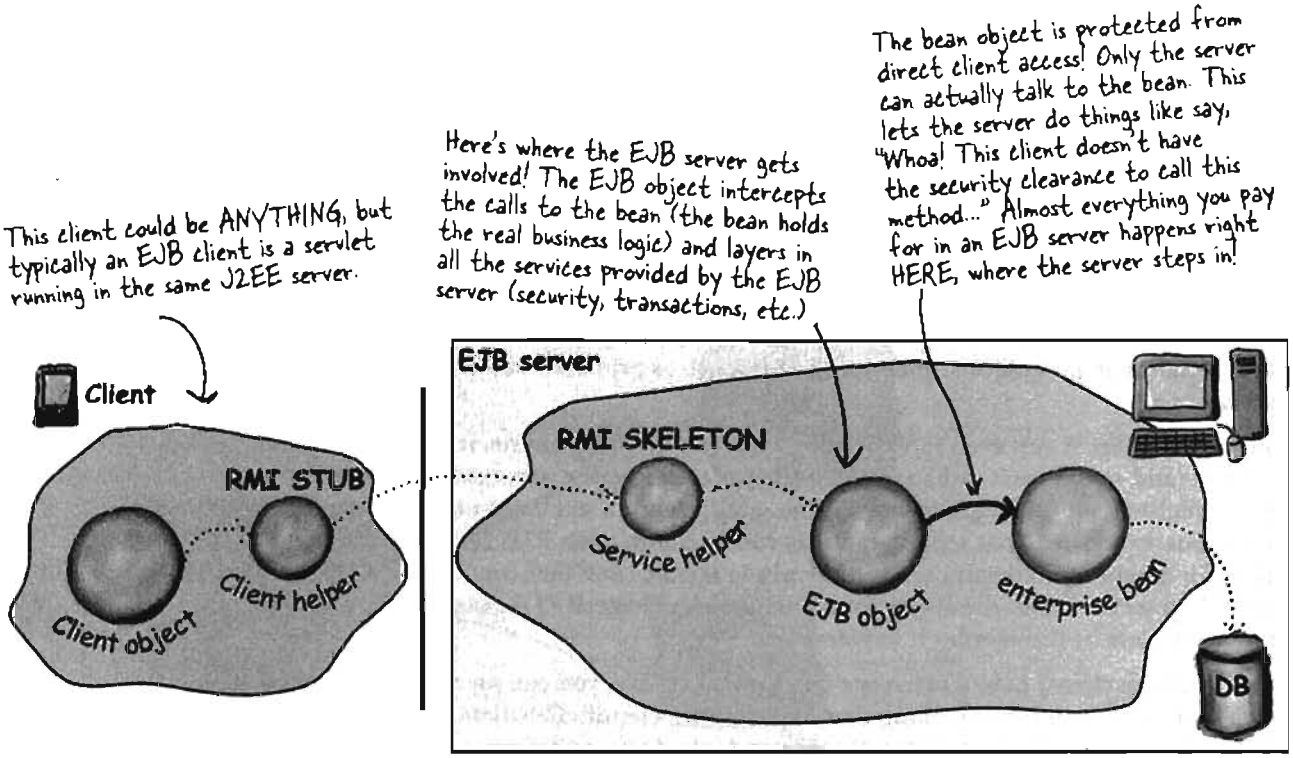
# Enterprise JavaBeans: RMI on steroids

RMI is great for writing and running remote services. But you wouldn't run something like an Amazon or eBay on RMI alone. For a large, deadly serious, enterprise application, you need something more. You need something that can handle transactions, heavy concurrency issues (like a gazillion people are hitting your server at once to buy those organic dog kibbles), security (not just anyone should hit your payroll database), and data management. For that, you need an *enterprise application server.*

In Java, that means a Java 2 Enterprise Edition (J2EE) server. A J2EE server includes both a web server and an Enterprise JavaBeans (EJB) server, so that you can deploy an application that includes both servlets and EJBs. Like servlets, EJB is way beyond the scope of this book, and there's no way to show "just a little" EJB example with code, but we *will* take a quick look at how it works. (For a much more detailed treatment of EJB, we can recommend the lively Head First EJB certification study guide.)

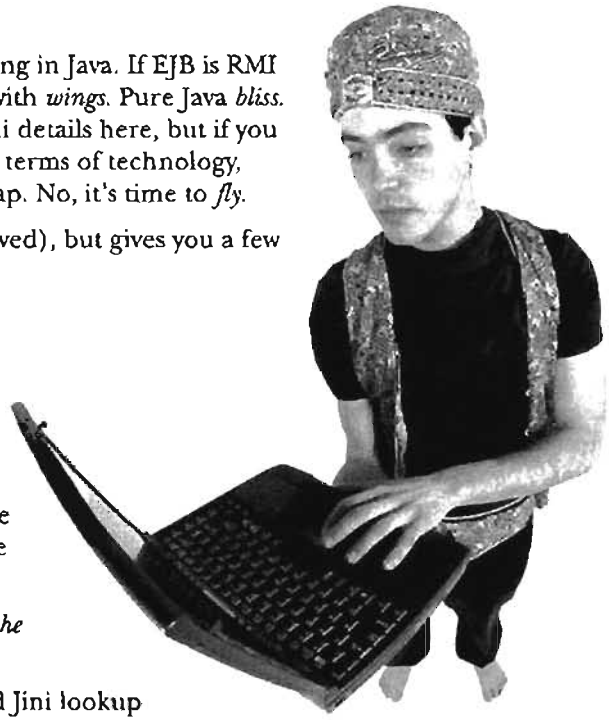
An EJB server adds a bunch of services that you don't get with straight RMI. Things like transactions, security, concurrency, database management, and networking.

An EJB server steps into the middle of an RMI call and layers in all of the services.




This is only a small part of the EJB picture!

# For our final trick... a little Jini



We love Jini. We think Jini is pretty much the best thing in Java. If EJB is RMI on steroids (with a bunch of managers), Jini is RMI with *wings*. Pure Java *bliss*. Like the EJB material, we can't get into any of the Jini details here, but if you know RMI, you're three-quarters of the way there. In terms of technology, anyway. In terms of *mindset*, it's time to make a big leap. No, it's time to *fly*.

Jini uses RMI (although other protocols can be involved), but gives you a few key features including:

***Adaptive discovery***

***Self-healing networks***

With RMI, remember, the client has to know the name and location of the remote service. The client code for the lookup includes the IP address or hostname of the remote service (because that's where the RMI registry is running) *and* the logical name the service was registered under.

But with Jini, the client has to know only one thing: ***the interface implemented by the service!*** That's it.

So how do you find things? The trick revolves around Jini lookup services. Jini lookup services are far more powerful and flexible than the RMI registry. For one thing, Jini lookup services announce themselves to the network, *automatically*. When a lookup service comes online, it sends a message (using IP multicast) out to the network saying, "I'm here, if anyone's interested."

But that's not all. Let's say you (a client) come online *after* the lookup service has already announced itself, *you* can send a message to the entire network saying, "Are there any lookup services out there?"

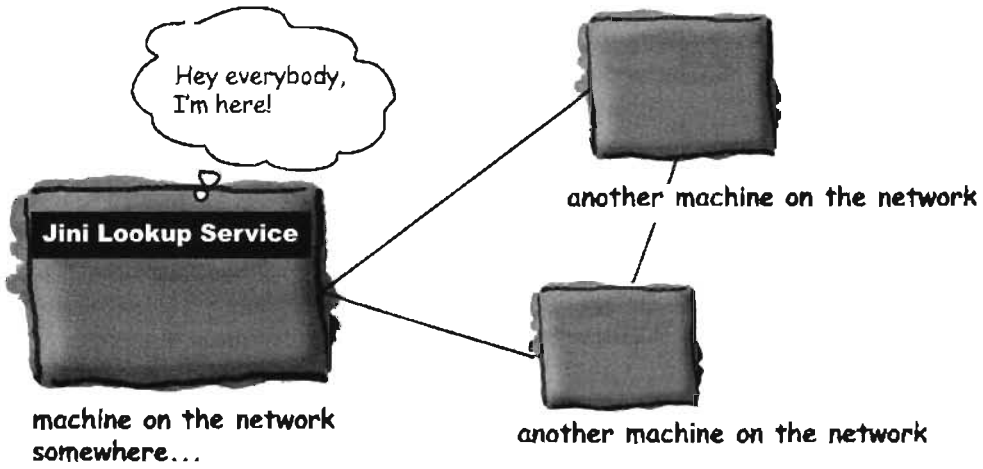
Except that you're not really interested in the lookup service *itself*—you're interested in the services that are *registered* with the lookup service. Things like RMI remote services, other serializable Java objects, and even devices such as printers, cameras, and coffee-makers.

And here's where it gets even more fun: when a service comes online, it will dynamically discover (and *register* itself with) any Jini lookup services on the network. When the service registers with the lookup service, the service sends a serialized object to be placed in the lookup service. That serialized object can be a stub to an RMI remote service, a driver for a networked device, or even the whole service itself that (once you get it from the lookup service) runs locally on your machine. And instead of registering by *name*, the service registers by the *interface* it implements.

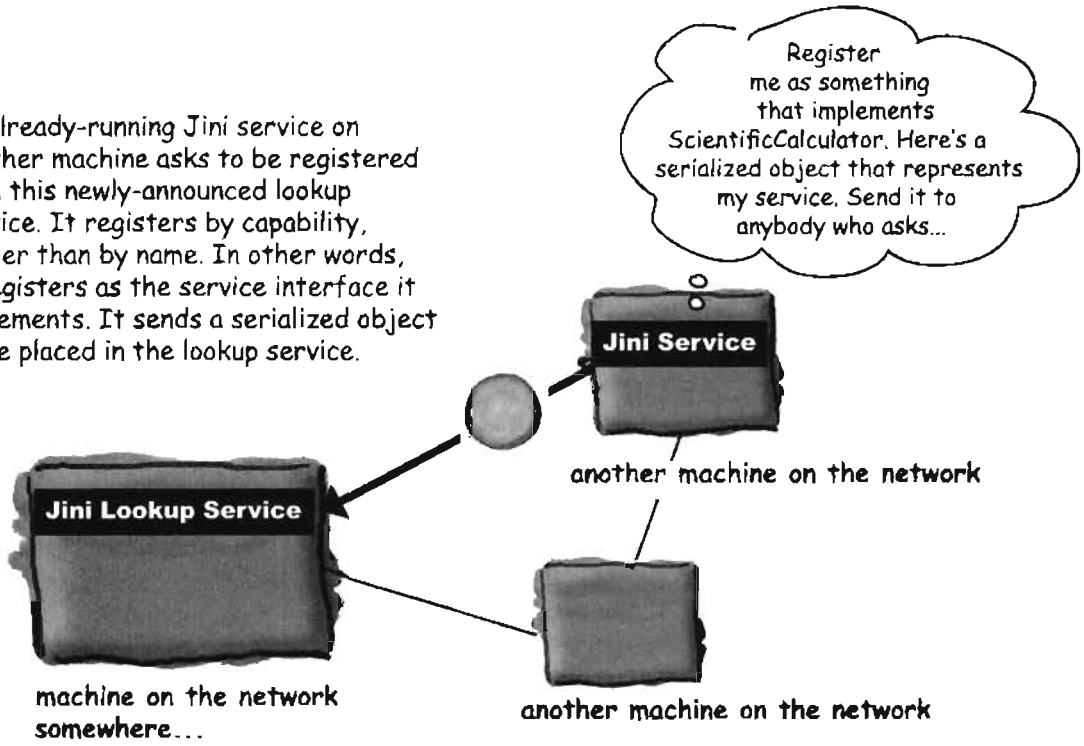
Once you (the client) have a reference to a lookup service, you can say to that lookup service, "Hey, do you have anything that implements ScientificCalculator?" At that point, the lookup service will check its list of registered interfaces, and assuming it finds a match, says back to you, "Yes I *do* have something that implements that interface. Here's the serialized object the ScientificCalculator service registered with me."

# Adaptive discovery in action

1. Jini lookup service is launched somewhere on the network, and announces itself using IP multicast.

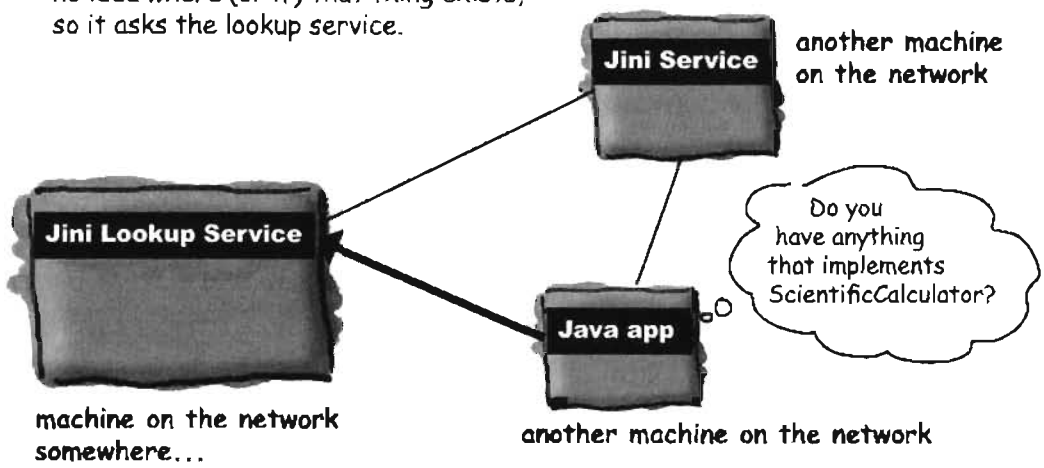



2. An already-running Jini service on another machine asks to be registered with this newly-announced lookup service. It registers by capability, rather than by name. In other words, it registers as the service interface it implements. It sends a serialized object to be placed in the lookup service.

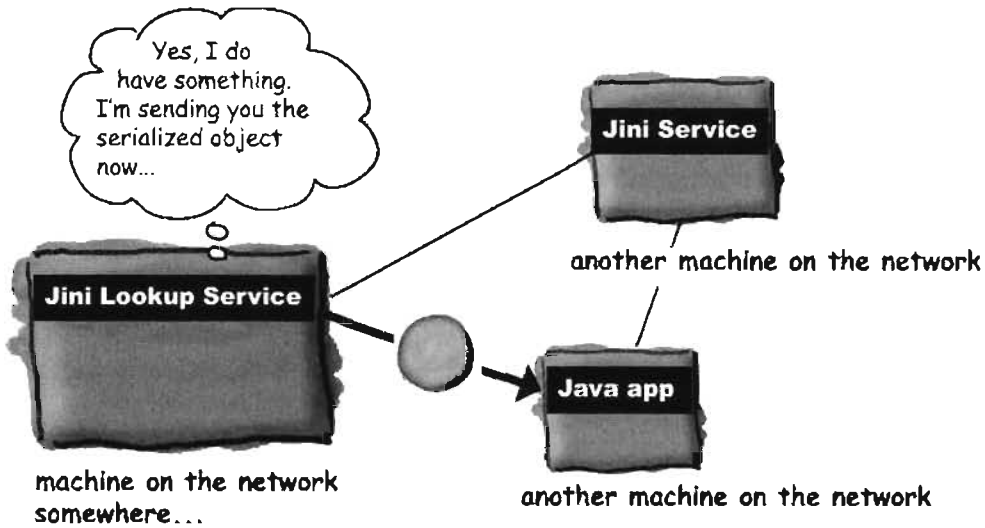


# Adaptive discovery in action, continued...

③ A client on the network wants something that implements the ScientificCalculator interface. It has no idea where (or if) that thing exists, so it asks the lookup service.

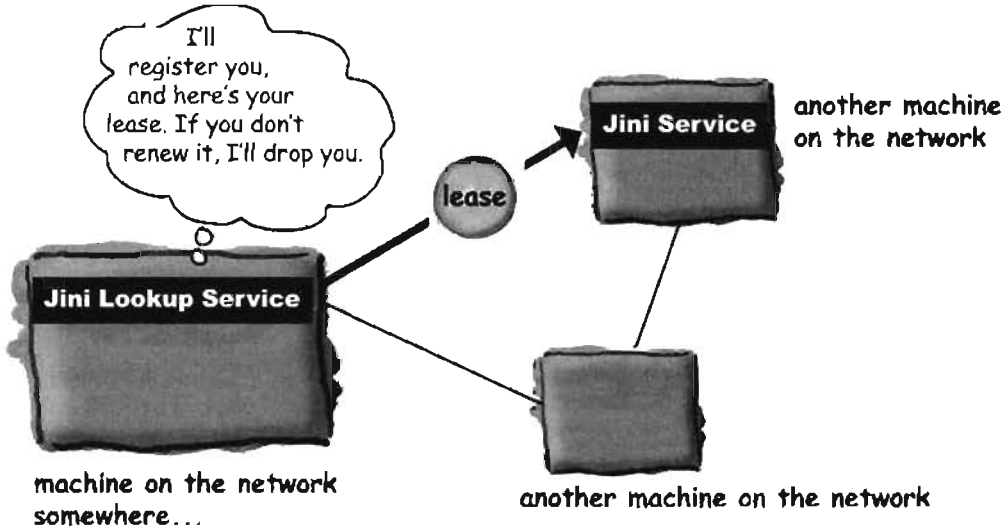



④ The lookup service responds, since it does have something registered as a ScientificCalculator interface.

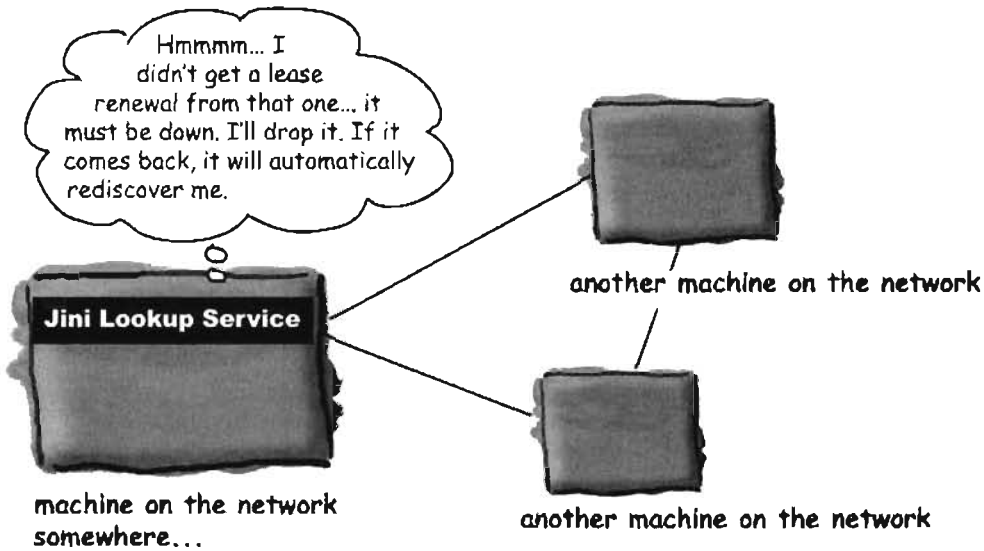


# Self-healing network in action

① A Jini Service has asked to register with the lookup service. The lookup service responds with a "lease". The newly-registered service must keep renewing the lease, or the lookup service assumes the service has gone offline. The lookup service wants always to present an accurate picture to the rest of the network about which services are available.




② The service goes offline (somebody shuts it down), so it fails to renew its lease with the lookup service. The lookup service drops it.



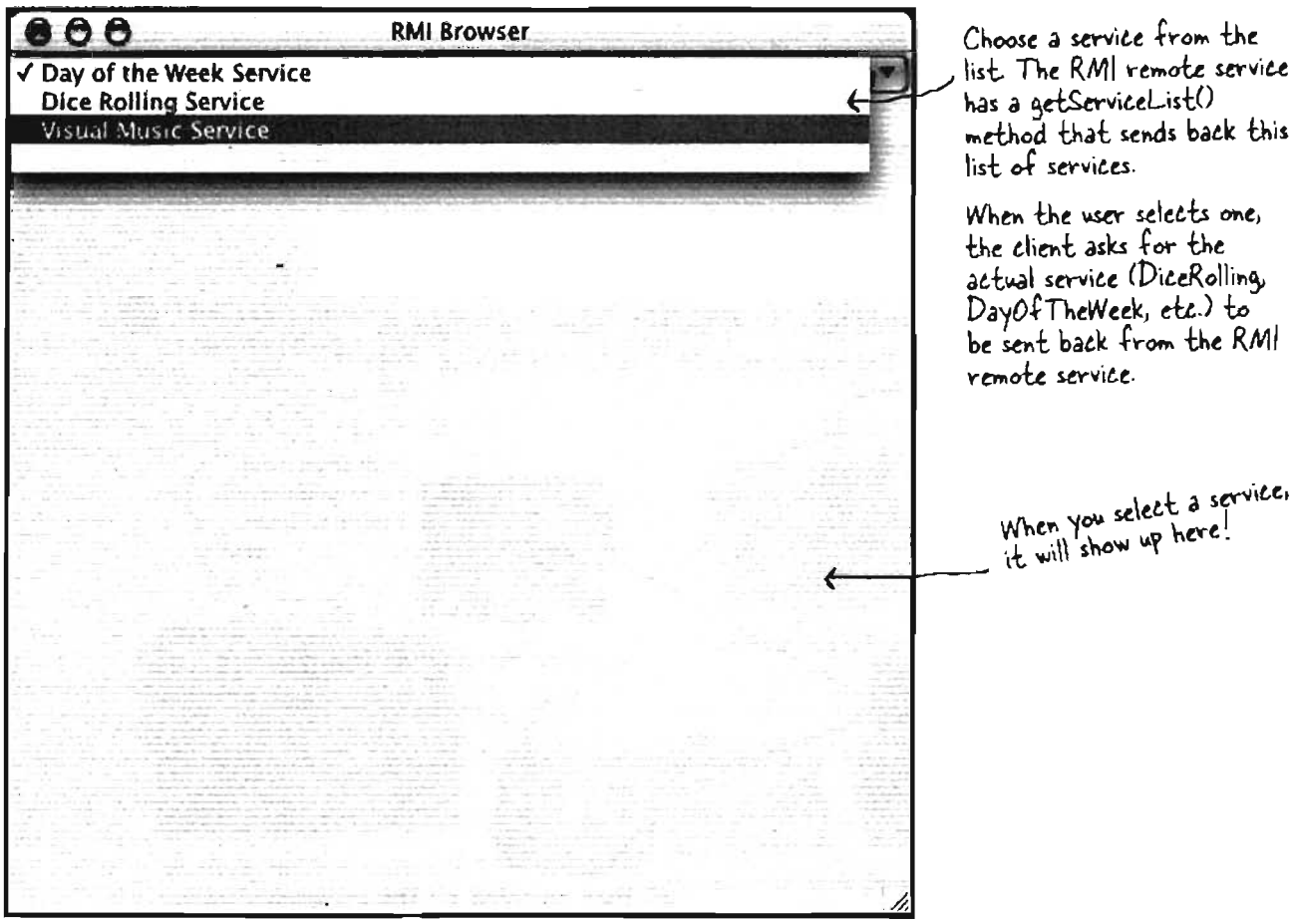
# Final Project: the Universal Service browser

We're going to make something that isn't Jini-enabled, but quite easily could be. It will give you the flavor and feeling of Jini, but using straight RMI. In fact the main difference between our application and a Jini application is how the service is discovered. Instead of the Jini lookup service, which automatically announces itself and lives anywhere on the network, we're using the RMI registry which must be on the same machine as the remote service, and which does not announce itself automatically.

And instead of our service registering itself automatically with the lookup service, *we* have to register it in the RMI registry (using Naming.rebind()).

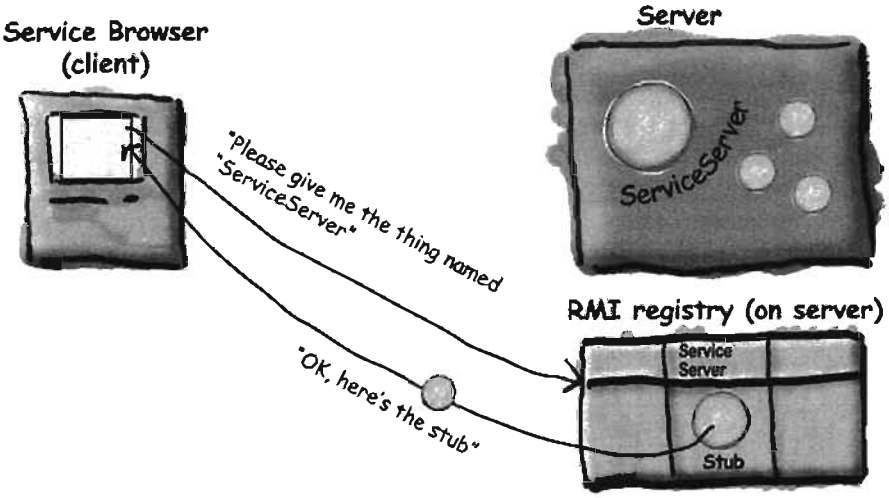
But once the client has found the service in the RMI registry, the rest of the application is almost identical to the way we'd do it in Jini. (The main thing missing is the *lease* that would let us have a self-healing network if any of the services go down.)

The universal service browser is like a specialized web browser, except instead of HTML pages, the service browser downloads and displays interactive Java GUIs that we're calling *universal services.*

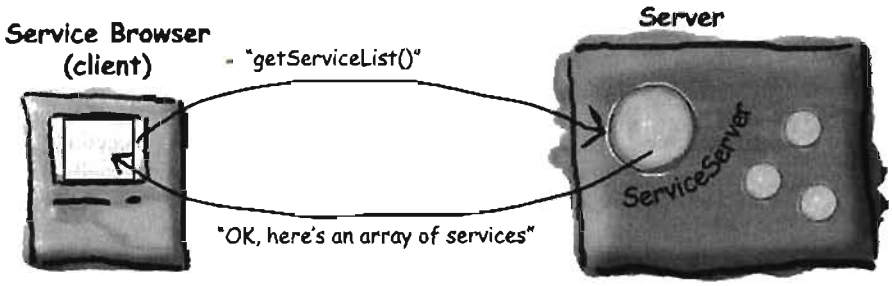


## How it works:

1. Client starts up and does a lookup on the RMI registry for the service called "ServiceServer", and gets back the stub.



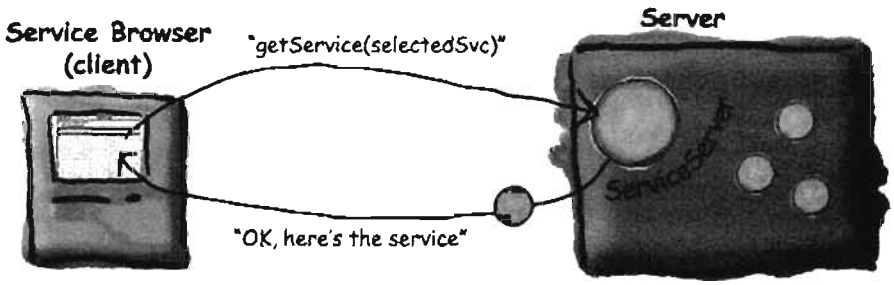

2. Client calls getServiceList() on the stub. The ServiceServer returns an array of services




3. Client displays the list of services in a GUI

## How it works, continued...

1. User selects from the list, so client calls the getService() method on the remote service. The remote service returns a serialized object that is an actual service that will run inside the client browser.




2. Client calls the getGuiPanel() on the serialized service object it just got from the remote service. The GUI for that service is displayed inside the browser, and the user can interact with it locally. At this point, we don't need the remote service unless/until the user decides to select another service.

## The classes and interfaces:

**1 interface ServiceServer implements Remote**
A regular old RMI remote interface for the remote service (the remote service has the method for getting the service list and returning a selected service).

**2 class ServiceServerImpl implements ServiceServer**
The actual RMI remote service (extends UnicastRemoteObject). Its job is to instantiate and store all the services (the things that will be shipped to the client), and register the server itself (ServiceServerImpl) with the RMI registry.

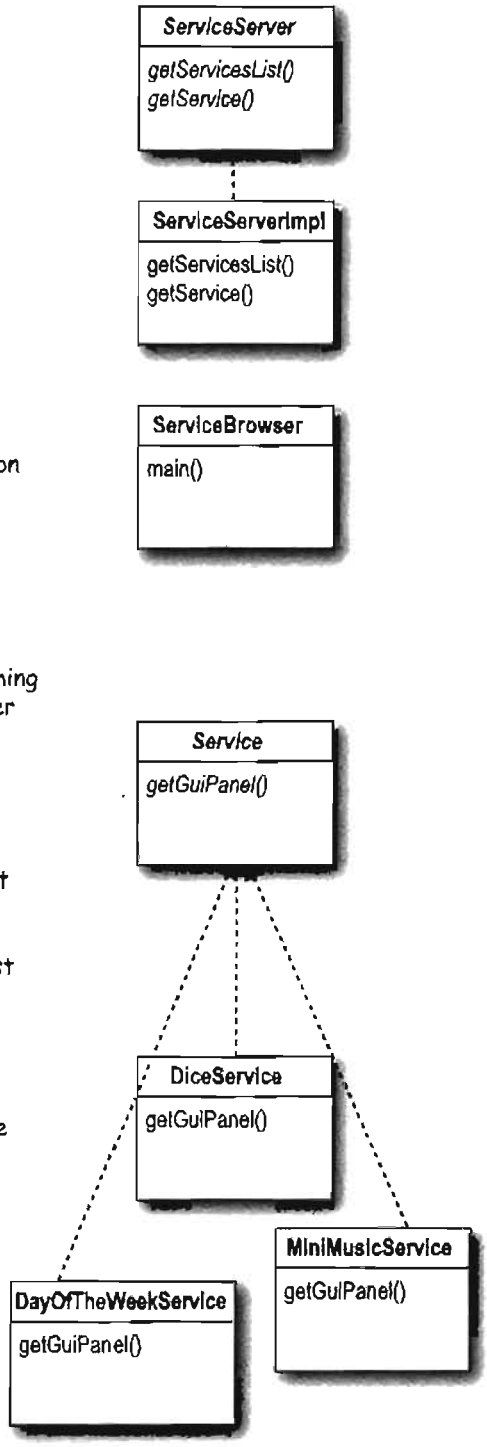
**3 class ServiceBrowser**
The client. It builds a very simple GUI, does a lookup in the RMI registry to get the ServiceServer stub, then calls a remote method on it to get the list of services to display in the GUI list.

**4 interface Service**
This is the key to everything. This very simple interface has just one method, getGuiPanel(). Every service that gets shipped over to the client must implement this interface. This is what makes the whole thing UNIVERSAL! By implementing this interface, a service can come over even though the client has no idea what the actual class (or classes) are that make up that service. All the client knows is that whatever comes over, it implements the Service interface, so it MUST have a getGuiPanel() method.
The client gets a serialized object as a result of calling getService(selectedSvc) on the ServiceServer stub, and all the client says to that object is, "I don't know who or what you are, but I DO know that you implement the Service interface, so I know I can call getGuiPanel() on you. And since getGuiPanel() returns a JPanel, I'll just slap it into the browser GUI and start interacting with it!

**5 class DiceService implements Service**
Got dice? If not, but you need some, use this service to roll anywhere from 1 to 6 virtual dice for you.

**6 class MiniMusicService implements Service**
Remember that fabulous little 'music video' program from the first GUI Code Kitchen? We've turned it into a service, and you can play it over and over and over until your roommates finally leave.

**7 class DayOfTheWeekService implements Service**
Were you born on a Friday? Type in your birthday and find out.



universal service code

## interface ServiceServer (the remote interface)

```
import java.rmi.*;

public interface ServiceServer extends Remote {

    Object[] getServiceList() throws RemoteException;

    Service getService(Object serviceKey) throws RemoteException;
}
```

A normal RMI remote interface, defines the two methods the remote service will have.

## interface Service (what the GUI services implement)

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

public interface Service extends Serializable {
    public JPanel getGuiPanel();
}
```

A plain old (i.e. non-remote) interface, that defines the one method that any universal service must have—getGuiPanel(). The interface extends Serializable, so that any class implementing the Service interface will automatically be Serializable.

That's a must, because the services get shipped over the wire from the server, as a result of the client calling getService() on the remote ServiceServer.

## class ServiceServerImpl (the remote implementation)

```
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;

public class ServiceServerImpl extends UnicastRemoteObject implements ServiceServer  {

    HashMap serviceList;

    public ServiceServerImpl() throws RemoteException {
       setUpServices();
    }

    private void setUpServices() {
       serviceList = new HashMap();
       serviceList.put("Dice Rolling Service", new DiceService());
       serviceList.put("Day of the Week Service", new DayOfTheWeekService());
       serviceList.put("Visual Music Service", new MiniMusicService());
    }

    public Object[] getServiceList() {
       System.out.println("in remote");
       return serviceList.keySet().toArray();
    }

    public Service getService(Object serviceKey) throws RemoteException {
       Service theService = (Service) serviceList.get(serviceKey);
       return theService;
    }

    public static void main (String[] args) {
       try {
         Naming.rebind("ServiceServer", new ServiceServerImpl());
       } catch(Exception ex) {
           ex.printStackTrace();
      }
       System.out.println("Remote service is running");
    }
```

A normal RMI implementation

The services will be stored in a HashMap collection. Instead of putting ONE object in the collection, you put TWO -- a key object (like a String) and a value object (whatever you want). (see appendix B for more on HashMap)

When the constructor is called, initialize the actual universal services (DiceService, MiniMusicService, etc.)

Make the services (the actual service objects) and put them into the HashMap, with a String name (for the 'key').

Client calls this in order to get a list of services to display in the browser (so the user can select one). We send an array of type Object (even though it has Strings inside) by making an array of just the KEYS that are in the HashMap. We won't send an actual Service object unless the client asks for it by calling getService().

Client calls this method after the user selects a service from the displayed list of services (that it got from the method above). This code uses the key (the same key originally sent to the client) to get the corresponding service out of the HashMap.

ServiceBrowser code

## class ServiceBrowser (the client)

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


public class ServiceBrowser {

   JPanel mainPanel;
   JComboBox serviceList;
   ServiceServer server;


   public void buildGUI() {
      JFrame frame = new JFrame("RMI Browser");
      mainPanel = new JPanel();
      frame.getContentPane().add(BorderLayout.CENTER, mainPanel);

      Object[] services = getServicesList();

      serviceList = new JComboBox(services);

      frame.getContentPane().add(BorderLayout.NORTH, serviceList);

      serviceList.addActionListener(new MyListListener());

      frame.setSize(500,500);
      frame.setVisible(true);

  }

   void loadService(Object serviceSelection) {
       try {
          Service svc = server.getService(serviceSelection);

          mainPanel.removeAll();
          mainPanel.add(svc.getGuiPanel());
          mainPanel.validate();
          mainPanel.repaint();
        } catch(Exception ex) {
          ex.printStackTrace();
        }
   }
```

this method does the RMI registry lookup, gets the stub, and calls getServiceList(). (The actual method is on the next page).

Add the services (an array of Objects) to the JComboBox (the list). The JComboBox knows how to make displayable Strings out of each thing in the array.

Here's where we add the actual service to the GUI, after the user has selected one. (This method is called by the event listener on the JComboBox). We call getService() on the remote server (the stub for ServiceServer) and pass it the String that was displayed in the list (which is the SAME String we originally got from the server when we called getServiceList()). The server returns the actual service (serialized), which is automatically deserialized (thanks to RMI) and we simply call the getGuiPanel() on the service and add the result (a JPanel) to the browser's mainPanel.

```
    Object[] getServicesList() {
       Object obj = null;
       Object[] services = null;

       try {

          obj = Naming.lookup("rmi://127.0.0.1/ServiceServer");

       }
       catch(Exception ex) {
        ex.printStackTrace();
       }
       server = (ServiceServer) obj;


       try {

         services = server.getServiceList();

       } catch(Exception ex) {
          ex.printStackTrace();
       }
       return services;

    }

    class MyListListener implements ActionListener {
       public void actionPerformed(ActionEvent ev) {

           Object selection =  serviceList.getSelectedItem();
           loadService(selection);
       }
     }

    public static void main(String[] args) {
       new ServiceBrowser().buildGUI();
    }
}
```

Do the RMI lookup, and get the stub

Cast the stub to the remote interface type, so that we can call getServiceList() on it

getServiceList() gives us the array of Objects, that we display in the JComboBox for the user to select from.

If we're here, it means the user made a selection from the JComboBox list. So, take the selection they made and load the appropriate service. (see the loadService method on the previous page, that asks the server for the service that corresponds with this selection)

DiceService code

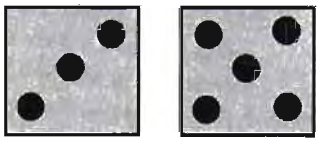
## class DiceService (a universal service, implements Service)


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

public class DiceService implements Service {

    JLabel label;
    JComboBox numOfDice;


    public JPanel getGuiPanel() {
       JPanel panel = new JPanel();
       JButton button = new JButton("Roll 'em!");
       String[] choices = {"1", "2", "3", "4", "5"};
       numOfDice = new JComboBox(choices);
       label = new JLabel("dice values here");
       button.addActionListener(new RollEmListener());
       panel.add(numOfDice);
       panel.add(button);
       panel.add(label);
       return panel;
    }

  public class RollEmListener implements ActionListener {
     public void actionPerformed(ActionEvent ev) {
        // roll the dice
        String diceOutput = "";
        String selection = (String)  numOfDice.getSelectedItem();
        int numOfDiceToRoll = Integer.parseInt(selection);
        for (int i = 0; i < numOfDiceToRoll; i++) {
           int r = (int) ((Math.random() * 6) + 1);
           diceOutput += (" " + r);
        }
       label.setText(diceOutput);

     }
  }
```

Here's the one important method! The method of the Service interface-- the one the client's gonna call when this service is selected and loaded. You can do whatever you want in the getGuiPanel() method, as long as you return a JPanel, so it builds the actual dice-rolling GUI.

### Sharpen your pencil

Think about ways to improve the DiceService. One suggestion: using what you learned in the GUI chapters, make the dice graphical. Use a rectangle, and draw the appropriate number of circles on each one, corresponding to the roll for that particular die.



## class MiniMusicService (a universal service, implements Service)

```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiniMusicService implements Service {

    MyDrawPanel myPanel;

    public JPanel getGuiPanel() {
       JPanel mainPanel = new JPanel();
       myPanel = new MyDrawPanel();
       JButton playItButton = new JButton("Play it");
       playItButton.addActionListener(new PlayItListener());
       mainPanel.add(myPanel);
       mainPanel.add(playItButton);
       return mainPanel;
    }

    public class PlayItListener implements ActionListener {
      public void actionPerformed(ActionEvent ev) {

        try {

         Sequencer sequencer = MidiSystem.getSequencer();
         sequencer.open();

         sequencer.addControllerEventListener(myPanel, new int[] {127});
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

         for (int i = 0; i < 100; i+= 4) {

            int  rNum = (int) ((Math.random() * 50) + 1);
            if (rNum < 38) {  // so now only do it if num <38 (75% of the time)
              track.add(makeEvent(144,1,rNum,100,i));
              track.add(makeEvent(176,1,127,0,i));
              track.add(makeEvent(128,1,rNum,100,i + 2));
            }
         } // end loop

         sequencer.setSequence(seq);
         sequencer.start();
         sequencer.setTempoInBPM(220);
      } catch (Exception ex) {ex.printStackTrace();}

     } // close actionperformed
    } // close inner class
```

The service method! All it does is display a button and the drawing service (where the rectangles will eventually be painted).

This is all the music stuff from the Code Kitchen in chapter 12, so we won't annotate it again here.

## class MiniMusicService, continued...

```
public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
     MidiEvent event = null;
        try {
           ShortMessage a = new ShortMessage();
           a.setMessage(comd, chan, one, two);
           event = new MidiEvent(a, tick);

         }catch(Exception e) { }
     return event;
     }


class MyDrawPanel extends JPanel implements ControllerEventListener {

     // only if we got an event do we want to paint
     boolean msg = false;

     public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
     }

     public Dimension getPreferredSize() {
       return new Dimension(300,300);
     }

     public void paintComponent(Graphics g) {
        if (msg) {

          Graphics2D g2 = (Graphics2D) g;

          int r = (int) (Math.random() * 250);
          int gr = (int) (Math.random() * 250);
          int b = (int) (Math.random() * 250);

          g.setColor(new Color(r,gr,b));

          int ht = (int) ((Math.random() * 120) + 10);
          int width = (int) ((Math.random() * 120) + 10);

          int x = (int) ((Math.random() * 40) + 10);
          int y =  (int) ((Math.random() * 40) + 10);

          g.fillRect(x,y,ht, width);
          msg = false;

       } // close if
     } // close method
   }  // close inner class
} // close class
```

Nothing new on this entire page. You've seen it all in the graphics CodeKitchen. If you want another exercise, try annotating this code yourself, then compare it with the CodeKitchen in the "A very graphic story" chapter.

## class DayOfTheWeekService (a universal service, Implements Service)


```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class DayOfTheWeekService implements Service {

    JLabel outputLabel;
    JComboBox month;
    JTextField day;
    JTextField year;

    public JPanel getGuiPanel() {
       JPanel panel = new JPanel();
       JButton button = new JButton("Do it!");
       button.addActionListener(new DoItListener());
       outputLabel = new JLabel("date appears here");
       DateFormatSymbols dateStuff = new DateFormatSymbols();
       month = new JComboBox(dateStuff.getMonths());
       day = new JTextField(8);
       year = new JTextField(8);
       JPanel inputPanel = new JPanel(new GridLayout(3,2));
       inputPanel.add(new JLabel("Month"));
       inputPanel.add(month);
       inputPanel.add(new JLabel("Day"));
       inputPanel.add(day);
       inputPanel.add(new JLabel("Year"));
       inputPanel.add(year);
       panel.add(inputPanel);
       panel.add(button);
       panel.add(outputLabel);
       return panel;
    }

    public class DoItListener implements ActionListener {
       public void actionPerformed(ActionEvent ev) {
            int monthNum = month.getSelectedIndex();
            int dayNum = Integer.parseInt(day.getText());
            int yearNum = Integer.parseInt(year.getText());
            Calendar c = Calendar.getInstance();
            c.set(Calendar.MONTH, monthNum);
            c.set(Calendar.DAY_OF_MONTH, dayNum);
            c.set(Calendar.YEAR, yearNum);
            Date date = c.getTime();
            String dayOfWeek = (new SimpleDateFormat("EEEE")).format(date);
            outputLabel.setText(dayOfWeek);
       }
    }
}
```

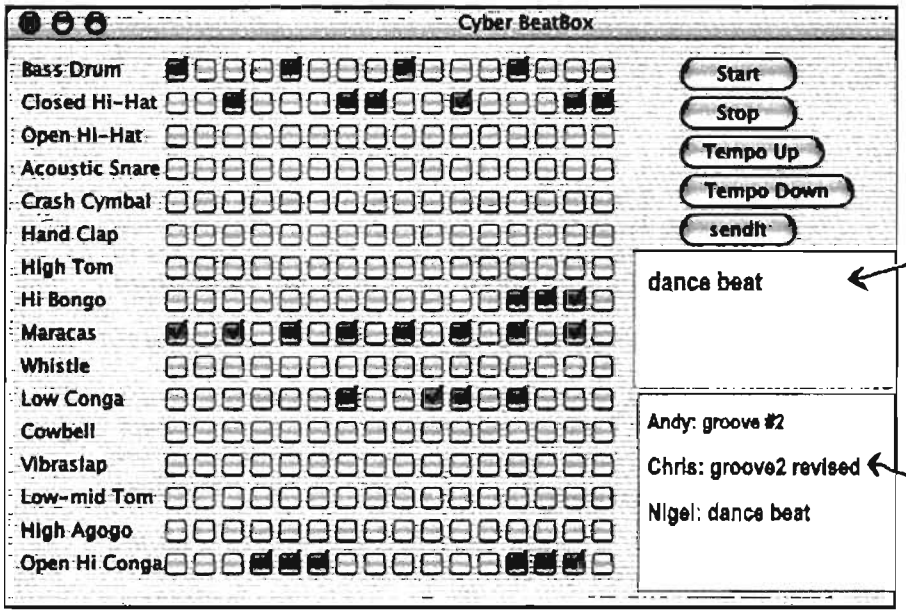
The Service interface method that builds the GUI

Refer to chapter 10 if you need a reminder of how number and date formatting works. This code is slightly different, however, because it uses the Calendar class. Also, the SimpleDateFormat lets us specify a pattern for how the date should print out


# Congratulations!
## You made it to the end.

**Of course, there's still the two appendices.**
**And the index.**
**And then there's the web site...**
**There's no escape, really.**

# Appendix A:
# Final Code Kitchen




Your message gets sent to the other players, along with your current beat pattern, when you hit "sendIt".

Incoming messages from players. Click one to load the pattern that goes with it, and then click 'Start' to play it

**Finally, the complete version of the BeatBox!**

**It connects to a simple MusicServer so that you can send and receive beat patterns with other clients.**

## Final BeatBox client program

Most of this code is the same as the code from the CodeKitchens in the previous chapters, so we don't annotate the whole thing again. The new parts include:

GUI - two new components are added for the text area that displays incoming messages (actually a scrolling list) and the text field.

NETWORKING - just like the SimpleChatClient in this chapter, the BeatBox now connects to the server and gets an input and output stream.

THREADS - again, just like the SimpleChatClient, we start a 'reader' class that keeps looking for incoming messages from the server. But instead of just text, the messages coming in include TWO objects: the String message and the serialized ArrayList (the thing that holds the state of all the checkboxes.)

```
import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.event.*;

public class BeatBoxFinal {

      JFrame theFrame;
      JPanel mainPanel;
      JList incomingList;
      JTextField userMessage;
      ArrayList<JCheckBox> checkboxList;
      int nextNum;
      Vector<String> listVector = new Vector<String>();
      String userName;
      ObjectOutputStream out;
      ObjectInputStream in;
      HashMap<String, boolean[]> otherSeqsMap = new HashMap<String, boolean[]>();

      Sequencer sequencer;
      Sequence sequence;
      Sequence mySequence = null;
      Track track;


      String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat", "Open Hi-Hat","Acoustic
      Snare", "Crash Cymbal", "Hand Clap", "High Tom", "Hi Bongo", "Maracas", "Whistle",
      "Low Conga", "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo", "Open Hi Conga"};

      int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};
```

```
public static void main (String[] args) {
   new BeatBoxFinal().startUp(args[0]);  // args[0] is your user ID/screen name
}

public void startUp(String name) {
    userName = name;
    // open connection to the server
    try {
      Socket sock = new Socket("127.0.0.1", 4242);
      out = new ObjectOutputStream(sock.getOutputStream());
      in = new ObjectInputStream(sock.getInputStream());
      Thread remote = new Thread(new RemoteReader());
      remote.start();
    } catch(Exception ex) {
        System.out.println("couldn't connect - you'll have to play alone.");
    }
    setUpMidi();
    buildGUI();
} // close startUp

 public void buildGUI() {

    theFrame = new JFrame("Cyber BeatBox");
    BorderLayout layout = new BorderLayout();
    JPanel background = new JPanel(layout);
    background.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

    checkboxList = new ArrayList<JCheckBox>();

    Box buttonBox = new Box(BoxLayout.Y_AXIS);
    JButton start = new JButton("Start");
    start.addActionListener(new MyStartListener());
    buttonBox.add(start);

    JButton stop = new JButton("Stop");
    stop.addActionListener(new MyStopListener());
    buttonBox.add(stop);

    JButton upTempo = new JButton("Tempo Up");
    upTempo.addActionListener(new MyUpTempoListener());
    buttonBox.add(upTempo);

    JButton downTempo = new JButton("Tempo Down");
    downTempo.addActionListener(new MyDownTempoListener());
    buttonBox.add(downTempo);

    JButton sendIt = new JButton("sendIt");
    sendIt.addActionListener(new MySendListener());
    buttonBox.add(sendIt);

    userMessage = new JTextField();
```

Add a command-line argument for your screen name.

Example: % java BeatBoxFinal theFlash

Nothing new... set up the networking, I/O, and make (and start) the reader thread.

GUI code, nothing new here

## final BeatBox code

```
        buttonBox.add(userMessage);

        incomingList = new JList();
        incomingList.addListSelectionListener(new MyListSelectionListener());
        incomingList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        JScrollPane theList = new JScrollPane(incomingList);
        buttonBox.add(theList);
        incomingList.setListData(listVector); // no data to start with

        Box nameBox = new Box(BoxLayout.Y_AXIS);
        for (int i = 0; i < 16; i++) {
            nameBox.add(new Label(instrumentNames[i]));
        }

        background.add(BorderLayout.EAST, buttonBox);
        background.add(BorderLayout.WEST, nameBox);

        theFrame.getContentPane().add(background);
        GridLayout grid = new GridLayout(16,16);
        grid.setVgap(1);
        grid.setHgap(2);
        mainPanel = new JPanel(grid);
        background.add(BorderLayout.CENTER, mainPanel);

        for (int i = 0; i < 256; i++) {
            JCheckBox c = new JCheckBox();
            c.setSelected(false);
            checkboxList.add(c);
            mainPanel.add(c);
        } // end loop

        theFrame.setBounds(50,50,300,300);
        theFrame.pack();
        theFrame.setVisible(true);
    } // close buildGUI

    public void setUpMidi() {
        try {
        sequencer = MidiSystem.getSequencer();
        sequencer.open();
        sequence = new Sequence(Sequence.PPQ,4);
        track = sequence.createTrack();
        sequencer.setTempoInBPM(120);
    } catch(Exception e) {e.printStackTrace();}

} // close setUpMidi
```

JList is a component we haven't used before. This is where the incoming messages are displayed. Only instead of a normal chat where you just LOOK at the messages, in this app you can SELECT a message from the list to load and play the attached beat pattern.

Nothing else on this page is new

Get the Sequencer, make a Sequence, and make a Track

```
public void buildTrackAndStart() {
   ArrayList<Integer> trackList = null; // this will hold the instruments for each
   sequence.deleteTrack(track);
   track = sequence.createTrack();

   for (int i = 0; i < 16; i++) {

     trackList = new ArrayList<Integer>();

     for (int j = 0; j < 16; j++) {
        JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
        if (jc.isSelected()) {
           int key = instruments[i];
           trackList.add(new Integer(key));
        } else {
          trackList.add(null);  // because this slot should be empty in the track
        }
     } // close inner loop
     makeTracks(trackList);
   } // close outer loop
   track.add(makeEvent(192,9,1,0,15)); // - so we always go to full 16 beats
   try {
      sequencer.setSequence(sequence);
      sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
      sequencer.start();
      sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}
 } // close method

public class MyStartListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      buildTrackAndStart();
   } // close actionPerformed
} // close inner class

public class MyStopListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      sequencer.stop();
  } // close actionPerformed
} // close inner class

public class MyUpTempoListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequencer.getTempoFactor();
      sequencer.setTempoFactor((float)(tempoFactor * 1.03));
  } // close actionPerformed
} // close inner class
```

Build a track by walking through the checkboxes to get their state, and mapping that to an instrument (and making the MidiEvent for it). This is pretty complex, but it is EXACTLY as it was in the previous chapters, so refer to previous CodeKitchens to get the full explanation again.

The GUI listeners. Exactly the same as the previous chapter's version.

```
 public class MyDownTempoListener implements ActionListener {
     public void actionPerformed(ActionEvent a) {
       float tempoFactor = sequencer.getTempoFactor();
       sequencer.setTempoFactor((float)(tempoFactor * .97));
     }
}

public class MySendListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
       // make an arraylist of just the STATE of the checkboxes
        boolean[] checkboxState = new boolean[256];
       for (int i = 0; i < 256; i++) {
          JCheckBox check = (JCheckBox) checkboxList.get(i);
          if (check.isSelected()) {
            checkboxState[i] = true;
          }
       } // close loop
     String messageToSend = null;
     try {
        out.writeObject(userName + nextNum++ + ": " + userMessage.getText());
        out.writeObject(checkboxState);
     } catch(Exception ex) {
         System.out.println("Sorry dude. Could not send it to the server.");
     }
     userMessage.setText("");
   } // close actionPerformed
} // close inner class

public class MyListSelectionListener implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent le) {
      if (!le.getValueIsAdjusting()) {
        String selected = (String) incomingList.getSelectedValue();
        if (selected != null) {
            // now go to the map, and change the sequence
            boolean[] selectedState = (boolean[]) otherSeqsMap.get(selected);
            changeSequence(selectedState);
            sequencer.stop();
            buildTrackAndStart();
         }
      }
   } // close valueChanged
} // close inner class
```

This is new... it's a lot like the SimpleChatClient, except instead of sending a String message, we serialize two objects (the String message and the beat pattern) and write those two objects to the socket output stream (to the server).

This is also new -- a ListSelectionListener that tells us when the user made a selection on the list of messages. When the user selects a message, we IMMEDIATELY load the associated beat pattern (it's in the HashMap called otherSeqsMap) and start playing it. There's some if tests because of little quirky things about getting ListSelectionEvents.

```
public class RemoteReader implements Runnable {
    boolean[] checkboxState = null;
    String nameToShow = null;
    Object obj = null;
    public void run() {
      try {
        while((obj=in.readObject()) != null) {
          System.out.println("got an object from server");
          System.out.println(obj.getClass());
          String nameToShow = (String) obj;
          checkboxState = (boolean[]) in.readObject();
          otherSeqsMap.put(nameToShow, checkboxState);
          listVector.add(nameToShow);
          incomingList.setListData(listVector);
        } // close while
      } catch(Exception ex) {ex.printStackTrace();}
    } // close run
} // close inner class
```

This is the thread job -- read in data from the server. In this code, 'data' will always be two serialized objects: the String message and the beat pattern (an ArrayList of checkbox state values)

When a message comes in, we read (deserialize) the two objects (the message and the ArrayList of Boolean checkbox state values) and add it to the JList component. Adding to a JList is a two-step thing: you keep a Vector of the lists data (Vector is an old-fashioned ArrayList), and then tell the JList to use that Vector as it's source for what to display in the list.

```
public class MyPlayMineListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      if (mySequence != null) {
         sequence = mySequence;  // restore to my original
      }
   } // close actionPerformed
} // close inner class
```

This method is called when the user selects something from the list. We IMMEDIATELY change the pattern to the one they selected.

```
public void changeSequence(boolean[] checkboxState) {
   for (int i = 0; i < 256; i++) {
      JCheckBox check = (JCheckBox) checkboxList.get(i);
       if (checkboxState[i]) {
           check.setSelected(true);
       } else {
            check.setSelected(false);
       }
   } // close loop
} // close changeSequence
```

All the MIDI stuff is exactly the same as it was in the previous version.

```
public void makeTracks(ArrayList list) {
   Iterator it = list.iterator();
   for (int i = 0; i < 16; i++) {
      Integer num = (Integer) it.next();
      if (num != null) {
         int numKey = num.intValue();
         track.add(makeEvent(144,9,numKey, 100, i));
         track.add(makeEvent(128,9,numKey,100, i + 1));
       }
    } // close loop
 } // close makeTracks()
```

## final BeatBox code

```
    public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
        MidiEvent event = null;
        try {
          ShortMessage a = new ShortMessage();
          a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);
        }catch(Exception e) { }
        return event;
    } // close makeEvent

} // close class
```

Nothing new. Just like the last version.


What are some of the ways you can improve this program?

Here are a few ideas to get you started:

1) Once you select a pattern, whatever current pattern was playing is blown away. If that was a new pattern you were working on (or a modification of another one), you're out of luck. You might want to pop up a dialog box that asks the user if he'd like to save the current pattern.

2) If you fail to type in a command-line argument, you just get an exception when you run it! Put something in the main method that checks to see if you've passed in a command-line argument. If the user doesn't supply one, either pick a default or print out a message that says they need to run it again, but this time with an argument for their screen name.

3) It might be nice to have a feature where you can click a button and it will generate a random pattern for you. You might hit on one you really like. Better yet, have another feature that lets you load in existing 'foundation' patterns, like one for jazz, rock, reggae, etc. that the user can add to.

You can find existing patterns on the Head First Java web start.

## Final BeatBox server program

Most of this code is identical to the SimpleChatServer we made in the Networking and Threads chapter. The only difference, in fact, is that this server receives, and then re-sends, two serialized objects instead of a plain String (although one of the serialized objects happens to *be* a String).

```
import java.io.*;
import java.net.*;
import java.util.*;

public class MusicServer {

    ArrayList<ObjectOutputStream> clientOutputStreams;

    public static void main (String[] args) {
         new MusicServer().go();
    }

    public class ClientHandler implements Runnable {

         ObjectInputStream in;
         Socket clientSocket;

         public ClientHandler(Socket socket) {
           try {
             clientSocket = socket;
              in = new ObjectInputStream(clientSocket.getInputStream());

           } catch(Exception ex) {ex.printStackTrace();}
          } // close constructor


        public void run() {
             Object o2 = null;
             Object o1 = null;
           try {

             while ((o1 = in.readObject()) != null) {

                o2 = in.readObject();

                System.out.println("read two objects");
                tellEveryone(o1, o2);
              } // close while

           } catch(Exception ex) {ex.printStackTrace();}
       } // close run
   } // close inner class
```

```
public void go() {
  clientOutputStreams = new ArrayList<ObjectOutputStream>();

     try {
       ServerSocket serverSock = new ServerSocket(4242);

       while(true) {
          Socket clientSocket = serverSock.accept();
          ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream())
          clientOutputStreams.add(out);

          Thread t = new Thread(new ClientHandler(clientSocket));
          t.start();

          System.out.println("got a connection");
       }
    }catch(Exception ex) {
      ex.printStackTrace();
    }
} // close go

public void tellEveryone(Object one, Object two) {
   Iterator it = clientOutputStreams.iterator();
   while(it.hasNext()) {
     try {
        ObjectOutputStream out = (ObjectOutputStream) it.next();
        out.writeObject(one);
        out.writeObject(two);
      }catch(Exception ex) {ex.printStackTrace();}
    }
} // close tellEveryone

} // close class
```

# Appendix B

## The Top Ten Topics that almost made it into the Real Book...


We covered a lot of ground, and you're almost finished with this book. We'll miss you, but before we let you go, we wouldn't feel right about sending you out into JavaLand without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small appendix. Actually, we *did* originally include everything you need to know about Java (not already covered by the other chapters), by reducing the type point size to .00003. It all fit, but nobody could read it. So, we threw most of it away, but kept the best bits for this Top Ten appendix.

This really *is* the end of the book. Except for the index (a must-read!).

# #10 Bit Manipulation

## Why do you care?

We've talked about the fact that there are 8 bits in a byte, 16 bits in a short, and so on. You might have occasion to turn individual bits on or off. For instance you might find yourself writing code for your new Java enabled toaster, and realize that due to severe memory limitations, certain toaster settings are controlled at the bit level. For easier reading, we're showing only the last 8 bits in the comments rather than the full 32 for an int).

### Bitwise NOT Operator: ~

This operator 'flips all the bits' of a primitive.

```
int x = 10;   //     bits are 00001010
x = ~x;       // bits are now 11110101
```

The next three operators compare two primitives on a bit by bit basis, and return a result based on comparing these bits. We'll use the following example for the next three operators:

```
int x = 10;    // bits are 00001010

int y =  6;    // bits are 00000110
```

### Bitwise AND Operator: &

This operator returns a value whose bits are turned on only if ***both*** original bits are turned on:

```
int a = x & y;  // bits are 00000010
```

### Bitwise OR Operator: |

This operator returns a value whose bits are turned on only if ***either*** of the original bits are turned on:

```
int a = x | y;  // bits are 00001110
```

### Bitwise XOR (exclusive OR) Operator: ^

This operator returns a value whose bits are turned on only if ***exactly one*** of the original bits are turned on:

```
int a = x ^ y;  // bits are 00001100
```

### The Shift Operators

These operators take a single integer primitive and shift (or slide) all of its bits in one direction or another. If you want to dust off your binary math skills, you might realize that shifting bits *left* effectively *multiplies* a number by a power of two, and shifting bits *right* effectively *divides* a number by a power of two.

We'll use the following example for the next three operators:

```
int x = -11;   //     bits are 11110101
```

Ok, ok, we've been putting it off, here is the world's shortest explanation of storing negative numbers, and *two's complement*. Remember, the leftmost bit of an integer number is called the ***sign bit***. A negative integer number in Java *always* has its sign bit turned *on* (i.e. set to 1). A positive integer number always has its sign bit turned *off* (0). Java uses the *two's complement* formula to store negative numbers. To change a number's sign using two's complement, flip all the bits, then add 1 (with a byte, for example, that would mean adding 00000001 to the flipped value).

### Right Shift Operator: >>

This operator shifts all of a number's bits right by a certain number, and fills all of the bits on the left side with whatever the original leftmost bit was. **The sign bit does *not* change:**

```
int y = x >> 2;  // bits are 11111101
```

### Unsigned Right Shift Operator: >>>

Just like the right shift operator BUT it ALWAYS fills the leftmost bits with zeros. **The sign bit *might* change**:

```
int y = x >>> 2;  // bits are 00111101
```

### Left Shift Operator: <<

Just like the unsigned right shift operator, but in the other direction; the rightmost bits are filled with zeros. **The sign bit *might* change.**

```
int y = x << 2;  // bits are 11010100
```

# #9 Immutability

## Why do you care that Strings are Immutable?

When your Java programs start to get big, you'll inevitably end up with lots and lots of String objects. For security purposes, and for the sake of conserving memory (remember your Java programs can run on teeny Java-enabled cell phones), Strings in Java are immutable. What this means is that when you say:

```
String s = "0";

for (int x = 1; x < 10; x++) {
  s = s + x;
}
```

What's actually happening is that you're creating ten String objects (with values "0", "01", "012", through "0123456789"). In the end *s* is referring to the String with the value "0123456789", but at this point there are *ten* Strings in existence!

Whenever you make a new String, the JVM puts it into a special part of memory called the 'String Pool' (sounds refreshing doesn't it?). If there is already a String in the String Pool with the same value, the JVM doesn't create a duplicate, it simply refers your reference variable to the existing entry. The JVM can get away with this because Strings are immutable; one reference variable can't change a String's value out from under another reference variable referring to the same String.

The other issue with the String pool is that the Garbage Collector *doesn't go there*. So in our example, unless by coincidence you later happen to make a String called "01234", for instance, the first nine Strings created in our *for* loop will just sit around wasting memory.

### How does this save memory?

Well, if you're not careful, *it doesn't!* But if you understand how String immutability works, than you can sometimes take advantage of it to save memory. If you have to do a lot of String manipulations (like concatenations, etc.), however, there is another class StringBuilder, better suited for that purpose. We'll talk more about StringBuilder in a few pages.

## Why do you care that Wrappers are Immutable?

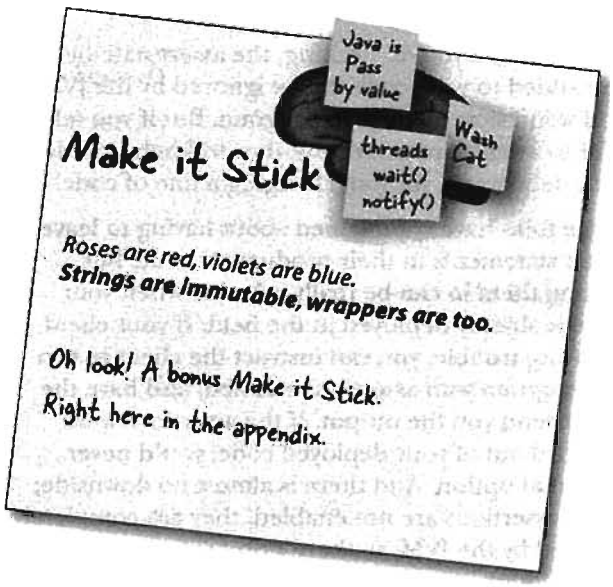
In the Math chapter we talked about the two main uses of the wrapper classes:

- Wrapping a primitive so it can pretend to be an object.
- Using the static utility methods (for example, Integer.parseInt()).

It's important to remember that when you create a wrapper object like:

```
Integer iWrap = new Integer(42);
```

That's it for that wrapper object. Its value will *always* be 42. ***There is no setter method for a wrapper object.*** You can, of course, refer *iWrap* to a *different* wrapper object, but then you'll have *two* objects. Once you create a wrapper object, there's no way to change the *value* of that object!



# #8 Assertions

We haven't talked much about how to debug your Java program while you're developing it. We believe that you should learn Java at the command line, as we've been doing throughout the book. Once you're a Java pro, if you decide to use an IDE*, you might have other debugging tools to use. In the old days, when a Java programmer wanted to debug her code, she'd stick a bunch of System.out.println( ) statements throughout the program, printing current variable values, and "I got here" messages, to see if the flow control was working properly. (The ready-bake code in chapter 6 left some debugging 'print' statements in the code.) Then, once the program was working correctly, she'd go through and take all those System.out.println( ) statements back out again. It was tedious and error prone. But as of Java 1.4 (and 5.0), debugging got a whole lot easier. The answer?

## Assertions

Assertions are like System.out.println( ) statements on steroids. Add them to your code as you would add println statements. The Java 5.0 compiler assumes you'll be compiling source files that are 5.0 compatible, so as of Java 5.0, compiling with assertions is enabled by default.

At runtime, if you do nothing, the assert statements you added to your code will be ignored by the JVM, and won't slow down your program. But if you tell the JVM to *enable* your assertions, they will help you do your debugging, without changing a line of code!

Some folks have complained about having to leave assert statements in their production code, but leaving them in can be really valuable when your code is already deployed in the field. If your client is having trouble, you can instruct the client to run the program with assertions enabled, and have the client send you the output. If the assertions were stripped out of your deployed code, you'd never have that option. And there is almost no downside; when assertions are not enabled, they are completely ignored by the JVM, so there's no performance hit to worry about.

## How to make Assertions work

Add assertion statements to your code wherever you believe that something *must be true*. For instance:

```
assert (height > 0);

// if true, program continues normally
// if false, throw an AssertionError
```

You can add a little more information to the stack trace by saying:

```
assert (height > 0) : "height = " +
height + " weight = " + weight;
```

The expression after the colon can be any legal Java expression ***that resolves to a non-null value.*** But whatever you do, ***don't create assertions that change an object's state!*** If you do, enabling assertions at runtime might change how your program performs.

## Compiling and running with Assertions

To *compile* with assertions:

```
javac TestDriveGame.java
```

(Notice that no command line options were necessary.)

To *run* with assertions:

```
java -ea TestDriveGame
```

* IDE stands for Integrated Development Environment and includes tools such as Eclipse, Borland's JBuilder, or the open source NetBeans (netbeans.org).

## #7 Block Scope

In chapter 9, we talked about how local variables live only as long as the method in which they're declared stays on the stack. But some variables can have even *shorter* lifespans. Inside of methods, we often create *blocks* of code. We've been doing this all along, but we haven't explicitly *talked* in terms of *blocks*. Typically, blocks of code occur within methods, and are bounded by curly braces { }. Some common examples of code blocks that you'll recognize include loops (*for*, *while*) and conditional expressions (like *if* statements).

Let's look at an example:

```
void doStuff() {          // start of the method block
    int x = 0;            // local variable scoped to the entire method
    for(int y = 0; y < 5; y++) {   // beginning of a for loop block, and y is scoped to only the for loop!
        x = x + y;        // No problem, x and y are both in scope
    }                     // end of the for loop block
    x = x * y;            // Aack! Won't compile! y is out of scope here! (this is not the way it works in some other languages, so beware!)
}                         // end of the method block, now x is also out of scope
```

In the previous example, *y* was a block variable, declared inside a block, and *y* went out of scope as soon as the for loop ended. Your Java programs will be more debuggable and expandable if you use local variables instead of instance variables, and block variables instead of local variables, whenever possible. The compiler will make sure that you don't try to use a variable that's gone out of scope, so you don't have to worry about runtime meltdowns.

## #6 Linked Invocations

While you did see a little of this in this book, we tried to keep our syntax as clean and readable as possible. There are, however, many legal shortcuts in Java, that you'll no doubt be exposed to, especially if you have to read a lot code you didn't write. One of the more common constructs you will encounter is known as *linked invocations*. For example:

```
StringBuffer sb = new StringBuffer("spring");
sb = sb.delete(3,6).insert(2,"umme").deleteCharAt(1);
System.out.println("sb = " + sb);
// result is sb = summer
```

What in the world is happening in the second line of code? Admittedly, this is a contrived example, but you need to learn how to decipher these.

1 - Work from left to right.

2 - Find the result of the leftmost method call, in this case `sb.delete(3,6)`. If you look up StringBuffer in the API docs, you'll see that the `delete()` method returns a StringBuffer object. The result of running the `delete()` method is a StringBuffer object with the value "spr".

3 - The next leftmost method (`insert()`)is called on the newly created StringBuffer object "spr". The result of that method call (the `insert()` method), is *also* a StringBuffer object (although it doesn't have to be the same type as the previous method return), and so it goes, the returned object is used to call the next method to the right. In theory, you can link as many methods as you want in a single statement (although it's rare to see more than three linked methods in a single statement). Without linking, the second line of code from above would be more readable, and look something like this:

```
sb = sb.delete(3,6);
sb = sb.insert(2,"umme");
sb = sb.deleteCharAt(1);
```

But here's a more common, and useful example, that you saw us using, but we thought we'd point it out again here. This is for when your main() method needs to invoke an instance method of the main class, but you don't need to keep a *reference* to the instance of the class. In other words, the main() needs to create the instance *only* so that main() can invoke one of the instance's *methods*.

```
class Foo {
   public static void main(String [] args) [
     new Foo().go();
   }
   void go() {
    // here's what we REALLY want...
   }
}
```

we want to call go(), but we don't care about the Foo instance, so we don't bother assigning the new Foo object to a reference.

# #5 Anonymous and Static Nested Classes

### Nested classes come in many flavors

In the GUI event-handling section of the book, we started using inner (nested) classes as a solution for implementing listener interfaces. That's the most common, practical, and readable form of an inner class—where the class is simply nested within the curly braces of another enclosing class. And remember, it means you need an instance of the outer class in order to get an instance of the inner class, because the inner class is a *member* of the outer/enclosing class.

But there are other kinds of inner classes including *static* and *anonymous*. We're not going into the details here, but we don't want you to be thrown by strange syntax when you see it in someone's code. Because out of virtually anything you can do with the Java language, perhaps nothing produces more bizarre-looking code than anonymous inner classes. But we'll start with something simpler—static nested classes.

### Static nested classes

You already know what static means—something tied to the class, not a particular instance. A static nested class looks just like the non-static classes we used for event listeners, except they're marked with the keyword `static`.

A static nested class is just that—a class enclosed within another, and marked with the static modifier.

```
public class FooOuter {
  static class BarInner {
      void sayIt() {
         System.out.println("method of a static inner class");
      }
  }
}

class Test {
   public static void main (String[] args) {
      FooOuter.BarInner foo = new FooOuter.BarInner();
      foo.sayIt();
   }
}
```

Because a static nested class is...static, you don't use an instance of the outer class. You just use the name of the class, the same way you invoke static methods or access static variables.

Static nested classes are more like regular non-nested classes in that they don't enjoy a special relationship with an enclosing outer object. But because static nested classes are still considered a *member* of the enclosing/outer class, they still get access to any private members of the outer class... but *only the ones that are also static*. Since the static nested class isn't connected to an instance of the outer class, it doesn't have any special way to access the non-static (instance) variables and methods.

# #5 Anonymous and Static Nested Classes, continued

### The difference between *nested* and *inner*

Any Java class that's defined within the scope of another class is known as a ***nested*** class. It doesn't matter if it's anonymous, static, normal, whatever. If it's inside another class, it's technically considered a *nested* class. But *non-static* nested classes are often referred to as *inner* classes, which is what we called them earlier in the book. The bottom line: all inner classes are nested classes, but not all nested classes are inner classes.

### Anonymous inner classes

Imagine you're writing some GUI code, and suddenly realize that you need an instance of a class that implements ActionListener. But you realize you don't *have* an instance of an ActionListener. Then you realize that you also never wrote a *class* for that listener. You have two choices at that point:

1) Write an inner class in your code, the way we did in our GUI code, and then instantiate it and pass that instance into the button's event registration (addActionListener()) method.

OR

2) Create an *anonymous* inner class and instantiate it, right there, just-in-time. ***Literally right where you are at the point you need the listener object.*** That's right, you create the class and the instance in the place where you'd normally be supplying just the instance. Think about that for a moment—it means you pass the entire *class* where you'd normally pass only an *instance* into a method argument!

```
import java.awt.event.*;
import javax.swing.*;
public class TestAnon {
   public static void main (String[] args) {

      JFrame frame = new JFrame();
      JButton button = new JButton("click");
      frame.getContentPane().add(button);
      // button.addActionListener(quitListener);

      button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          System.exit(0);
        }
      });

   }
}
```

We made a frame and added a button, and now we need to register an action listener with the button. Except we never made a class that implements the ActionListener interface...

Normally we'd do something like this—passing in a reference to an instance of an inner class... an inner class that implements ActionListener (and the actionPerformed() method).

But now instead of passing in an object reference, we pass in... the whole new class definition!! In other words, we write the class that implements ActionListener RIGHT HERE WHERE WE NEED IT. The syntax also creates an instance of the class automatically.

This statement

ends down here!

Notice that we say "new ActionListener()" even though ActionListener is an interface and so you can't MAKE an instance of it! But this syntax really means, "create a new class (with no name) that implements the ActionListener interface, and by the way, here's the implementation of the interface methods actionPerformed().

# #4 Access Levels and Access Modifiers (Who Sees What)

Java has *four* access *levels* and *three* access *modifiers*. There are only *three* modifiers because the *default* (what you get when you don't use any access modifier) *is* one of the four access levels.

**Access Levels** (in order of how restrictive they are, from least to most restrictive)

*public* ← public means any code anywhere can access the public thing (by 'thing' we mean class, variable, method, constructor, etc.).

*protected* ← protected works just like default (code in the same package has access), EXCEPT it also allows subclasses outside the package to inherit the protected thing.

*default* ← default access means that only code within the same package as the class with the default thing can access the default thing.

*private* ← private means that only code within the same class can access the private thing. Keep in mind it means private to the class, not private to the object. One Dog can see another Dog object's private stuff, but a Cat can't see a Dog's privates.

**Access modifiers**

```
public
protected
private
```

Most of the time you'll use only public and private access levels.

### public

Use public for classes, constants (static final variables), and methods that you're exposing to other code (for example getters and setters) and most constructors.

### private

Use private for virtually all instance variables, and for methods that you don't want outside code to call (in other words, methods *used* by the public methods of your class).

But although you might not use the other two (protected and default), you still need to know what they do because you'll see them in other code.

# #4 Access Levels and Access Modifiers, cont.

## default and protected

### default

Both protected and default access levels are tied to packages. Default access is simple—it means that only code *within the same package* can access code with default access. So a default class, for example (which means a class that isn't explicitly declared as *public*) can be accessed by only classes within the same package as the default class.

But what does it really mean to *access* a class? Code that does not have access to a class is not allowed to even *think* about the class. And by think, we mean *use* the class in code. For example, if you don't have access to a class, because of access restriction, you aren't allowed to instantiate the class or even declare it as a type for a variable, argument, or return value. You simply can't type it into your code at all! If you do, the compiler will complain.

Think about the implications—a default class with public methods means the public methods aren't really public at all. You can't access a method if you can't *see* the class.

Why would anyone want to restrict access to code within the same package? Typically, packages are designed as a group of classes that work together as a related set. So it might make sense that classes within the same package need to access one another's code, while as a package, only a small number of classes and methods are exposed to the outside world (i.e. code outside that package).

OK, that's default. It's simple—if something has default access (which, remember, means no explicit access modifier!), only code within the same package as the default *thing* (class, variable, method, inner class) can access that *thing*.

Then what's *protected* for?

### protected

Protected access is almost identical to default access, with one exception: it allows subclasses to *inherit* the protected thing, *even if those subclasses are outside the package of the superclass they extend.* That's it. That's *all* protected buys you—the ability to let your subclasses be outside your superclass package, yet still *inherit* pieces of the class, including methods and constructors.

Many developers find very little reason to use protected, but it is used in some designs, and some day you might find it to be exactly what you need. One of the interesting things about protected is that—unlike the other access levels—protected access applies only to *inheritance.* If a subclass-outside-the-package has a *reference* to an instance of the superclass (the superclass that has, say, a protected method), the subclass can't access the protected method using that superclass reference! The only way the subclass can access that method is by *inheriting* it. In other words, the subclass-outside-the-package doesn't have *access* to the protected method, it just *has* the method, through inheritance.

## #3 String and StringBuffer/StringBuilder Methods

Two of the most commonly used classes in the Java API are String and StringBuffer (remember from #9 a few pages back, Strings are immutable, so a StringBuffer/StringBuilder can be a lot mor efficient if you're manipulating a String). As of Java 5.0 you should use the String***Builder*** class instead of String***Buffer***, unless your String manipulations need to be thread-safe, which is not common. Here's a brief overview of the **key** methods in these classes:

***Both String and StringBuffer/StringBuilder classes have:***

```
char charAt(int index);                  // what char is at a certain position
int length();                            // how long is this
String substring(int start, int end);    // get a part of this
String toString();                       // what's the String value of this
```

***To concatenate Strings:***

```
String concat(string);                   // for the String class
String append(String);                   // for StringBuffer & StringBuilder
```

***The String class has:***

```
String replace(char old, char new);      // replace all occurences of a char
String substring(int begin, int end);    // get a portion of a String
char [] toCharArray();                   // convert to an array of chars
String toLowerCase();                    // convert all characters to lower case
String toUpperCase();                    // convert all characters to upper case
String trim();                           // remove whitespace from the ends
String valueOf(char [])                  // make a String out of a char array
String valueOf(int i)                    // make a String out of a primitive
                                         // other primitives are supported as well
```

***The StringBuffer & StringBuilder classes have:***

```
StringBxxxx delete(int start, int end);                    // delete a portion
StringBxxxx insert(int offset, any primitive or a char []); // insert something
StringBxxxx replace(int start, int end, String s);         // replace this part with this String
StringBxxxx reverse();                                     // reverse the SB from front to back
void setCharAt(int index, char ch);                        // replace a given character
```

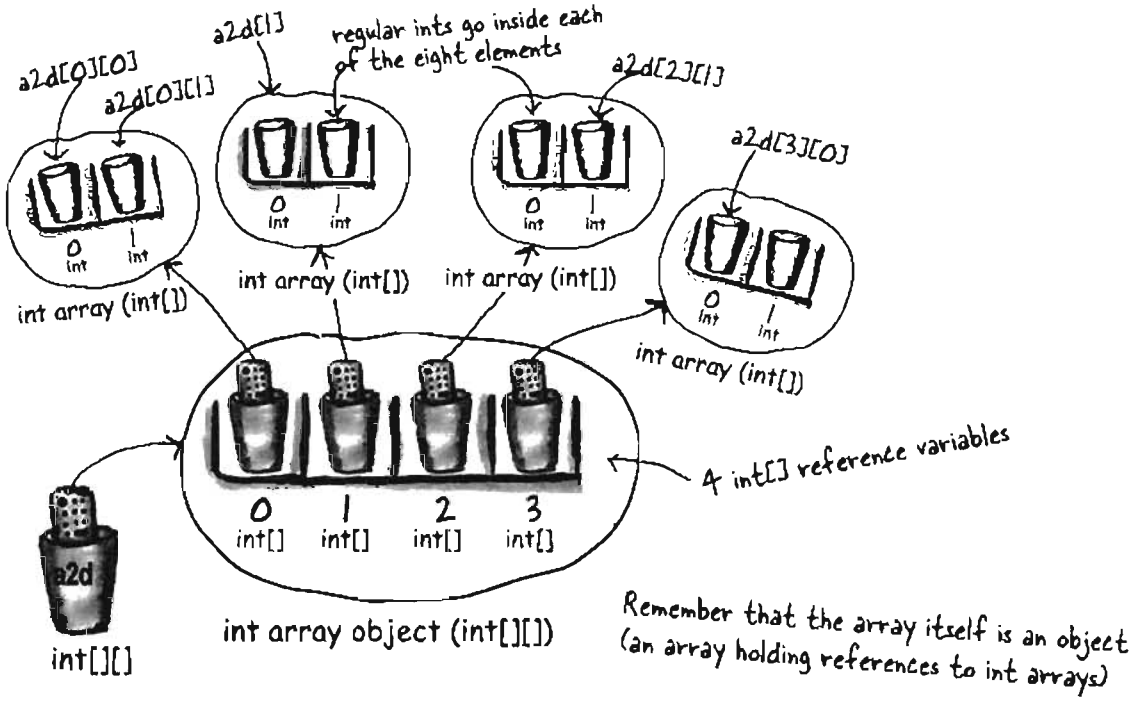
Note: StringB~~xxxx~~ refers to either String*Buffer* or String*Builder*, as appropriate.

## #2 Multidimensional Arrays

In most languages, if you create, say, a 4 x 2 two-dimensional array, you would visualize a rectangle, 4 elements by 2 elements, with a total of 8 elements. But in Java, such an array would actually be *5* arrays linked together! In Java, a two dimensional array is simply *an array of arrays*. (A three dimensional array is an array of arrays of arrays, but we'll leave that for you to play with.) Here's how it works

```
int[][] a2d  = new int [4][2];
```

The JVM creates an array with 4 elements. *Each* of these four elements is actually a reference variable referring to a (newly created), int array with 2 elements.




### Working with multidimensional arrays

- To access the second element in the third array: `int x = a2d[2][1];  // remember, 0 based!`
- To make a one-dimensional reference to one of the sub-arrays: `int[] copy = a2d[1];`
- Short-cut initialization of a 2 x 3 array: `int[][] x = { { 2,3,4 }, { 7,8,9 } };`
- To make a 2d array with irregular dimensions:

```
int[][] y = new int [2][];     //  makes only the first array, with a length of 2
y[0] = new int [3];    // makes the first sub-array 3 elements in length
y[1] = new int [5];    // makes the second sub-array 5 elements in length
```

## And the number one topic that didn't quite make it in...

## #1 Enumerations (also called Enumerated Types or Enums)

We've talked about constants that are defined in the API, for instance, **`JFrame.EXIT_ON_CLOSE`**. You can also create your own constants by marking a variable **`static final`**. But sometimes you'll want to create a set of constant values to represent the ***only*** valid values for a variable. This set of valid values is commonly referred to as an *enumeration*. Before Java 5.0 you could only do a half-baked job of creating an enumeration in Java. As of Java 5.0 you can create full fledged enumerations that will be the envy of all your pre-Java 5.0-using friends.

**Who's in the band?**

Let's say that you're creating a website for your favorite band, and you want to make sure that all of the comments are directed to a particular band member.

**The old way to fake an "enum":**

```
public static final int JERRY = 1;
public static final int BOBBY = 2;
public static final int PHIL = 3;

// later in the code

if (selectedBandMember == JERRY) {
  // do JERRY related stuff
}
```

We're hoping that by the time we got here "selectedBandMember" has a valid value!

The good news about this technique is that it DOES make the code easier to read. The other good news is that you can't ever change the value of the fake enums you've created; `JERRY` will always be `1`. The bad news is that there's no easy or good way to make sure that the value of `selectedBandMember` will always be `1`, `2`, or `3`. If some hard to find piece of code sets `selectedBandMember` equal to `812`, it's pretty likely your code will break...

## #1 Enumerations, cont.

The same situation using a genuine Java 5.0 enum. While this is a very basic enumeration, most enumerations usually *are* this simple.

**A new, official "enum":**

```
public enum Members { JERRY, BOBBY, PHIL };
public Members selectedBandMember;

// later in the code

if (selectedBandMember == Members.JERRY) {
  // do JERRY related stuff
}
```

This kind of looks like a simple class definition doesn't it? It turns out that enums ARE a special kind of class. Here we've created a new enumerated type called "Members".

The "selectedBandMember" variable is of type "Members", and can ONLY have a value of "JERRY", "BOBBY", or "PHIL".

No need to worry about this variable's value!

The syntax to refer to an enum "instance".

### Your enum extends java.lang.Enum

When you create an enum, you're creating a new class, and ***you're implicitly extending `java.lang.Enum`.*** You can declare an enum as its own standalone class, in its own source file, or as a member of another class.

### Using "if" and "switch" with Enums

Using the enum we just created, we can perform branches in our code using either the `if` or `switch` statement. Also notice that we can compare enum instances using either == or the `.equals()` method. Usually == is considered better style.

```
Members n = Members.BOBBY;
if (n.equals(Members.JERRY)) System.out.println("Jerrrry!");
if (n == Members.BOBBY) System.out.println("Rat Dog");

Members ifName = Members.PHIL;
switch (ifName) {
  case JERRY: System.out.print("make it sing ");
  case PHIL: System.out.print("go deep ");
  case BOBBY: System.out.println("Cassidy! ");
}
```

Assigning an enum value to a variable.

Both of these work fine! "Rat Dog" is printed.

Pop Quiz! What's the output?

Answer: ¡Apissed deep ob

# #1 Enumerations, completed

### A really tricked-out version of a similar enum

You can add a bunch of things to your enum like a constructor, methods, variables, and something called a constant-specific class body. They're not common, but you might run into them:

```
public class HfjEnum {

  enum Names {
    JERRY("lead guitar") { public String sings() {
                           return "plaintively"; }
                         },
    BOBBY("rhythm guitar") { public String sings() {
                             return "hoarsely"; }
                           },
    PHIL("bass");

    private String instrument;

    Names(String instrument) {
      this.instrument = instrument;
    }
    public String getInstrument() {
      return this.instrument;
    }
    public String sings() {
      return "occasionally";
    }
  }

  public static void main(String [] args) {
    for (Names n : Names.values()) {
      System.out.print(n);
      System.out.print(", instrument: "+ n.getInstrument());
      System.out.println(", sings: " + n.sings());
    }
  }
}
```

This is an argument passed in to the constructor declared below.

These are the so-called "constant-specific class bodies". Think of them as overriding the basic enum method (in this case the "sing()" method), if sing() is called on a variable with an enum value of JERRY or BOBBY.

This is the enum's constructor. It runs once for each declared enum value (in this case it runs three times).

You'll see these methods being called from "main()".

Every enum comes with a built-in "values()" method which is typically used in a "for" loop as shown.

```
File Edit Window Help Bootleg
%java HfjEnum

JERRY, instrument: lead guitar, sings: plaintively
BOBBY, instrument: rhythm guitar, sings: hoarsely
PHIL, instrument: bass, sings: occasionally
%
```

Notice that the basic "sing()" method is only called when the enum value has no constant-specific class body.

Five-Minute Mystery

## A Long Trip Home

Captain Byte of the Flatland starship "Traverser" had received an urgent, Top Secret transmission from headquarters. The message contained 30 heavily encrypted navigational codes that the Traverser would need to successfully plot a course home through enemy sectors. The enemy Hackarians, from a neighboring galaxy, had devised a devilish code-scrambling ray that was capable of creating bogus objects on the heap of the Traverser's only navigational computer. In addition, the alien ray could alter valid reference variables so that they referred to these bogus objects. The only defense the Traverser crew had against this evil Hackarian ray was to run an inline virus checker which could be imbedded into the Traverser's state of the art Java 1.4 code.

Captain Byte gave Ensign Smith the following programming instructions to process the critical navigational codes:

"Put the first five codes in an array of type ParsecKey. Put the last 25 codes in a five by five, two dimensional array of type QuadrantKey. Pass these two arrays into the plotCourse() method of the public final class ShipNavigation. Once the course object is returned run the inline virus checker against all the programs reference variables and then run the NavSim program and bring me the results."

A few minutes later Ensign Smith returned with the NavSim output. "NavSim output ready for review, sir", declared Ensign Smith. "Fine", replied the Captain, "Please review your work". "Yes sir!", responded the Ensign, "First I declared and constructed an array of type ParsecKey with the following code; ParsecKey [] p = new ParsecKey[5]; , next I declared and constructed an array of type QuadrantKey with the following code: QuadrantKey [] [] q = new QuadrantKey [5] [5]; . Next, I loaded the first 5 codes into the ParsecKey array using a 'for' loop, and then I loaded the last 25 codes into the QuadrantKey array using nested 'for' loops. Next, I ran the virus checker against all 32 reference variables, 1 for the ParsecKey array, and 5 for its elements, 1 for the QuadrantKey array, and 25 for its elements. Once the virus check returned with no viruses detected, I ran the NavSim program and re-ran the virus checker, just to be safe... Sir ! "

Captain Byte gave the Ensign a cool, long stare and said calmly, "Ensign, you are confined to quarters for endangering the safety of this ship, I don't want to see your face on this bridge again until you have properly learned your Java! Lieutenant Boolean, take over for the Ensign and do this job correctly!"

Why did the captain confine the Ensign to his quarters?

# Five-Minute Mystery Solution

## A Long Trip Home

Captain Byte knew that in Java, multidimensional arrays are actually arrays of arrays. The five by five QuadrantKey array 'q', would actually need a total of 31 reference variables to be able to access all of its components:

1 - reference variable for 'q'

5 - reference variables for q[0] - q[4]

25 - reference variables for q[0][0] - q[4][4]

The ensign had forgotten the reference variables for the five one dimensional arrays embedded in the 'q' array. Any of those five reference variables could have been corrupted by the Hackarian ray, and the ensign's test would never reveal the problem.

# This isn't goodbye

**Bring your brain over to *wickedlysmart.com***