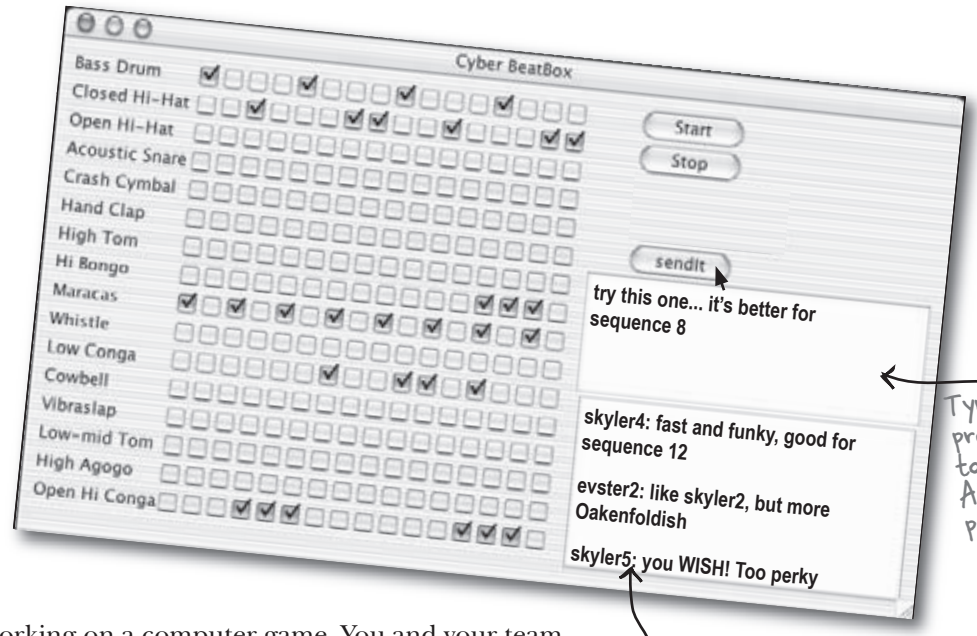
Make a Connection

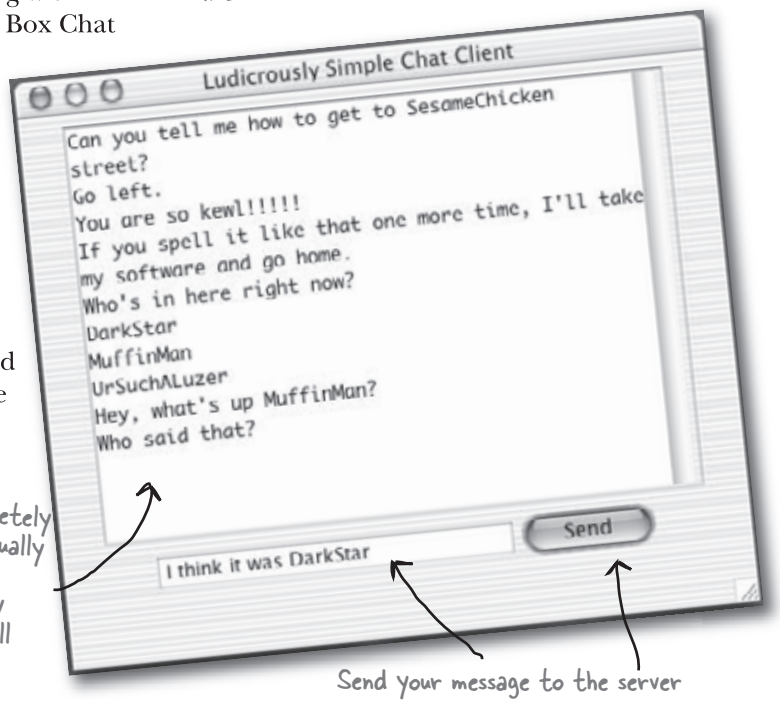
Connect with the outside world. Your Java program can reach out and touch a program on another machine. It's easy. All the low-level networking details are taken care of by classes in the java.net library. One of Java's big benefits is that sending and receiving data over a network is just I/O with a slightly different connection stream at the end of the chain. If you've got a BufferedReader, you can *read*. And the BufferedReader could care less if the data came out of a file or flew down an ethernet cable. In this chapter we'll connect to the outside world with sockets. We'll make *client* sockets. We'll make *server* sockets. We'll make *clients* and *servers*. And we'll make them talk to each other. Before the chapter's done, you'll have a fully-functional, multithreaded chat client. Did we just say *multithreaded*? Yes, now you *will* learn the secret of how to talk to Bob while simultaneously listening to Suzy.

Real-time Beat Box Chat



You're working on a computer game. You and your team are doing the sound design for each part of the game. Using a 'chat' version of the Beat Box, your team can collaborate—you can send a beat pattern along with your chat message, and everybody in the Beat Box Chat gets it. So you don't just get to *read* the other participants' messages, you get to load and *play* a beat pattern simply by clicking the message in the incoming messages area.

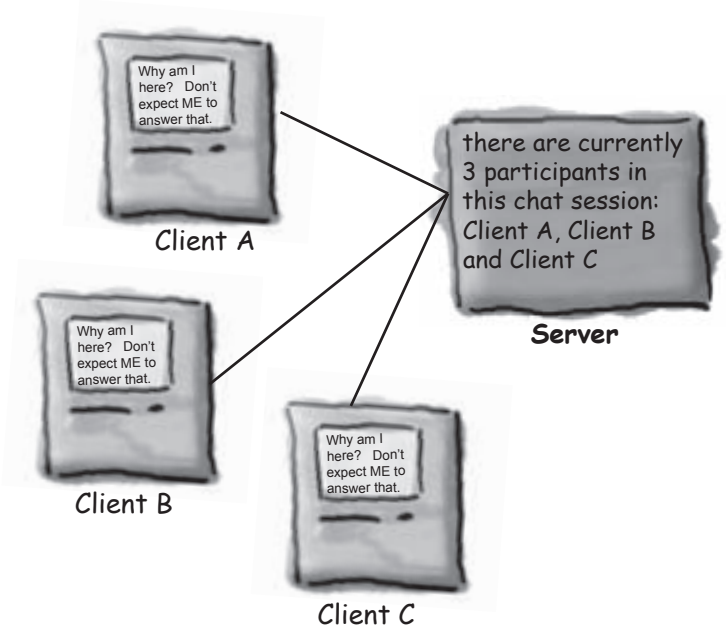
In this chapter we're going to learn what it takes to make a chat client like this. We're even going to learn a little about making a chat *server*. We'll save the full Beat Box Chat for the Code Kitchen, but in this chapter you *will* write a Ludicrously Simple Chat Client and Very Simple Chat Server that send and receive text messages.



Chat Program Overview

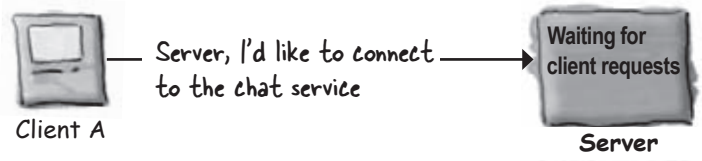
The Client has to know about the Server.

The Server has to know about ALL the Clients.

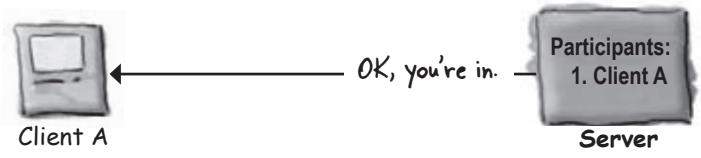


How it Works:

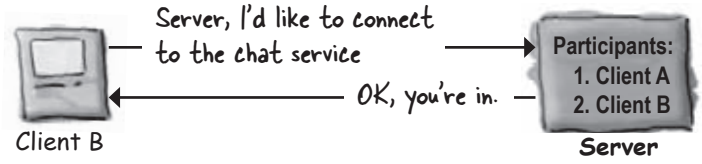
1. Client connects to the server



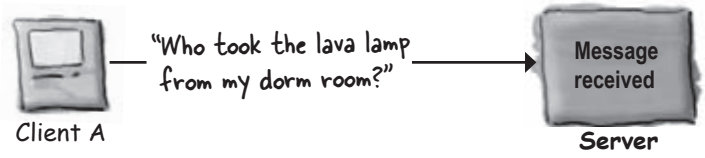
2. The server makes a connection and adds the client to the list of participants



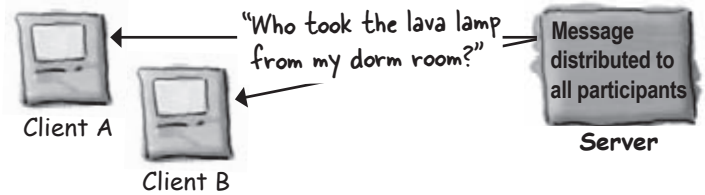
3. Another client connects



4. Client A sends a message to the chat service



5. The server distributes the message to ALL participants (including the original sender)



Connecting, Sending, and Receiving

The three things we have to learn to get the client working are :

1) How to establish the initial **connection** between the client and server

2) How to **send** messages *to* the server

3) How to **receive** messages *from* the server

There's a lot of low-level stuff that has to happen for these things to work. But we're lucky, because the Java API networking package (java.net) makes it a piece of cake for programmers. You'll see a lot more GUI code than networking and I/O code.

And that's not all.

Lurking within the simple chat client is a problem we haven't faced so far in this book: doing two things at the same time. Establishing a connection is a one-time operation (that either works or fails). But after that, a chat participant wants to *send outgoing messages* and **simultaneously** *receive incoming messages* from the other participants (via the server). Hmmmm... that one's going to take a little thought, but we'll get there in just a few pages.

1 Connect

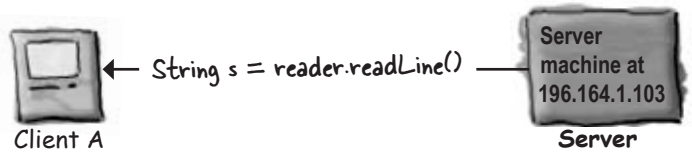
Client connects to the server by establishing a **Socket** connection.

2 Send

Client **sends** a message to the server

3 Receive

Client **gets** a message from the server



Make a network Socket connection

To connect to another machine, we need a Socket connection. A Socket (java.net.Socket class) is an object that represents a network connection between two machines. What's a connection? A *relationship* between two machines, where **two pieces of software know about each other**. Most importantly, those two pieces of software know how to *communicate* with each other. In other words, how to send *bits* to each other.

We don't care about the low-level details, thankfully, because they're handled at a much lower place in the 'networking stack'. If you don't know what the 'networking stack' is, don't worry about it. It's just a way of looking at the layers that information (bits) must travel through to get from a Java program running in a JVM on some OS, to physical hardware (ethernet cables, for example), and back again on some other machine. *Somebody* has to take care of all the dirty details. But not you. That somebody is a combination of OS-specific software and the Java networking API. The part that you have to worry about is high-level—make that *very* high-level—and shockingly simple. Ready?

To make a Socket connection, you need to know two things about the server: who it is, and which port it's running on.

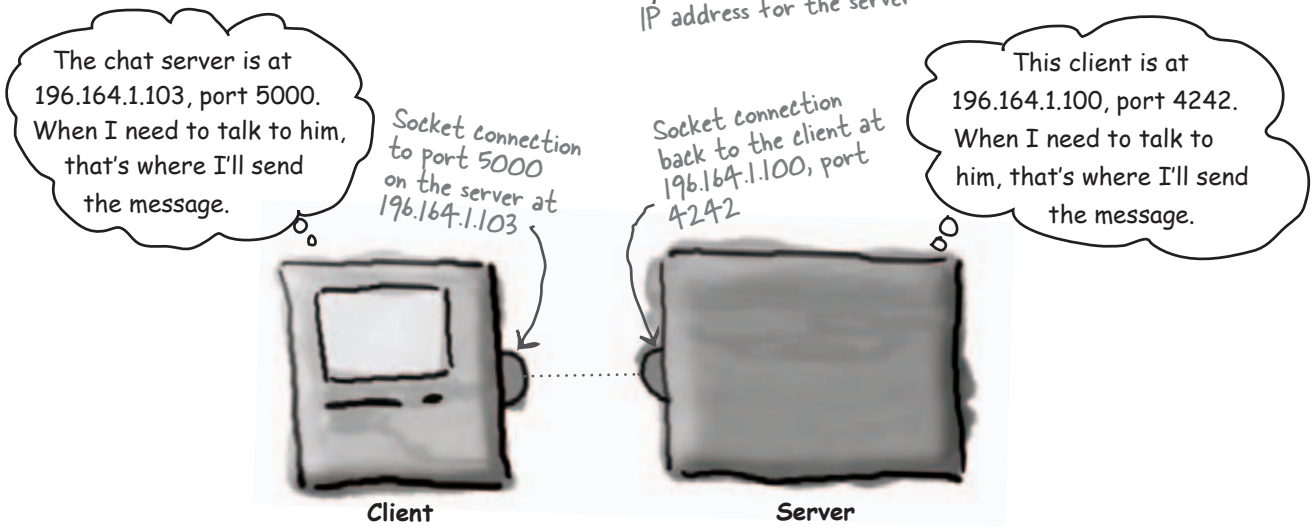
In other words,

IP address and TCP port number.

TCP port number

```
Socket chatSocket = new Socket("196.164.1.103", 5000);
```

IP address for the server



A Socket connection means the two machines have information about each other, including network location (IP address) and TCP port.

A TCP port is just a number. A 16-bit number that identifies a specific program on the server.

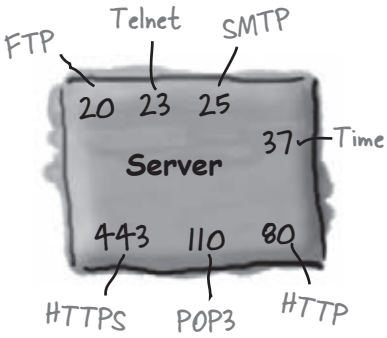
Your internet web (HTTP) server runs on port 80. That's a standard. If you've got a Telnet server, its running on port 23. FTP? 20. POP3 mail server? 110. SMTP? 25. The Time server sits at 37. Think of port numbers as unique identifiers. They represent a logical connection to a particular piece of software running on the server. That's it. You can't spin your hardware box around and find a TCP port. For one thing, you have 65536 of them on a server (0 - 65535). So they obviously don't represent a place to plug in physical devices. They're just a number representing an application.

Without port numbers, the server would have no way of knowing which application a client wanted to connect to. And since each application might have its own unique protocol, think of the trouble you'd have without these identifiers. What if your web browser, for example, landed at the POP3 mail server instead of the HTTP server? The mail server won't know how to parse an HTTP request! And even if it did, the POP3 server doesn't know anything about servicing the HTTP request.

When you write a server program, you'll include code that tells the program which port number you want it to run on (you'll see how to do this in Java a little later in this chapter). In the Chat program we're writing in this chapter, we picked 5000. Just because we wanted to. And because it met the criteria that it be a number between 1024 and 65535. Why 1024? Because 0 through 1023 are reserved for the well-known services like the ones we just talked about.

And if you're writing services (server programs) to run on a company network, you should check with the sys-admins to find out which ports are already taken. Your sys-admins might tell you, for example, that you can't use any port number below, say, 3000. In any case, if you value your limbs, you won't assign port numbers with abandon. Unless it's your *home* network. In which case you just have to check with your *kids*.

Well-known TCP port numbers for common server applications



A server can have up to 65536 different server apps running, one per port.

The TCP port numbers from 0 to 1023 are reserved for well-known services. Don't use them for your own server programs!*

The chat server we're writing uses port 5000. We just picked a number between 1024 and 65535.

*Well, you *might* be able to use one of these, but the sys-admin where you work will probably kill you.

there are no Dumb Questions

Q: How do you know the port number of the server program you want to talk to?

A: That depends on whether the program is one of the well-known services. If you're trying to connect to a well-known service, like the ones on the opposite page (HTTP, SMTP, FTP, etc.) you can look these up on the internet (Google "Well-Known TCP Port"). Or ask your friendly neighborhood sys-admin.

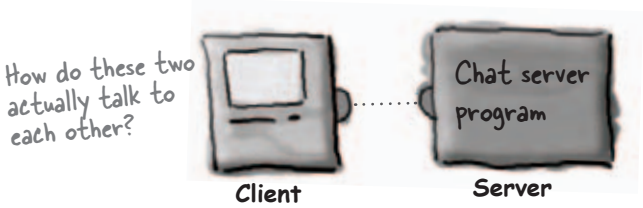
But if the program isn't one of the well-known services, you need to find out from whoever is deploying the service. Ask him. Or her. Typically, if someone writes a network service and wants others to write clients for it, they'll publish the IP address, port number, and protocol for the service. For example, if you want to write a client for a GO game server, you can visit one of the GO server sites and find information about how to write a client for that particular server.

Q: Can there ever be more than one program running on a single port? In other words, can two applications on the same server have the same port number?

A: No! If you try to bind a program to a port that is already in use, you'll get a BindException. To *bind* a program to a port just means starting up a server application and telling it to run on a particular port. Again, you'll learn more about this when we get to the server part of this chapter.

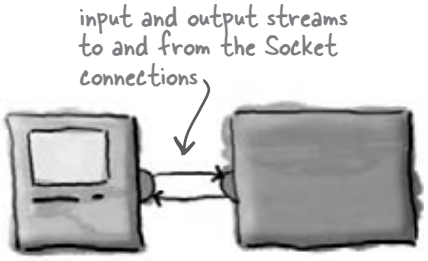
Brain Barbell

OK, you got a Socket connection. The client and the server know the IP address and TCP port number for each other. Now what? How do you communicate over that connection? In other words, how do you move bits from one to the other? Imagine the kinds of messages your chat client needs to send and receive.



To read data from a Socket, use a BufferedReader

To communicate over a Socket connection, you use streams. Regular old I/O streams, just like we used in the last chapter. One of the coolest features in Java is that most of your I/O work won't care what your high-level chain stream is actually connected to. In other words, you can use a BufferedReader just like you did when you were writing to a file, the difference is that the underlying connection stream is connected to a *Socket* rather than a *File!*



1 Make a Socket connection to the server

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

The port number, which you know because we TOLD you that 5000 is the port number for our chat server.

127.0.0.1 is the IP address for "localhost", in other words, the one this code is running on. You can use this when you're testing your client and server on a single, stand-alone machine.

2 Make an InputStreamReader chained to the Socket's low-level (connection) input stream

```
InputStreamReader stream = new InputStreamReader(chatSocket.getInputStream());
```

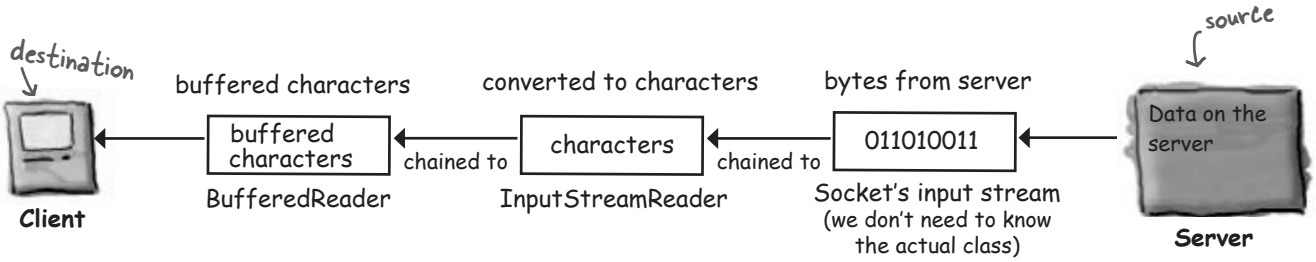
InputStreamReader is a 'bridge' between a low-level byte stream (like the one coming from the Socket) and a high-level character stream (like the BufferedReader we're after as our top of the chain stream).

All we have to do is ASK the socket for an input stream! It's a low-level connection stream, but we're just gonna chain it to something more text-friendly.

3 Make a BufferedReader and read!

Chain the BufferedReader to the InputStreamReader(which was chained to the low-level connection stream we got from the Socket.)

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```



To write data to a Socket, use a PrintWriter

We didn't use PrintWriter in the last chapter, we used BufferedWriter. We have a choice here, but when you're writing one String at a time, PrintWriter is the standard choice. And you'll recognize the two key methods in PrintWriter, print() and println()! Just like good ol' System.out.

this part's the same as it was on the opposite page -- to write to the server, we still have to connect to it.

1 Make a Socket connection to the server

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

2 Make a PrintWriter chained to the Socket's low-level (connection) output stream

```
PrintWriter writer = new PrintWriter(chatSocket.getOutputStream());
```

PrintWriter acts as its own bridge between character data and the bytes it gets from the Socket's low-level output stream. By chaining a PrintWriter to the Socket's output stream, we can write Strings to the Socket connection.

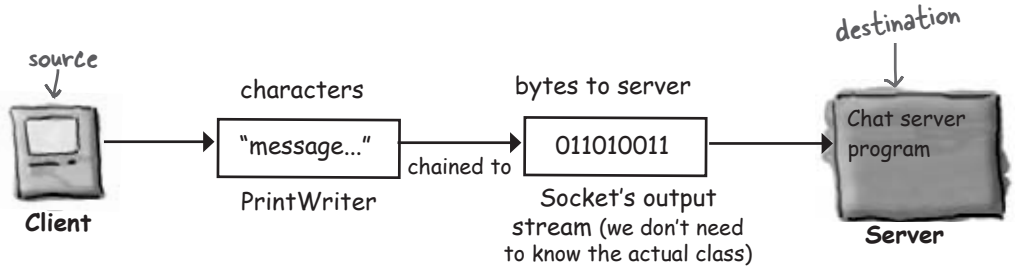
The Socket gives us a low-level connection stream and we chain it to the PrintWriter by giving it to the PrintWriter constructor.

3 Write (print) something

```
writer.println("message to send");
writer.print("another message");
```

println() adds a new line at the end of what it sends.

print() doesn't add the new line.



The DailyAdviceClient

Before we start building the Chat app, let's start with something a little smaller. The Advice Guy is a server program that offers up practical, inspirational tips to get you through those long days of coding.

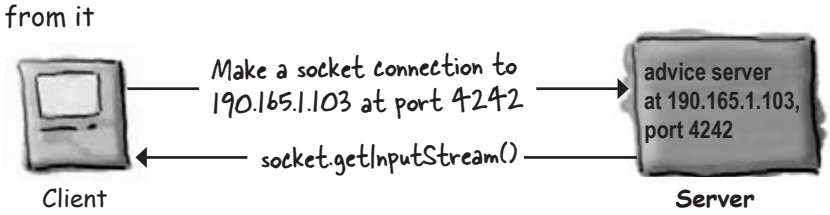
We're building a client for The Advice Guy program, which pulls a message from the server each time it connects.

What are you waiting for? Who *knows* what opportunities you've missed without this app.

The Advice Guy

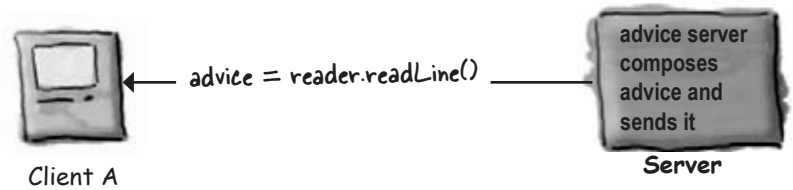
1 Connect

Client connects to the server and gets an input stream from it



2 Read

Client reads a message from the server



DailyAdviceClient code

This program makes a Socket, makes a BufferedReader (with the help of other streams), and reads a single line from the server application (whatever is running at port 4242).

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

public class DailyAdviceClient {

   public void go() {
        try {
            Socket s = new Socket("127.0.0.1", 4242);

            InputStreamReader streamReader = new InputStreamReader(s.getInputStream());
            BufferedReader reader = new BufferedReader(streamReader);

            String advice = reader.readLine();
            System.out.println("Today you should: " + advice);

            reader.close();

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

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

class Socket is in java.net

a lot can go wrong here

make a Socket connection to whatever is running on port 4242, on the same host this code is running on. (The 'localhost')

chain a BufferedReader to an InputStreamReader to the input stream from the Socket.

this readLine() is EXACTLY the same as if you were using a BufferedReader chained to a FILE.. In other words, by the time you call a BufferedWriter method, the writer doesn't know or care where the characters came from.

this closes ALL the streams

Sharpen your pencil

Test your memory of the streams/classes for reading and writing from a Socket. Try not to look at the opposite page!

To ***read*** text from a Socket:

Client

write/draw in the chain of streams the client uses to read from the server

source

Server

To ***send*** text to a Socket:

Client

write/draw in the chain of streams the client uses to send something to the server

destination

Server

Sharpen your pencil

Fill in the blanks:

What two pieces of information does the client need in order to make a Socket connection with a server? _____ _____

Which TCP port numbers are reserved for 'well-known services' like HTTP and FTP? _____

TRUE or FALSE: The range of valid TCP port numbers can be represented by a short primitive? _____

Writing a simple server

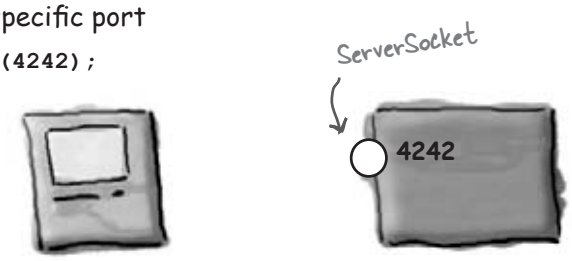
So what's it take to write a server application? Just a couple of Sockets. Yes, a couple as in *two*. A ServerSocket, which waits for client requests (when a client makes a new Socket()) and a plain old Socket socket to use for communication with the client.

How it Works:

1 Server application makes a ServerSocket, on a specific port

```
ServerSocket serverSock = new ServerSocket(4242);
```

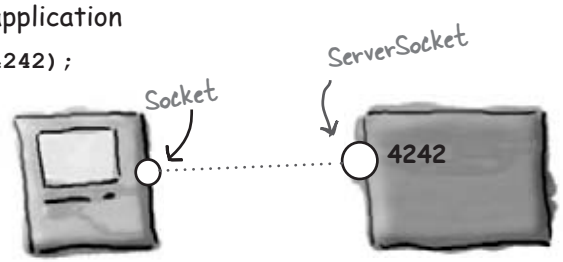
This starts the server application listening for client requests coming in for port 4242.



2 Client makes a Socket connection to the server application

```
Socket sock = new Socket("190.165.1.103", 4242);
```

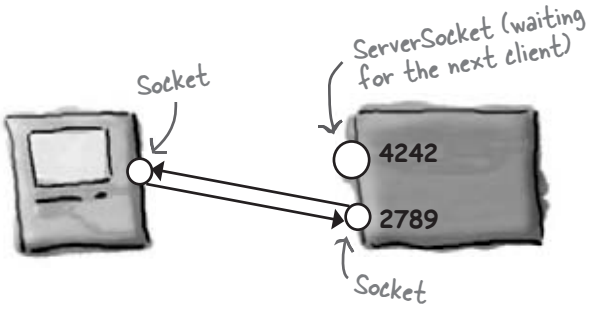
Client knows the IP address and port number (published or given to him by whomever configures the server app to be on that port)



3 Server makes a new Socket to communicate with this client

```
Socket sock = serverSock.accept();
```

The accept() method blocks (just sits there) while it's waiting for a client Socket connection. When a client finally tries to connect, the method returns a plain old Socket (on a *different* port) that knows how to communicate with the client (i.e., knows the *client's* IP address and port number). The Socket is on a different port than the ServerSocket, so that the ServerSocket can go back to waiting for other clients.



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

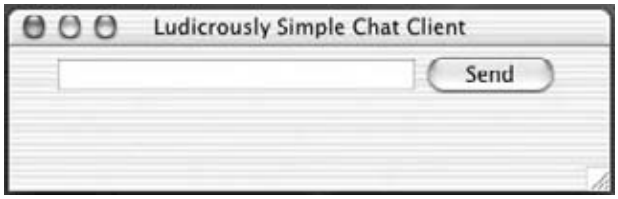
- Client and server applications communicate over a Socket connection.
- A Socket represents a connection between two applications which may (or may not) be running on two different physical machines.
- A client must know the IP address (or domain name) and TCP port number of the server application.
- 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.*;

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

    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

   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
```

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

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

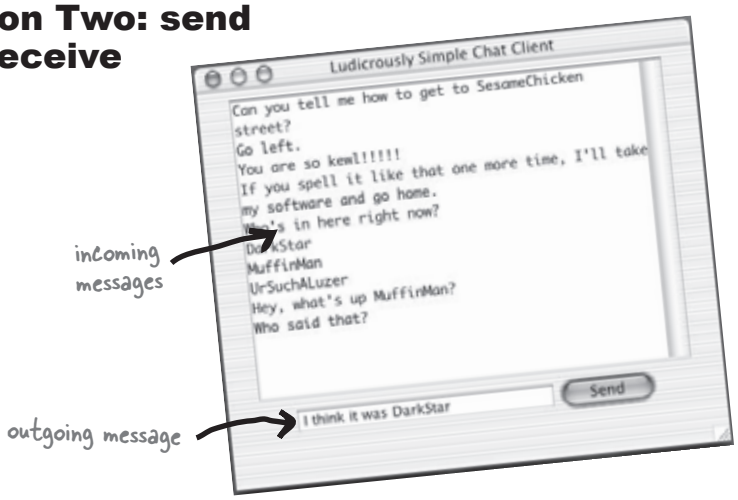
we're using localhost so you can test the client and server on one machine

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

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?

1 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.

2 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?

3 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.

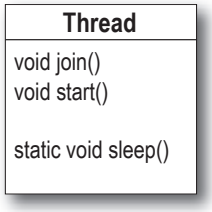
thread

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:

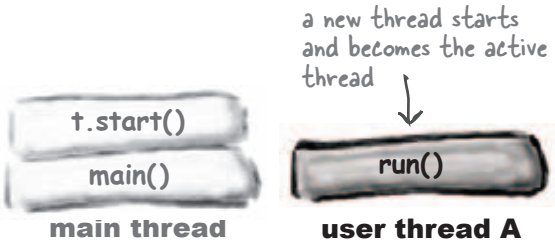
1 **The JVM calls the main() method.**

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

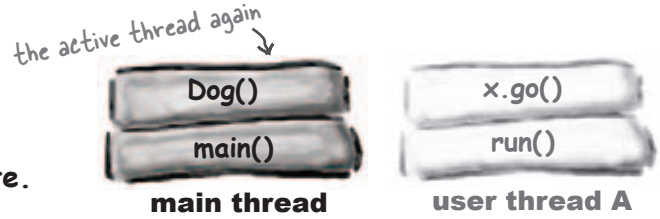
2 **main() starts a new thread. The main thread is temporarily frozen while the new thread starts running.**

```
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)

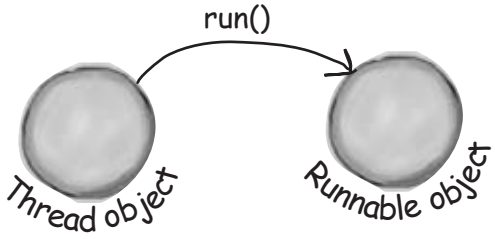
```
Thread myThread = new Thread(threadJob);
```

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");
   }
}
```

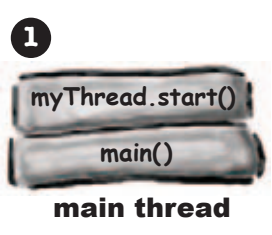
Runnable has only one method to implement: public void run() (with no arguments). This is where you put the JOB the thread is supposed to run. This is the method that goes at the bottom of the new stack.

2

```
class ThreadTester {

   public static void main (String[] args) {

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

1     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.

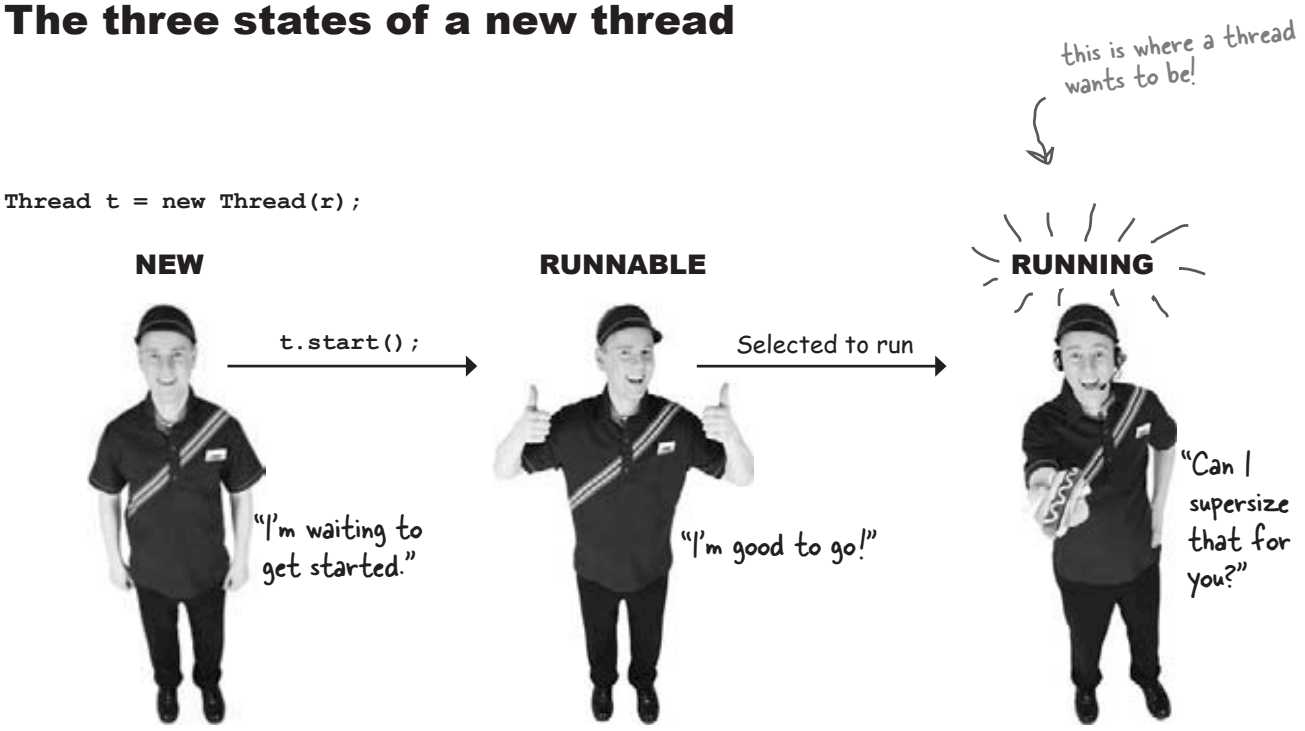
You won't get a new thread of execution until you call start() on the Thread instance. A thread is not really a thread until you start it. Before that, it's just a Thread instance, like any other object, but it won't have any real 'threadness'.



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();`

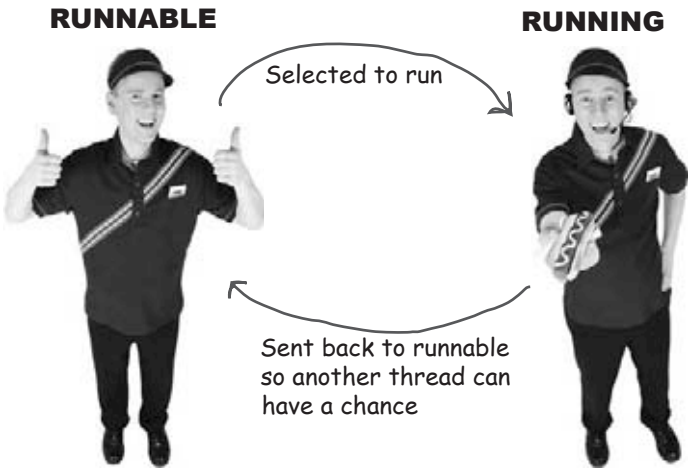
When you start the thread, it moves into the runnable state. This means the thread is ready to run and just waiting for its Big Chance to be selected for execution. At this point, there is a new call stack for this thread.

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

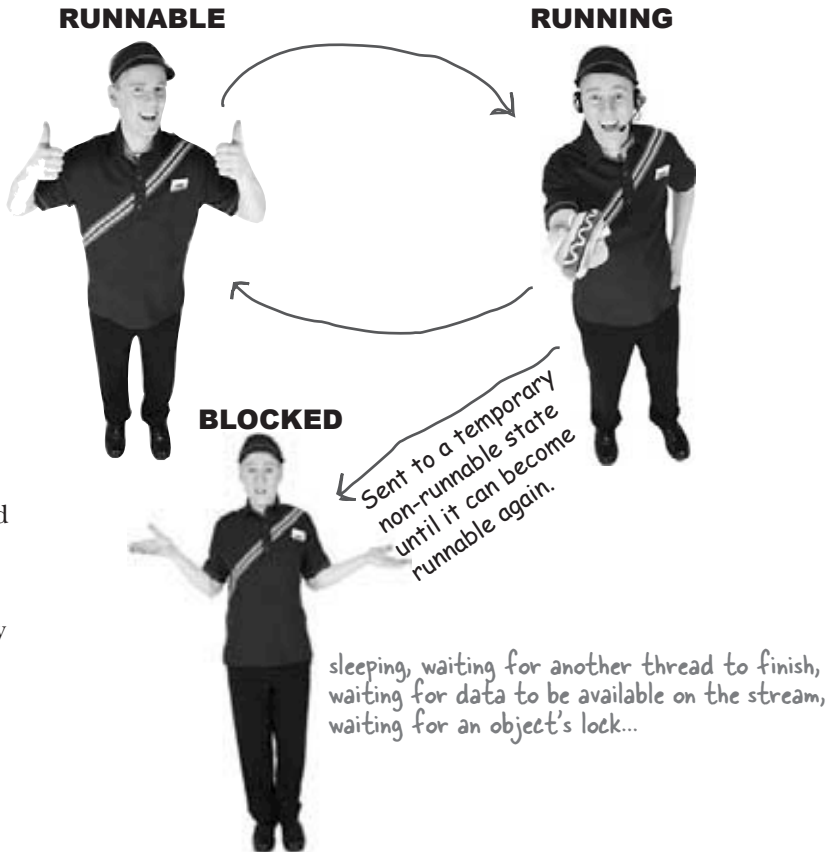
Typically, a thread moves back and forth between runnable and running, as the JVM thread scheduler selects a thread to run and then kicks it back out so another thread gets a chance.



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

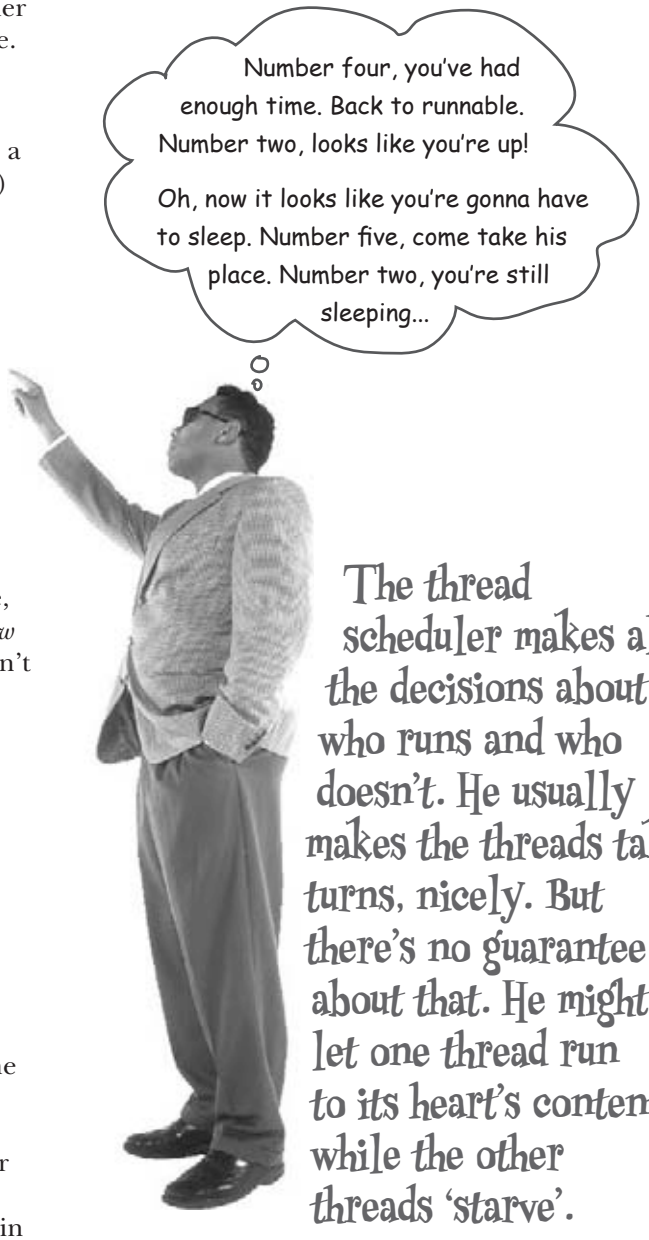
The thread scheduler makes all the decisions about who moves from runnable to running, and about when (and under what circumstances) a thread leaves the running state. The scheduler decides who runs, and for how long, and where the threads go when the scheduler decides to kick them out of the currently-running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

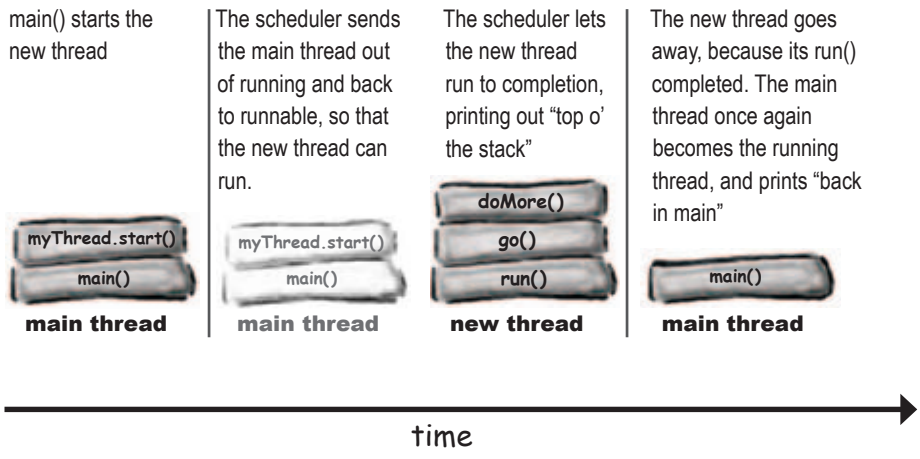
Produced this output:

```
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
```

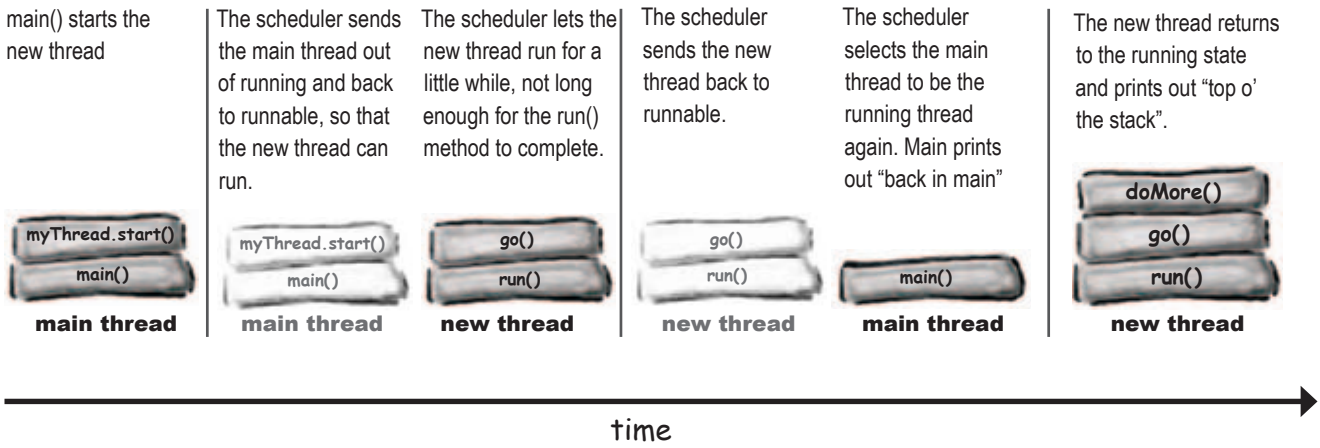
Notice how the order changes randomly. Sometimes the new thread finishes first, and sometimes the main thread finishes first.

How did we end up with different results?

Sometimes it runs like this:



And sometimes it runs like this:



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 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.

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!**

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.

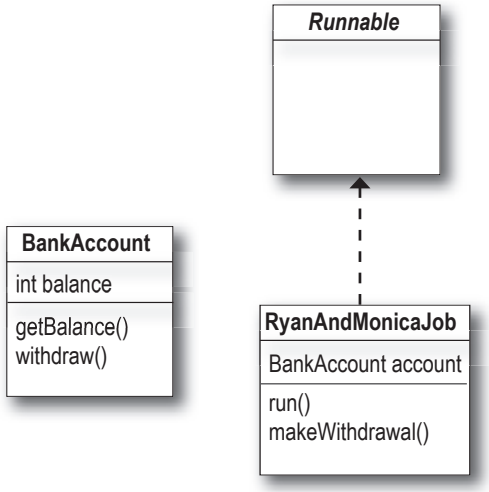
The Ryan and Monica problem, in code

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:



1 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();
```

2 Make two threads with the same Runnable (the RyanAndMonicaJob instance)

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

3 Name and start the threads

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

4 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.

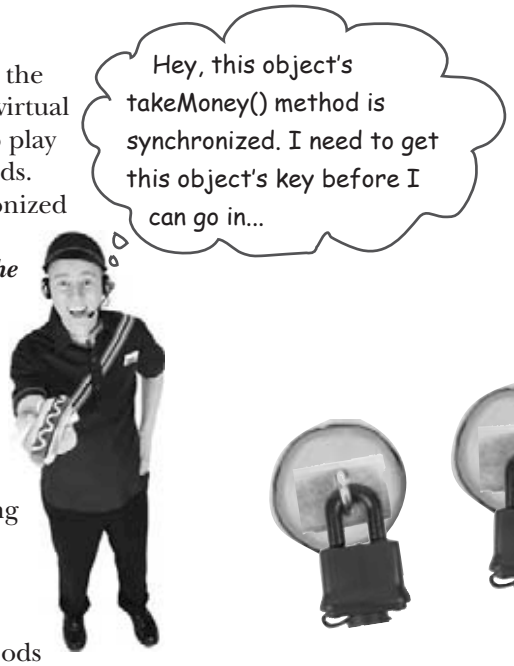
To protect your *data* (like the bank account), synchronize the *methods* that act on that data.

```
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.

doStuff() doesn't need to be synchronized, so we don't synchronize the whole method.

```
public void go() {
   doStuff();

   synchronized(this) {
      criticalStuff();
      moreCriticalStuff();
   }
}
```

Now, only these two method calls are grouped into one atomic unit. When you use the synchronized keyword WITHIN a method, rather than in a method declaration, you have to provide an argument that is the object whose key the thread needs to get.

Although there are other ways to do it, you will almost always synchronize on the current object (this). That's the same object you'd lock if the whole method were synchronized.

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...]

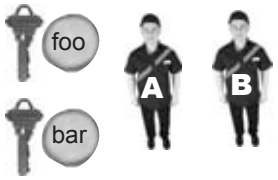
The deadly side of synchronization

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.

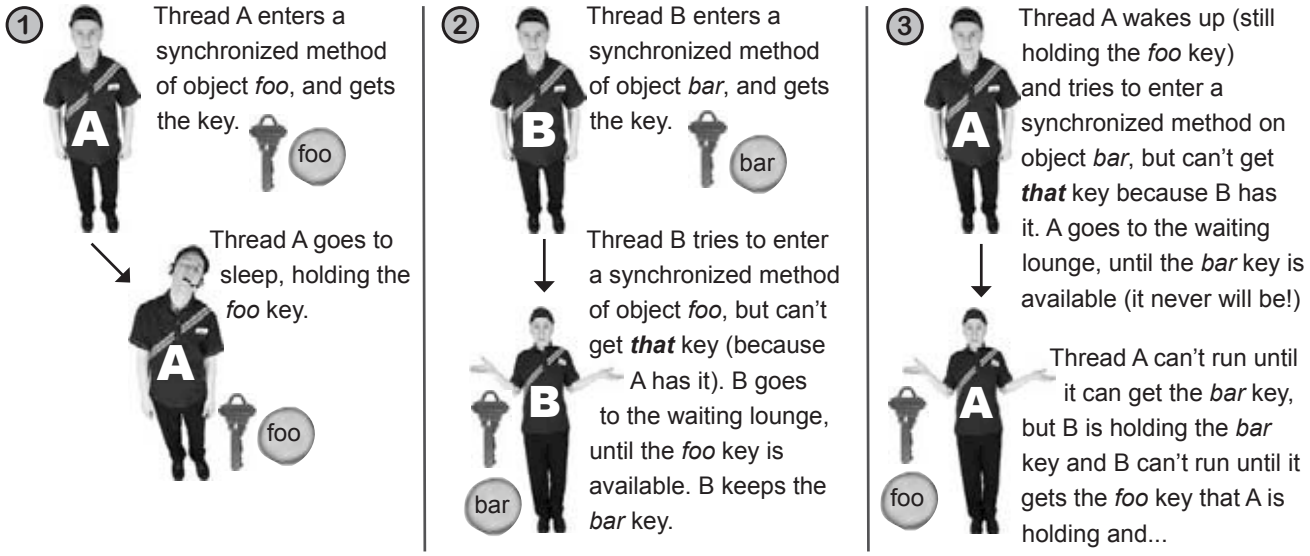
If you're familiar with databases or other application servers, you might recognize the problem; databases often have a locking mechanism somewhat like synchronization. But a real transaction management system can sometimes deal with deadlock. It might assume, for example, that deadlock might have occurred 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() {

        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
```

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

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
```

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).

```
} // close outer class
```

Ready-bake Code

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?

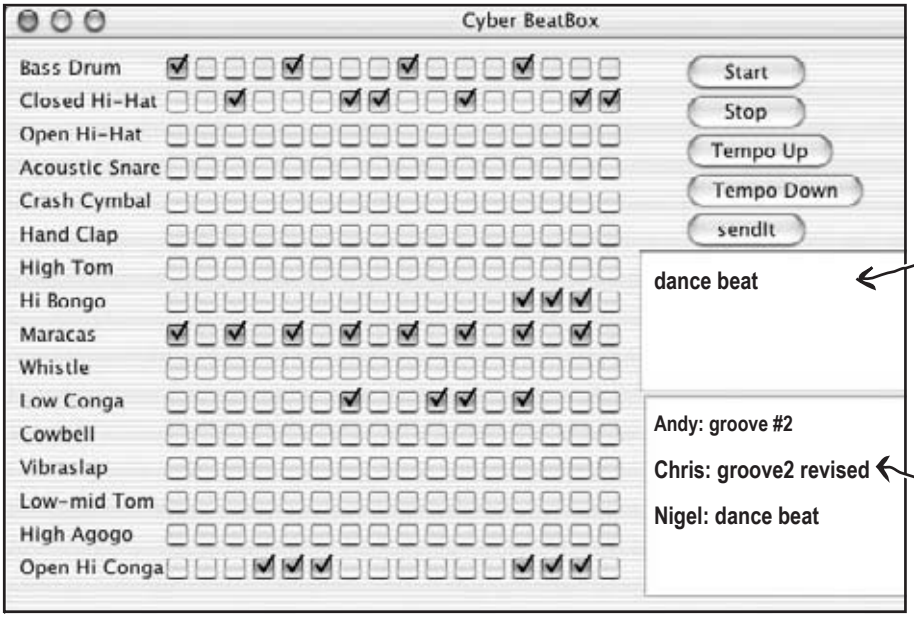
A: Actually, we *should* have synchronized those methods, to prevent other threads from accessing those methods in other ways. We didn't bother, because our example didn't have any other code accessing the account.

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.