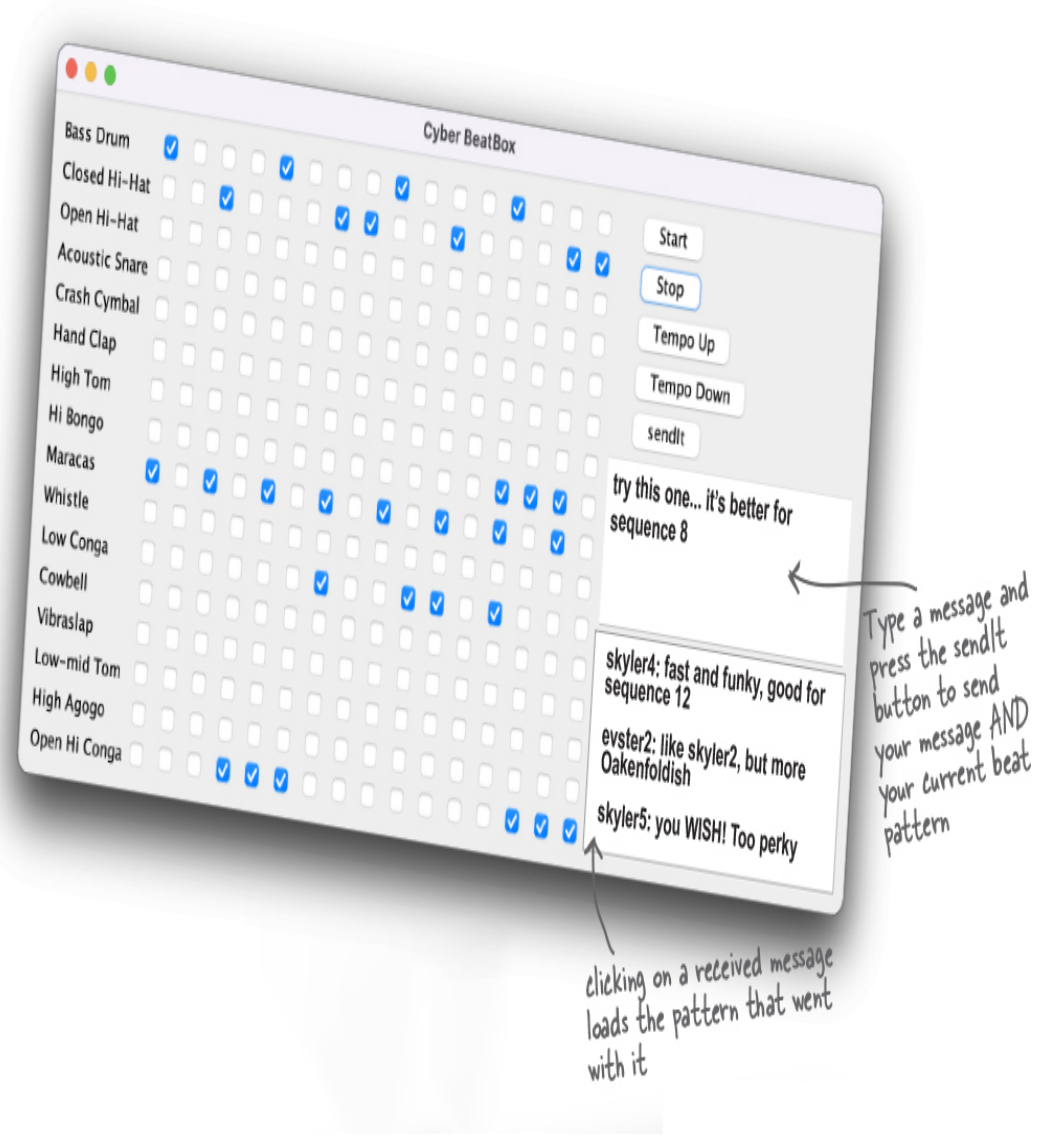
# Chapter 17. Networking and Threads: Make a Connection


**Connect with the outside world.** Your Java program can talk to a program on another machine. It's easy. All the low-level networking details are taken care of by the built in Java libraries. One of Java's big benefits is that sending and receiving data over a network can be just I/O with a slightly

different connection at the end of the I/O chain. In this chapter we'll connect to the outside world with *channels*. We'll make *client* channels. We'll make *server* channels. We'll make *clients* and *servers*, and we'll make them talk to each other. And we'll also have to learn how to do more than one thing at once. 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



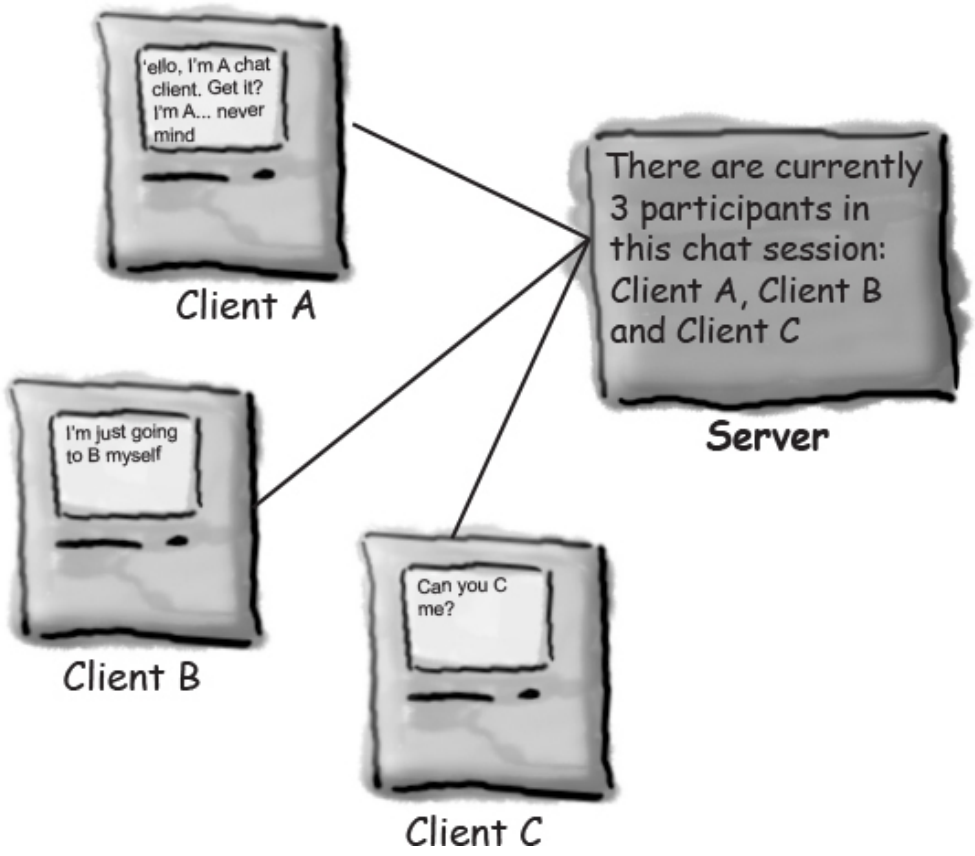

You're working on a computer game. You and your team are doing the sound design for each part of the game. Using a 'chat' version of the Beat Box, your team can collaborate— you can send a beat pattern along with your chat message, and everybody in the Beat Box Chat gets it. So you don't just get to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

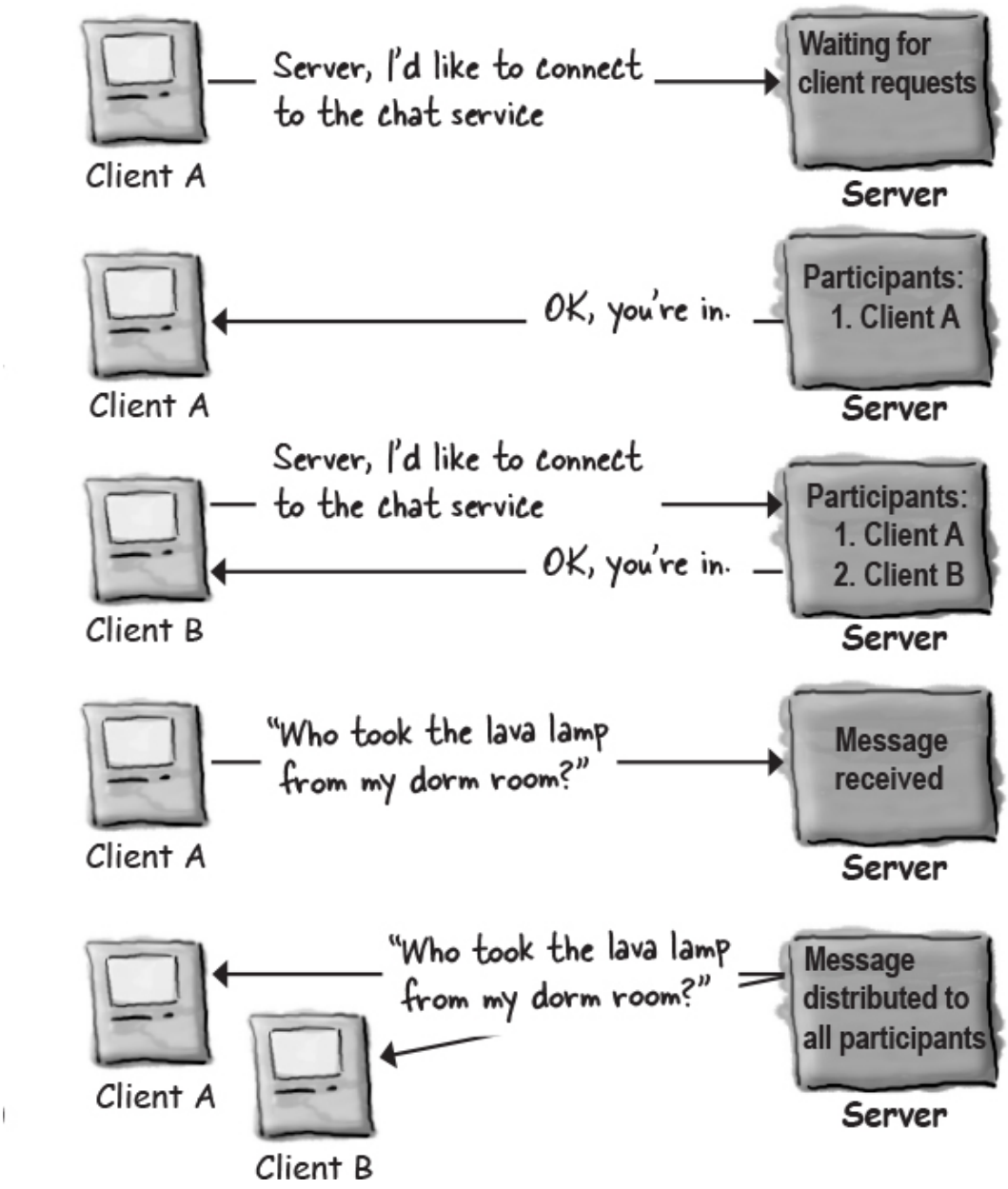

## Chat Program Overview

> **NOTE**
>
> **Each Client has to know about the Server.**
>
> **The Server has to know about ALL the Clients.**




**How it Works:**

1. ❶ Client connects to the server
2. ❷ The server makes a connection and adds the client to the list of participants
3. ❸ Another client connects
4. ❹ Client A sends a message to the chat service
5. ❺ 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 **receive** messages *from* the server

3. How to **send** messages *to* the server

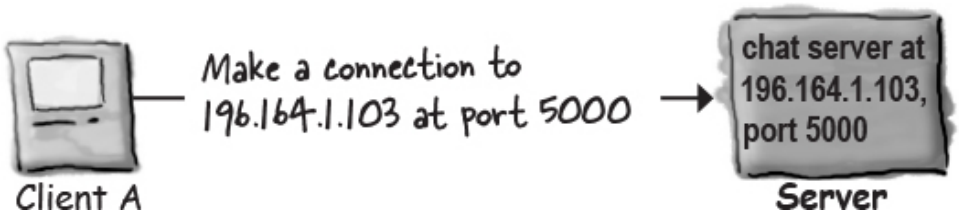
There's a lot of low-level stuff that has to happen for these things to work. But we're lucky, because the Java APIs make it a piece of cake for programmers. You'll see a lot more GUI code than networking and I/O code in this chapter.

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.

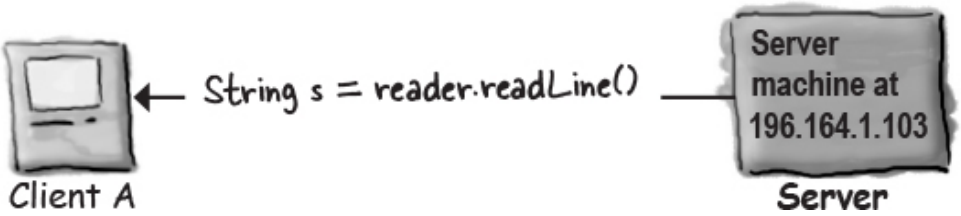
1. ❶ **Connect**

   Client **connects** to the server




2. ❷ **Receive**

   Client **reads** a message from the server



3. ❸ **Send**

   Client **writes** a message to the server


# 1. Connect

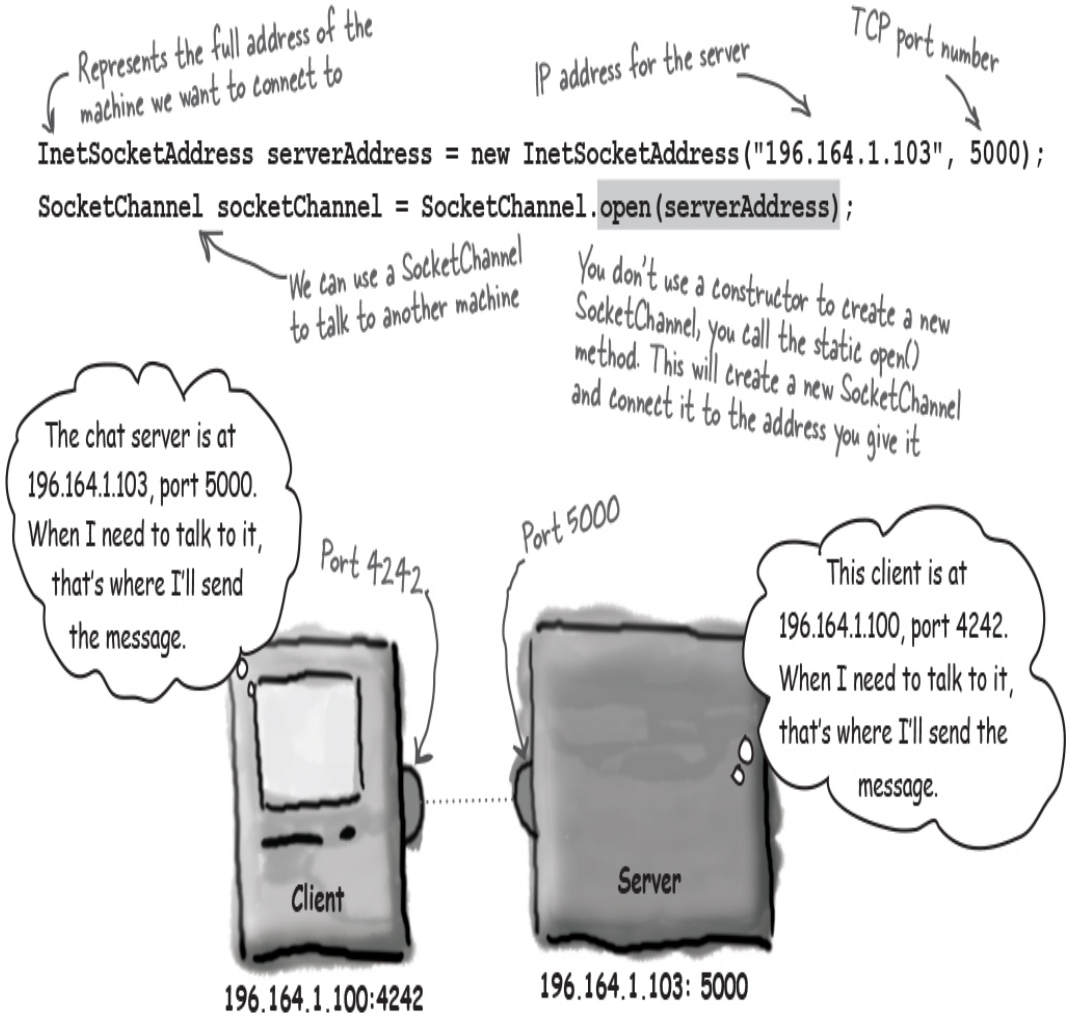
To talk to another machine, we need an object that represents a network connection between two machines. We can open a java.nio.channels. SocketChannel to give us this connection object.

> **NOTE**
>
> **To make a connection, you need to know two things about the server: where it is, and which port it's running on.**
>
> **In other words,**
>
> **IP address and TCP port number.**

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.

The part that you have to worry about is high-level. You just have to create an object for the server's address, then open a channel to that server. Ready?



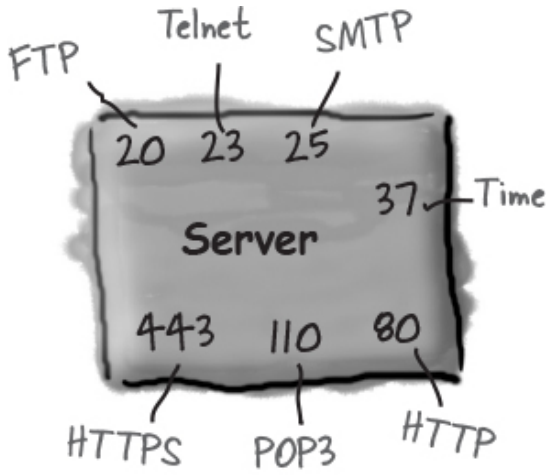

**NOTE**

**A 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.

Well-known TCP port numbers for common server applications:




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

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*.

**NOTE**

**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.**

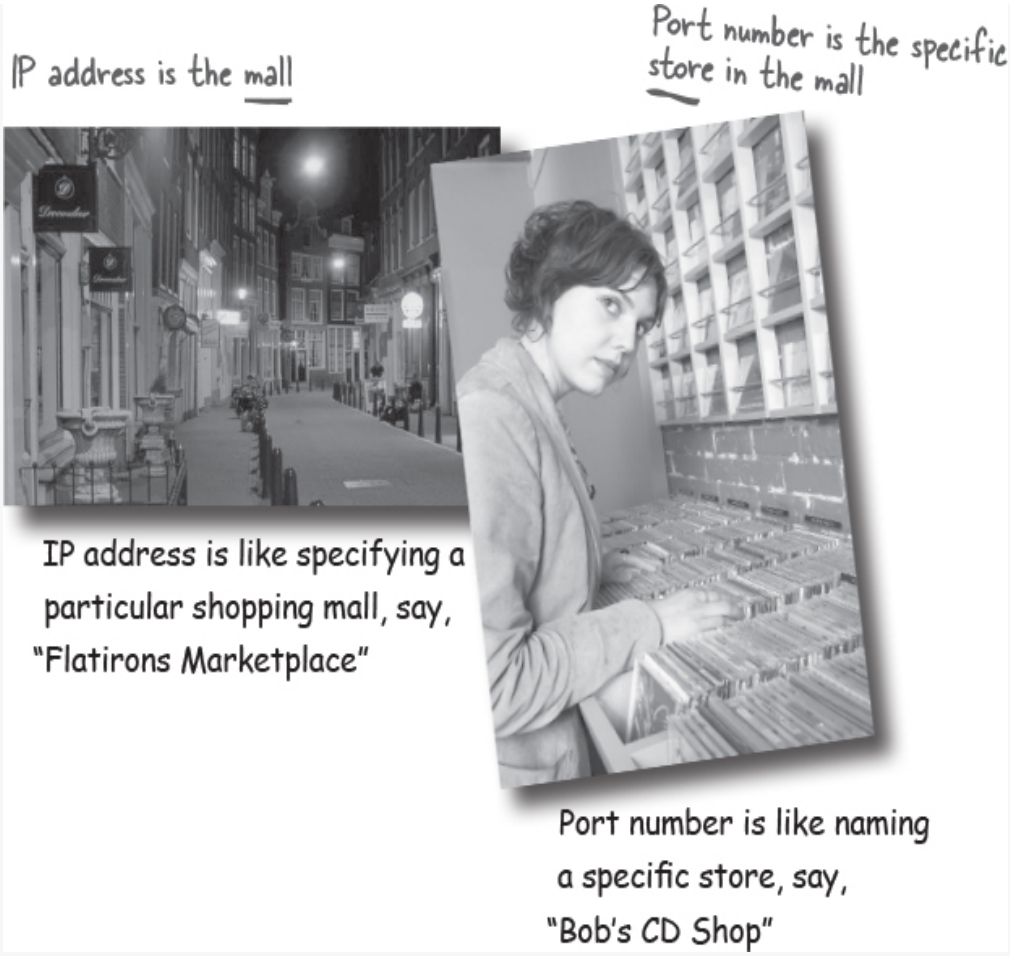
*Well, you *might* be able to use one of these, but the sys-admin where you work will write you a strongly-worded message and CC your boss.

## 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 them. 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.

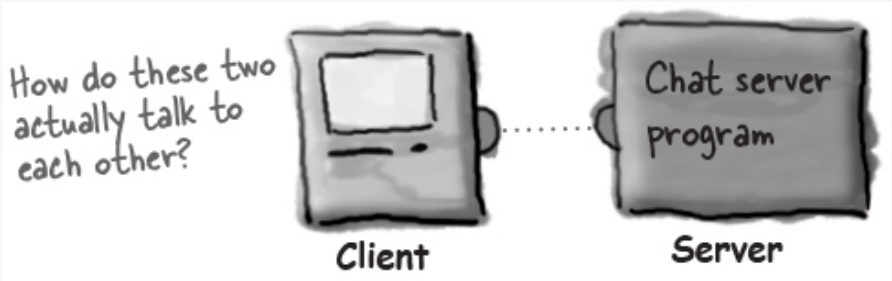



**Q: Can there ever be more than one program running on a single port? In other words, can two applications on the same server have the same port number?**

**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.

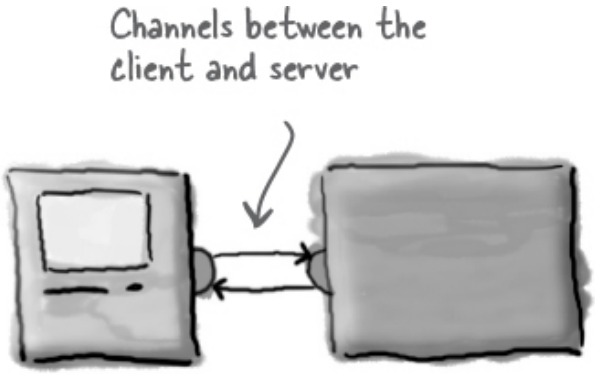
**BRAIN BARBELL**

OK, you got a 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.




## 2. Receive

To communicate over a remote connection, you can use 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 reading from a file, the difference is that the underlying connection stream is connected to a *Channel* rather than a *File!*




**Reading from the network with BufferedReader**

## 1 Make a connection to the 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. You could also use "localhost" here instead.

```
SocketAddress serverAddr = new InetSocketAddress("127.0.0.1", 5000);

SocketChannel socketChannel = SocketChannel.open(serverAddr);
```

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

You need to open a SocketChannel that connects to this address

## 2 Create or get a Reader from the connection

```
Reader reader = Channels.newReader(socketChannel, StandardCharsets.UTF_8);
```

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

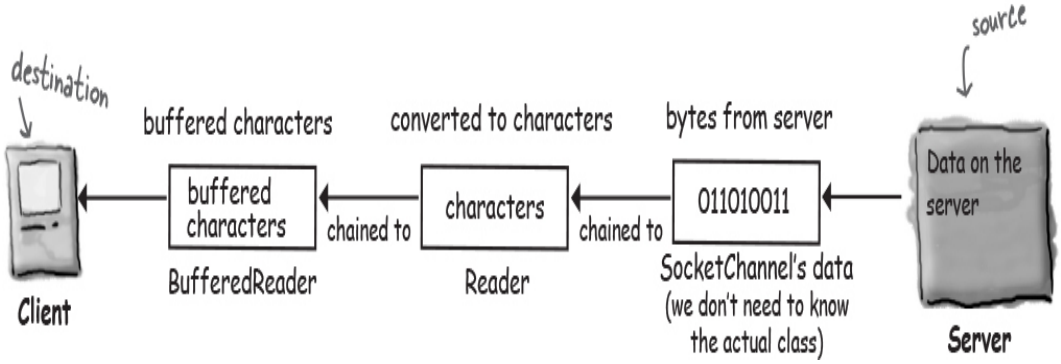
You can use the static helper methods on the Channels class to create a Reader from your SocketChannel

You need to say which Charset to use for reading the values from the network. UTF 8 is a common one to use

## 3 Make a BufferedReader and read!

Chain the BufferedReader to the Reader (which is from our SocketChannel)

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




## 3. Send

In the last chapter, we used BufferedWriter. We have a choice here, but when you're writing one String at a time, **PrintWriter** is a standard choice. And you'll recognize the two key methods in PrintWriter, print() and println()! Just like good ol' System.out.

### Writing to the network with PrintWriter

## 1 Make a connection to the server

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

```
SocketAddress serverAddr = new InetSocketAddress("127.0.0.1", 5000);

SocketChannel socketChannel = SocketChannel.open(serverAddr);
```

## 2 Create or get a Writer from the connection

You need to say which Charset to use to write Strings. You should use the same one for reading as for writing!

```
Writer writer = Channels.newWriter(socketChannel, StandardCharsets.UTF_8);
```

Writer acts as a bridge between character data and the bytes to be written to the Channel

The Channels class contains utility methods to create a Writer

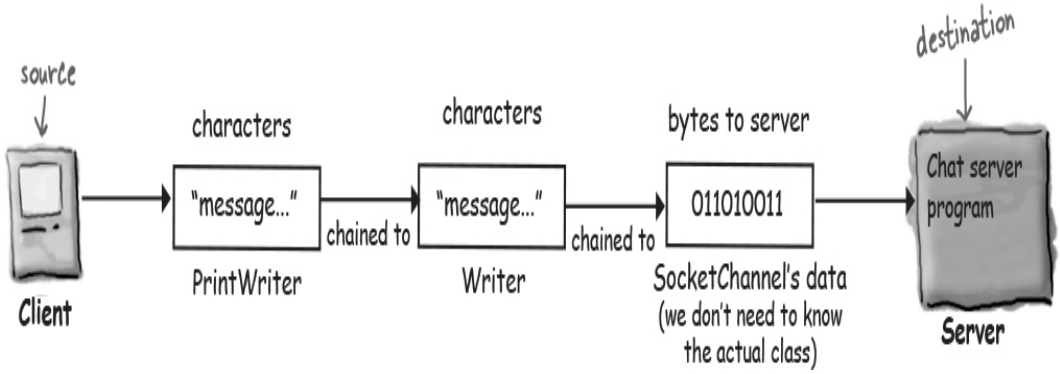
## 3 Make a PrintWriter and write (print) something

By chaining a PrintWriter to the Channel's Writer, we can write Strings to the Channel which will be sent over the connection.

```
PrintWriter printWriter = new PrintWriter(writer);
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.



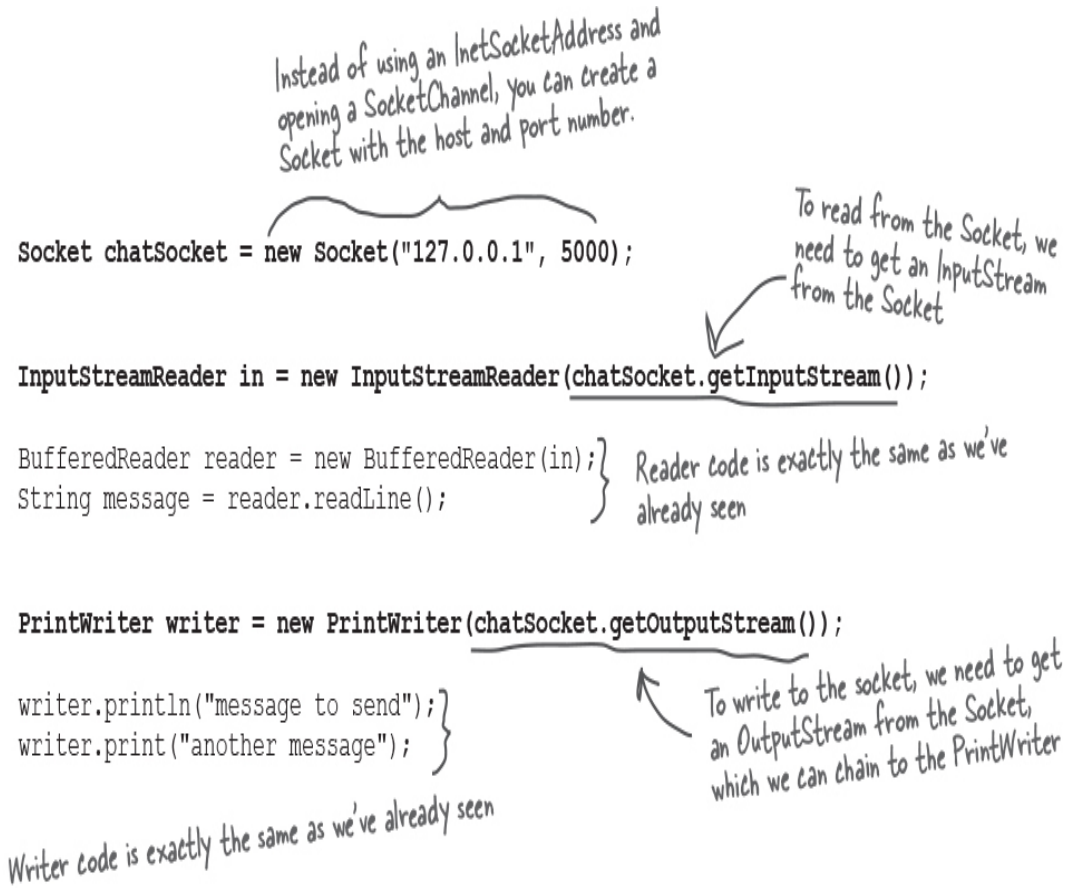

## There's more than one way to make a connection


If you look at real life code that talks to a remote machine, you'll probably see a number of different ways to make connections and to read from and write to a remote computer.

Which approach you use depends on a number of things, including (but not limited to) the version of Java you're using and the needs of the application (for example, how many clients connect at once, the size of messages sent, frequency or message etc). One of the simplest approaches is to use a **java.net.Socket** instead of a Channel.

## Using a Socket

You can get an *InputStream* or *OutputStream* from a Socket, and read and write from it in a very similar way to what we've already seen.



**NOTE**

**The java.net.Socket class is available in all versions of Java.**

**It supports simple network I/O via the I/O streams we've already used for file I/O.**


**As we've become an increasingly connected world, Java has evolved to offer more ways to work with remote machines.**

Remember that Channels are in the java.**nio**.channels package? The java.nio package (NIO) was introduced in Java 1.4, and there were more changes and additions made (sometimes called NIO.2) in Java 7.

There are ways to use Channels and NIO to get better performance when you're working with lots of network connections, or lots of data coming over those connections.

**In this chapter, we're using Channels to provide the same very basic connection functionality we could get from Sockets**. However, if our application needed to work well with a very busy network connection (or lots of them!) we could configure our Channels differently, and use them to their full potential, and our program would cope better with a high network I/O load.

We've chosen to teach you the **simplest way to get started** with network I/O using *Channels*, so that if you need to "level up" to working with more advanced features, it shouldn't be such a big step.

If you do want to learn more about NIO, read Java NIO by Ron Hitchens, and Java I/O, NIO and NIO.2 by Jeff Friesen.

**RELAX**

**Channels support advanced networking features that you don't need for these exercises.**

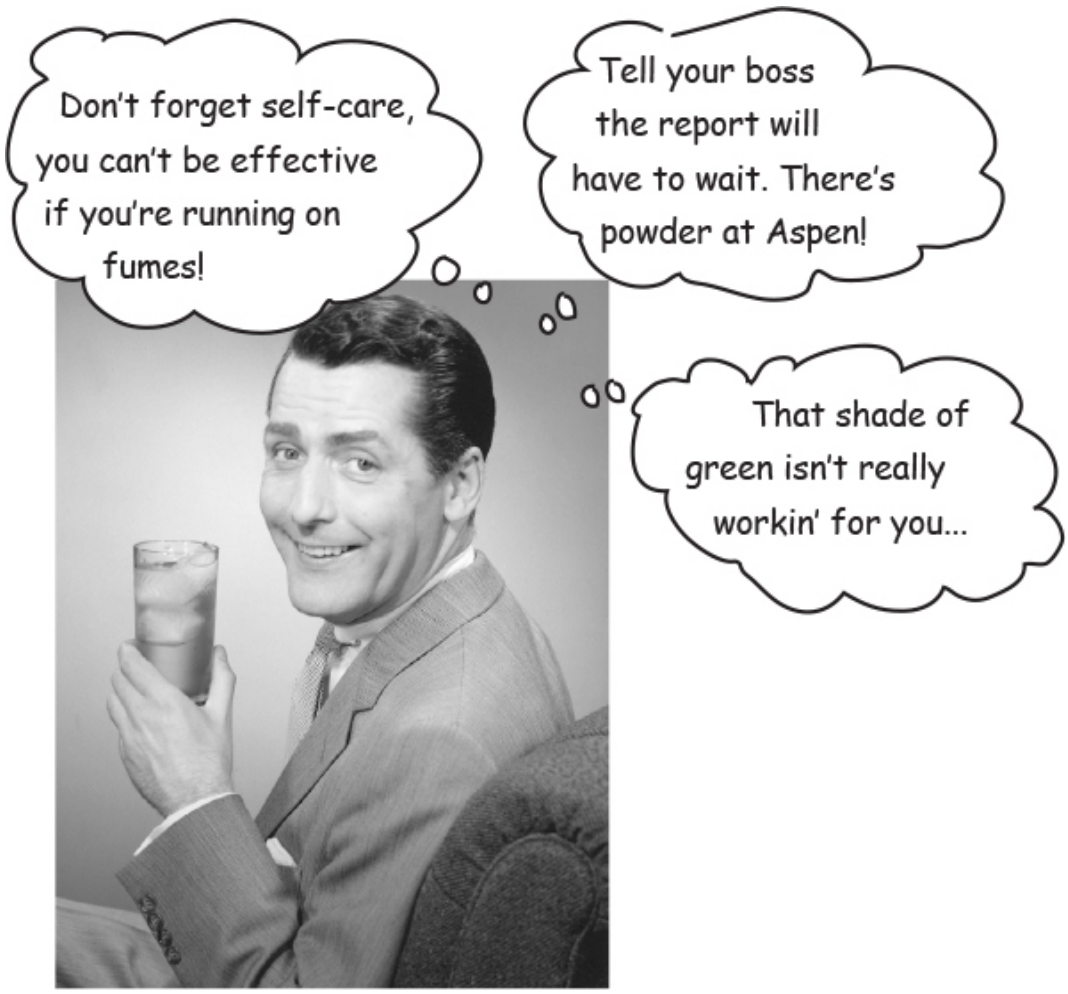
Channels can support: non-blocking I/O; reading and writing via ByteBuffers; Asynchronous I/O. We're not going to show you any of this! But at least now you have some keywords to put into your search engine when you want to know more.

## 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.




**NOTE**

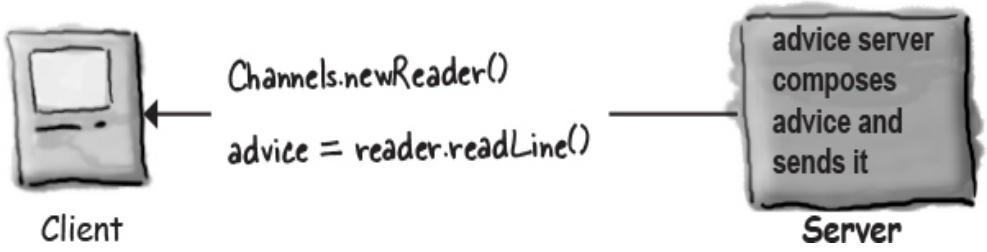
**The Advice Guy**

1. **Connect**

   Client connects to the server


2. ❷ **Read**

   Client gets a Reader for the Channel, and reads a message from the server




## DailyAdviceClient code

This program makes a SocketChannel, makes a BufferedReader (with the help of the channel's Reader), and reads a single line from the server application (whatever is running at port 5000).

```
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.Channels;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;

public class DailyAdviceClient {
  public void go() {
    InetSocketAddress serverAddress = new InetSocketAddress("127.0.0.1", 5000);
    try (SocketChannel socketChannel = SocketChannel.open(serverAddress)) {
      Reader channelReader = Channels.newReader(socketChannel, StandardCharsets.UTF_8);
      BufferedReader reader = new BufferedReader(channelReader);

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

      reader.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

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

Define the server address as being port 5000, on the same host this code is running on (the 'localhost')

This uses try-with-resources to automatically close the SocketChannel when the code is complete

Create a SocketChannel by opening one for the server's address.

Create a Reader that reads from the SocketChannel

Chain a BufferedReader to the Reader from the SocketChannel

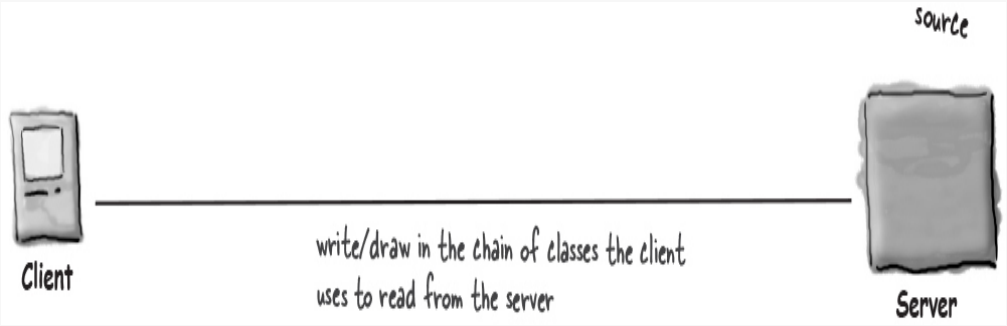
This readLine() is EXACTLY the same as if you were using a BufferedReader chained to a FILE.. In other words, by the time you call a BufferedReader method, the reader doesn't know or care where the characters came from.

This closes the channelReader and this BufferedReader

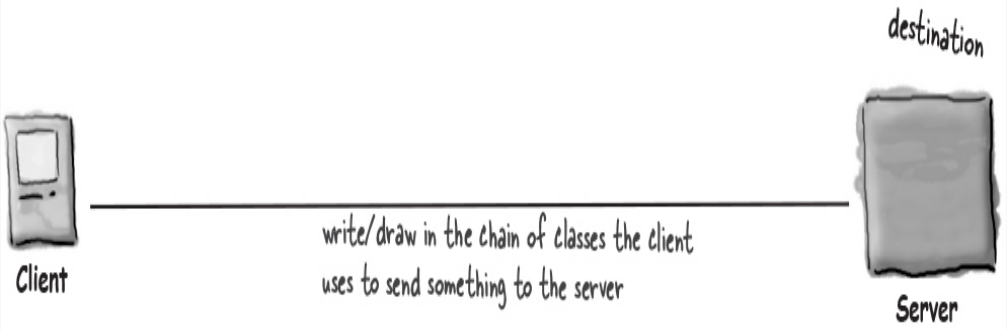
## SHARPEN YOUR PENCIL

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

To *read* text from a SocketChannel:




To *send* text to a SocketChannel:



### SHARPEN YOUR PENCIL

**Fill in the blanks:**

What two pieces of information does the client need in order to make a 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 application

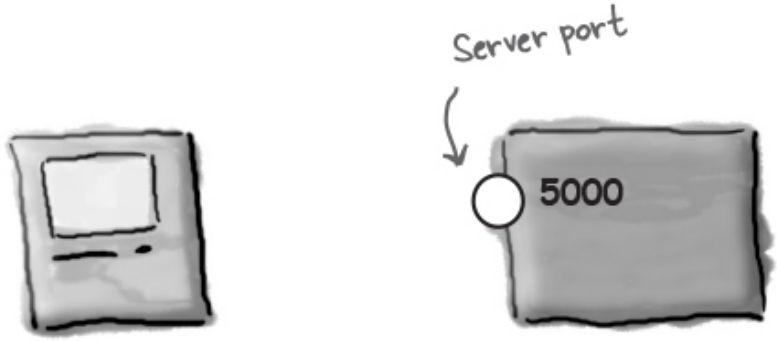
So what's it take to write a server application? Just a couple of Channels. Yes, a couple as in two. A ServerSocketChannel, which waits for client requests (when a client connects) and a SocketChannel to use for communication with the client. If there's more than one client, we'll need more than one channel, but we'll get to that later.

**How it Works:**

1. ❶ Server application makes a ServerSocketChannel, and binds it to a specific port

```
ServerSocketChannel serverChannel =
ServerSocketChannel.open();
serverChannel.bind(new InetSocketAddress(5000));
```

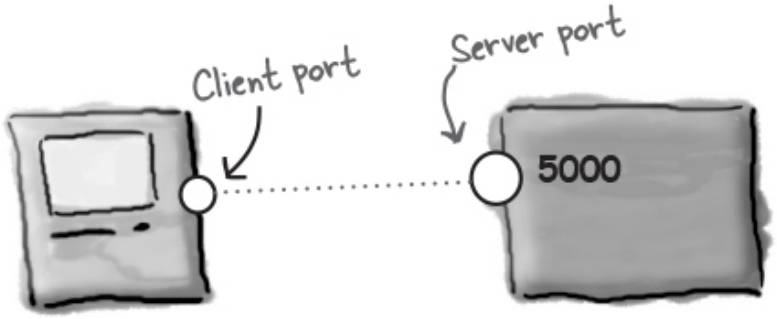
This starts the server application listening for client requests coming in for port 5000.




2. ❷ Client makes a SocketChannel connected to the server application

```
SocketChannel svr = SocketChannel.open(new
InetSocketAddress("190.165.1.103", 5000));
```

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

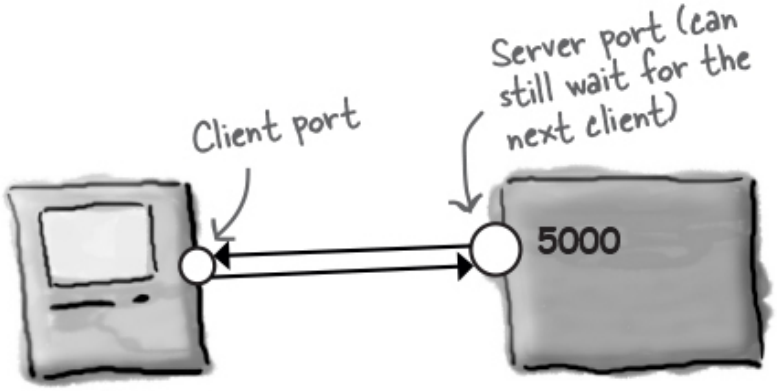



3. ❸ Server makes a new SocketChannel to communicate with this client

```
SocketChannel clientChannel = serverChannel.accept();
```

The accept() method blocks (just sits there) while it's waiting for a client connection. When a client finally connects, the method returns a SocketChannel that knows how to communicate with this client.

The ServerSocketChannel can go back to waiting for other clients. The server has just one ServerSocketChannel, and a SocketChannel per client




## DailyAdviceServer code

This program makes a ServerSocketChannel and waits for client requests. When it gets a client request (i.e. client created a new SocketChannel to this server), the server makes a new SocketChannel to that client. The server makes a PrintWriter (using a Writer created from the SocketChannel) and sends a message to the client.

```
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.util.Random;

public class DailyAdviceServer {
  final private 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."};
  private final Random random = new Random();

  public void go() {

    try (ServerSocketChannel serverChannel = ServerSocketChannel.open()) {
      serverChannel.bind(new InetSocketAddress(5000));

      while (serverChannel.isOpen()) {
        SocketChannel clientChannel = serverChannel.accept();
        PrintWriter writer = new PrintWriter(Channels.newOutputStream(clientChannel));

        String advice = getAdvice();
        writer.println(advice);
        writer.close();
        System.out.println(advice);
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

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

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

Remember the imports

Daily advice comes from this array

ServerSocketChannel makes this server application 'listen' for client requests on the port it's bound to

You have to bind the ServerSocketChannel to the port you want to run the application 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 SocketChannel for communicating with the client

Create an output stream for the client's channel, and wrap it in a PrintWriter. You can use newOutputStream or newWriter here.

Send the client a String advice message.

Close the writer, which will also close the client SocketChannel

Print in the server console, so we can see what's happening.

**BRAIN BARBELL**

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

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


Yes, that's right, **the server can't accept a request from a client until it has finished with the current client**. At which point, it starts the next iteration of the infinite loop, sitting, waiting, at the accept() call until a new request comes in, at which point it makes a SocketChannel to send data to the new client, and starts the process over again.

To get this to work with multiple clients *at the same time*, we need to use separate threads.

We’d give each new client’s SocketChannel to a new thread, and each thread can work independently.

We’re just about to learn how to do that!

## BULLET POINTS

- Client and server applications communicate using Channels.
- A Channel 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 host 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 server applications to run on the same machine, clients connect to a specific application using its port number.
- 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 opening a SocketChannel

```
SocketChannel.open(
 new InetSocketAddress("127.0.0.1", 4200))
```

- Once connected, a client can create readers (to read data from the server) and writers (to send data to the server) for the channel.

```
Reader reader = Channels.newReader(sockCh,
 StandardCharsets.UTF_8);

Writer writer = Channels.newWriter(sockCh,
 StandardCharsets.UTF_8);
```

- To read text data from the server, create a BufferedReader, chained to the Reader. The Reader is a 'bridge' 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 connection.

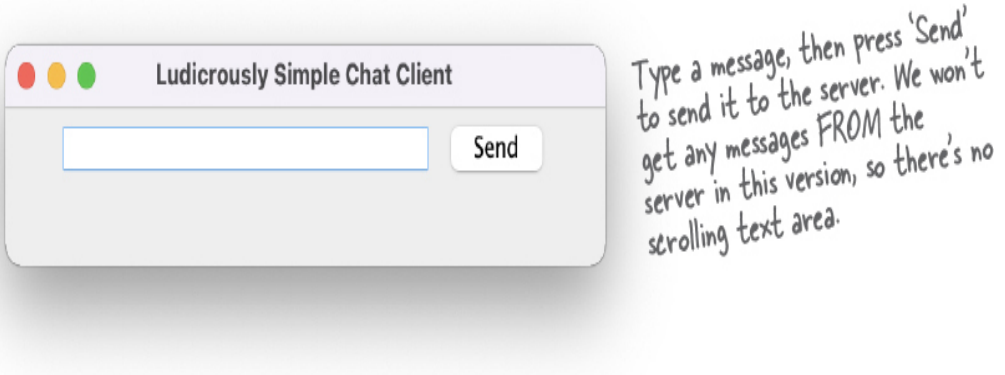
- To write text data to the server, create a PrintWriter chained to the Writer. Call the print() or println() methods to send Strings to the server.
- Servers use a ServerSocketChannel that waits for client requests on a particular port number.
- When a ServerSocketChannel gets a request, it 'accepts' the request by making a SocketChannel for 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**




**Code outline**

Here's an outline of the main functionality the chat client needs to provide. The full code is on the next page.

```
public class SimpleChatClientA {
  private JTextField outgoing;
  private PrintWriter writer;

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

  private void setUpNetworking() {
    // open a SocketChannel to the server
    // make a PrintWriter and assign to writer instance variable
  }

  private void sendMessage() {
    // get the text from the text field and
    // send it to the server using the writer (a PrintWriter)
  }
}
```

```
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import static java.nio.charset.StandardCharsets.UTF_8;

public class SimpleChatClientA {
  private JTextField outgoing;
  private PrintWriter writer;

  public void go() {
    setUpNetworking();

    outgoing = new JTextField(20);

    JButton sendButton = new JButton("Send");
    sendButton.addActionListener(e -> sendMessage());

    JPanel mainPanel = new JPanel();
    mainPanel.add(outgoing);
    mainPanel.add(sendButton);
    JFrame frame = new JFrame("Ludicrously Simple Chat Client");
    frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
    frame.setSize(400, 100);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
  }

  private void setUpNetworking() {
    try {
      InetSocketAddress serverAddress = new InetSocketAddress("127.0.0.1", 5000);

      SocketChannel socketChannel = SocketChannel.open(serverAddress);
      writer = new PrintWriter(Channels.newWriter(socketChannel, UTF_8));
      System.out.println("Networking established.");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  private void sendMessage() {
    writer.println(outgoing.getText());
    writer.flush();
    outgoing.setText("");
    outgoing.requestFocus();
  }

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

Imports for writing (java.io), network connections (java.nio.channels) and the GUI stuff (awt and swing)

This is a static import, we looked at static imports in Chapter 10

Call the method that will connect to the server

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

Open a SocketChannel that connects to the server.

This is where we make the PrintWriter from a writer that writes to the SocketChannel

Now we actually do the writing. Remember, the writer is chained to the writer from the SocketChannel, 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 on the next page.**

**First, start the server in one terminal. Next, use another terminal to start this client.**

## Ready-bake Code

## The really really simple Chat Server

You can use this server code for all 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 to really sharpen your skills after you've finished this book, come back and make this server code more robust.

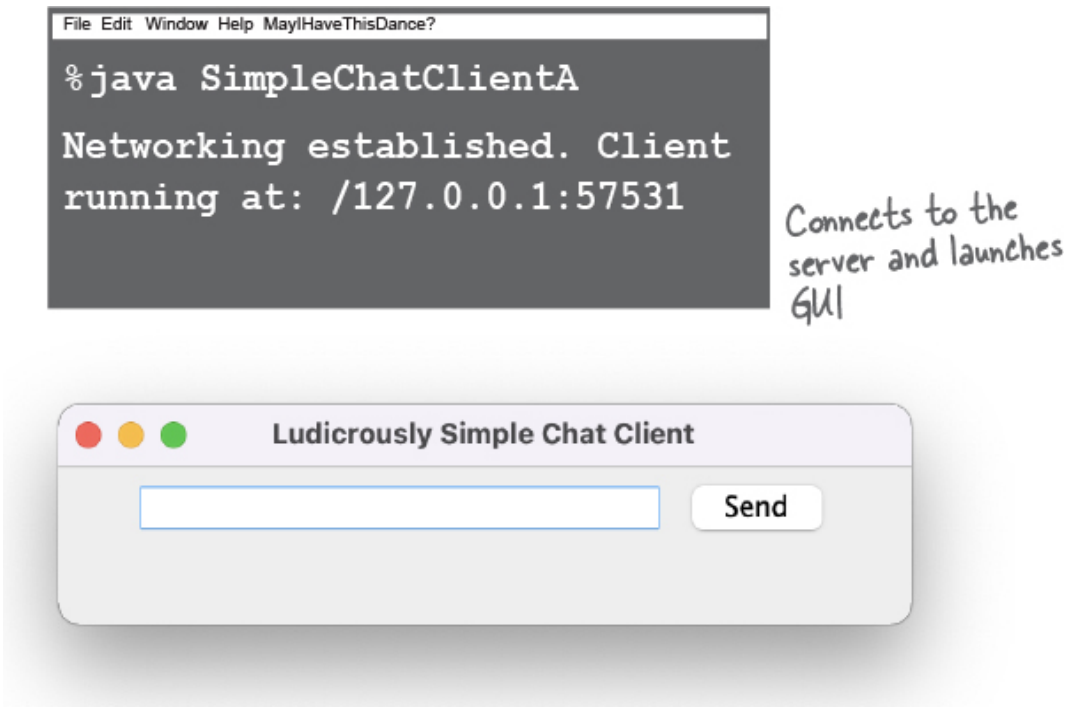
After you finish this chapter, you should be able to annotate this code yourself. You'll understand it much better if *you* work out what's happening than if we explained it to you. Then again, this is Ready-bake code, so you really don't have to understand it at all. It's here just to support the two 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**

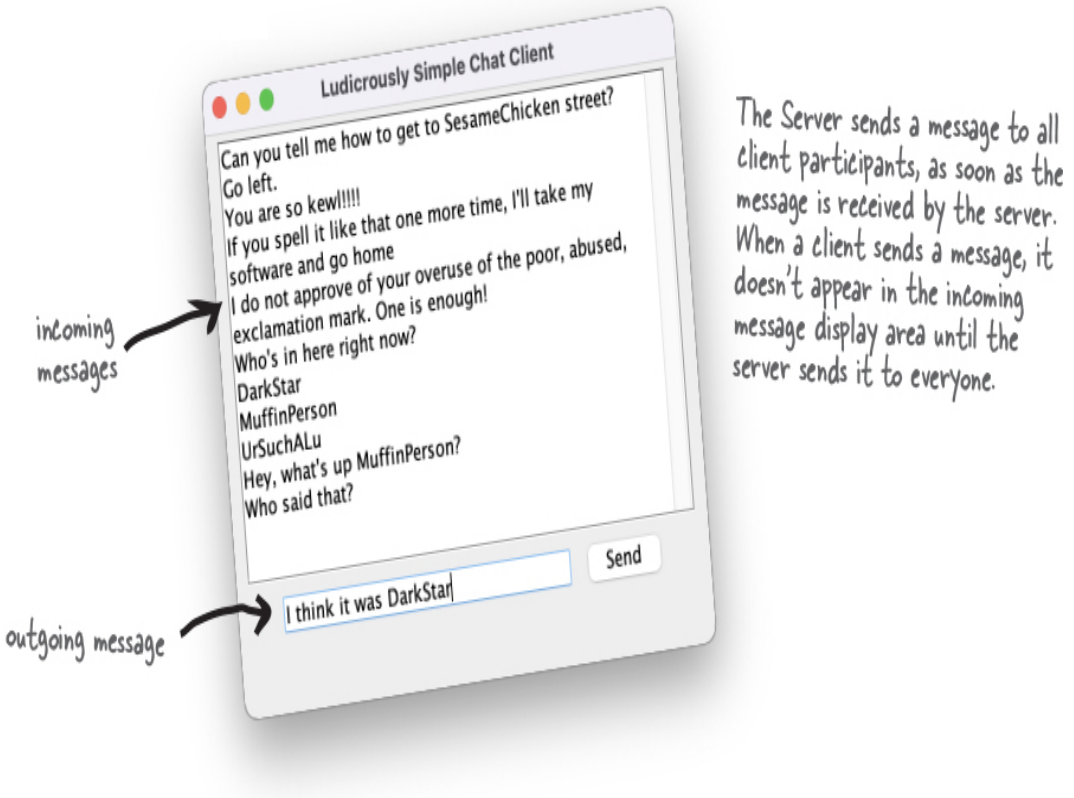
```
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.util.*;
import java.util.concurrent.*;

import static java.nio.charset.StandardCharsets.UTF_8;

public class SimpleChatServer {
  private final List<PrintWriter> clientWriters = new ArrayList<>
();

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

  public void go() {
    ExecutorService threadPool = Executors.newCachedThreadPool();
    try {
      ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();
     serverSocketChannel.bind(new InetSocketAddress(5000));

     while (serverSocketChannel.isOpen()) {
       SocketChannel clientSocket = serverSocketChannel.accept();
       PrintWriter writer = new
PrintWriter(Channels.newWriter(clientSocket, UTF_8));
       clientWriters.add(writer);
       threadPool.submit(new ClientHandler(clientSocket));
       System.out.println("got a connection");
     }
    } catch (IOException ex) {
     ex.printStackTrace();
    }
  }
  private void tellEveryone(String message) {
```

```
    for (PrintWriter writer : clientWriters) {
      writer.println(message);
      writer.flush();
    }
  }

  public class ClientHandler implements Runnable {
    BufferedReader reader;
    SocketChannel socket;

    public ClientHandler(SocketChannel clientSocket) {
      socket = clientSocket;
      reader = new BufferedReader(Channels.newReader(socket,
UTF_8));
    }

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


Runs in the background




**Version Two: send and receive**




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

Should be easy; when you set up the networking make a Reader as well. Then read messages using **`readLine`**.

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

Think about that. What are the options?

1. ① **Option One: 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?

2. ② **Option Two: Poll the server every 20 seconds**

**Pros:** It's do-able, and it fixes the lurker problem.

**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.

3. ③ **Option Three: Read messages as soon as they're sent from the server**

   **Pros:** Most efficient, best usability

   **Cons:** How 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.

> **NOTE**
>
> **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. Each new Java thread is not actually a separate process running on the OS. But it almost *feels* as though it is.

We're going to take a break from the chat application for a bit while we explore how this works. Then we'll come back and add it to our chat client at the end of the chapter.

**Multithreading in Java**

Java has support for multiple threads 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. In fact, **there's more than one way to run multiple jobs in Java**, not just with 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.)

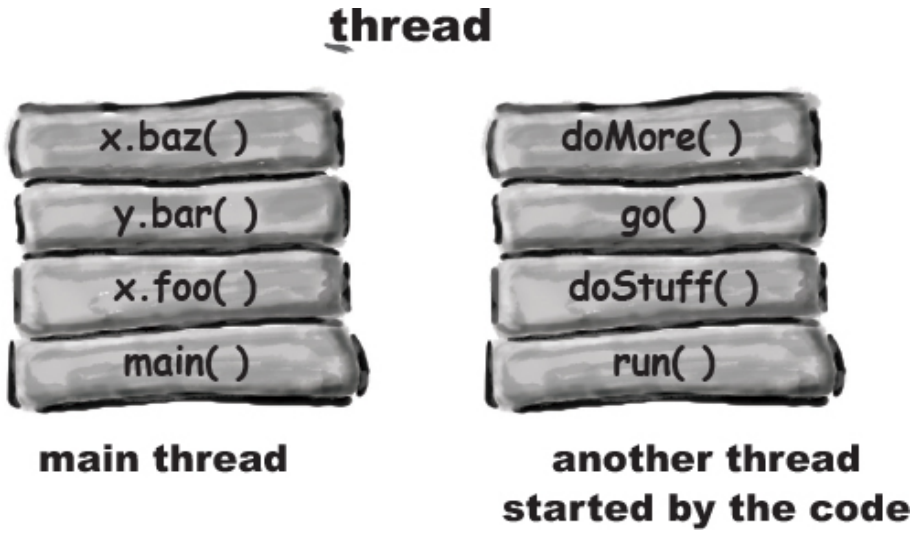
## Java has multiple threads but only one Thread class

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*. In older versions of Java, you always had to create an instance of class **Thread** each time you wanted to start up a new *thread* of execution. Java has evolved over time and now using the **Thread** class directly is not the only way. We'll see this in more detail as we go through the rest of the chapter.

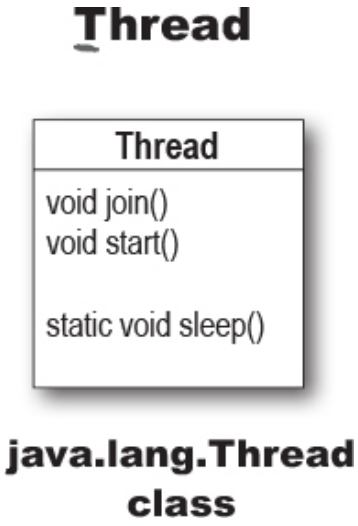
**A thread is a separate 'thread of execution', a separate call stack.**

**A Thread is a Java class that represents a thread.**

**Using the Thread class is not the only way to do multithreading in Java**




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`** (capital ‘T’) is a class that represents a thread of execution. It has methods for starting a thread, joining one thread with another, putting a thread to sleep, and more.

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

With more than one call stack, you can have multiple things happen at the same time. If you're running on a multiprocessor system (like most modern computers and phones), you can actually do more than one thing at a time. With Java threads, even if you're not running on a multiprocessor system, or if you're running more processes than available cores, it can *appear* that you're doing all these 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:

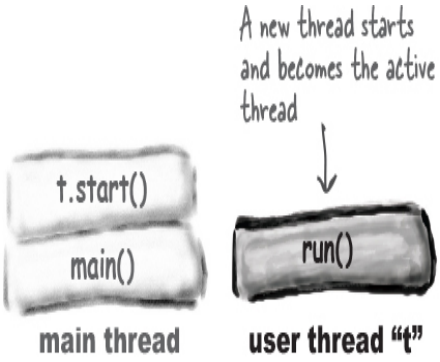
1. **The JVM calls the main() method.**

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

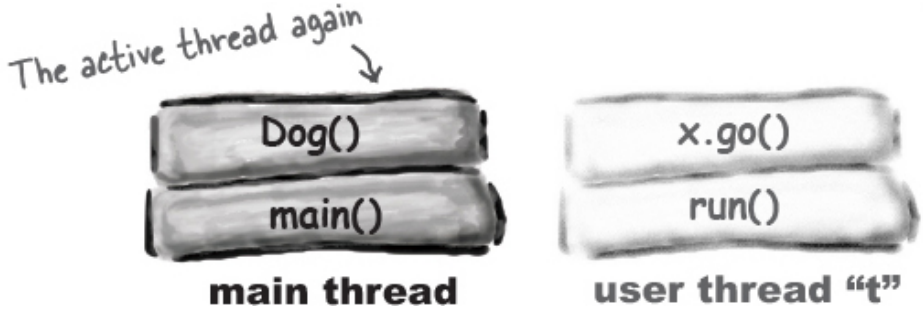
2. ❷ **main() starts a new thread. The main thread may be temporarily frozen while the new thread starts running.**

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




## To create a new call stack you need a job to run

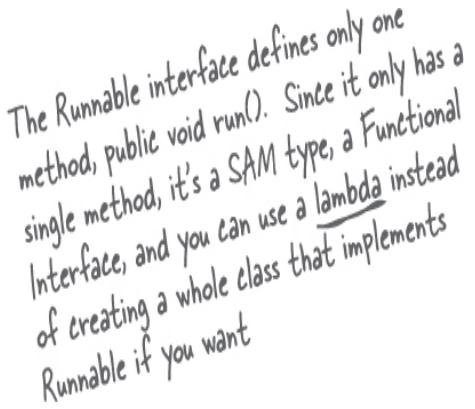

**NOTE**

**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 call stack: run().**

To start a new call stack the thread needs a job - a job the thread will run when it's 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:

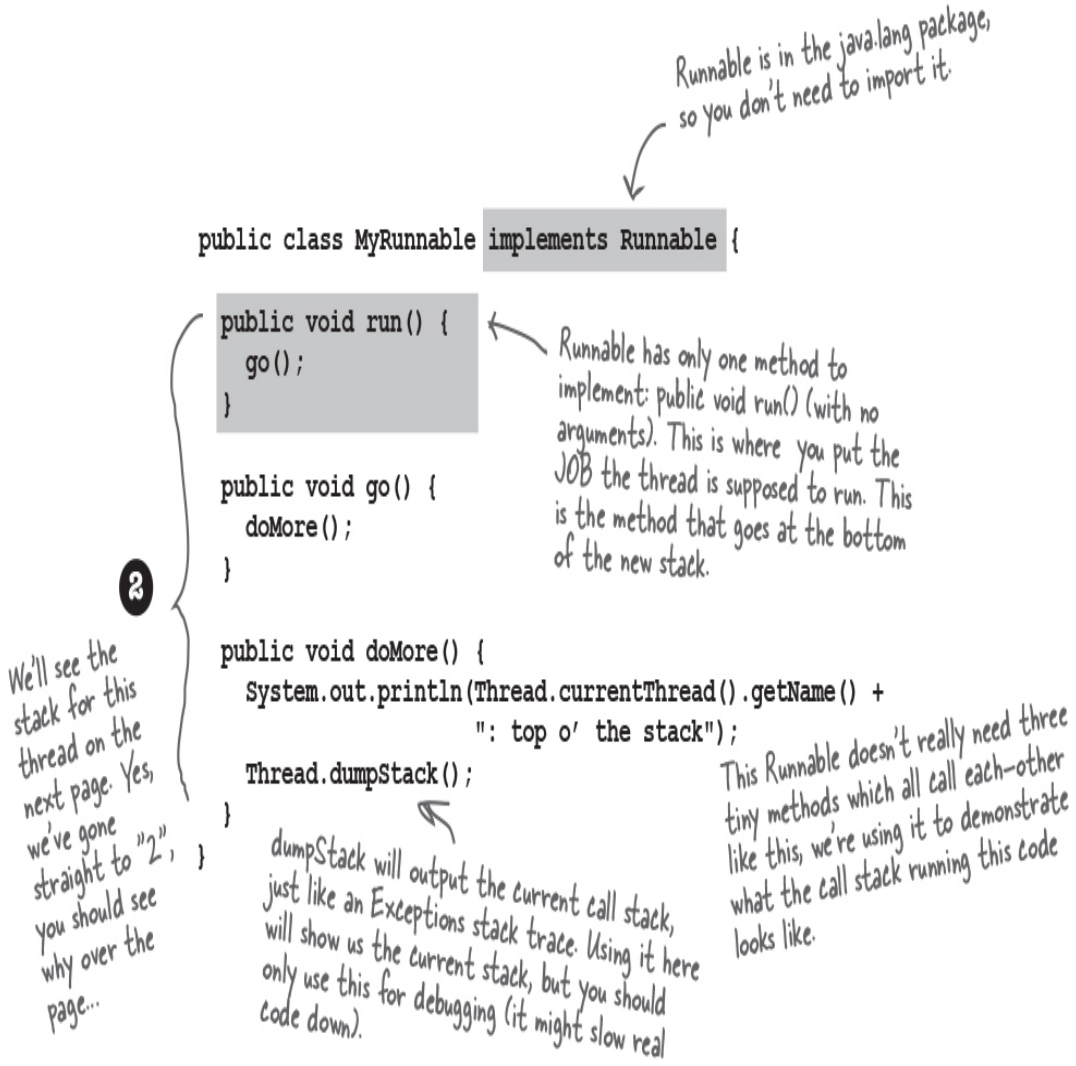
```
public void run() {
  // code that will be run by the new thread
}
```




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, or a lambda expression that is the right shape for the run method.

Once you have a Runnable class or lambda expression, you can tell the JVM to run this code in a separate thread - you're giving the thread its job.

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




## How NOT to run the Runnable

It may be tempting to create a new instance of the Runnable and call the run method, but that's **not enough to create a new call stack**.

```
class RunTester {
  public static void main(String[] args) {
    MyRunnable runnable = new MyRunnable();
    runnable.run();
    System.out.println(Thread.currentThread().getName() +
```

```
                    ": back in main");
    Thread.dumpStack();
  }
}
```

**NOTE**

**This will NOT do what we want!**


***The run() method was called directly from inside the main() method, so it's part of the call stack of the main thread***

## How we used to launch a new thread

The simplest way to launch a new thread is with the Thread class that we mentioned earlier. This method has been around in Java since the very beginning, but **it is no longer the recommended approach to use**. We're showing it here because a) it's simple b) you'll see it in the Real World. We will talk later about why it might not be the best approach.

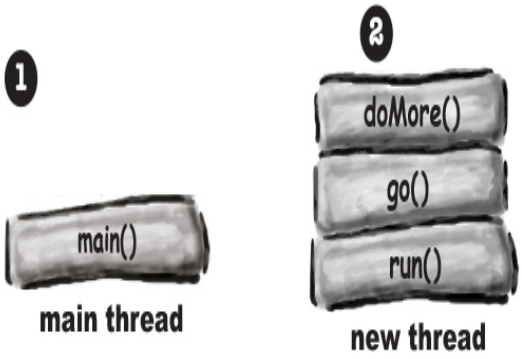
```
class ThreadTester {

  public static void main(String[] args) {

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

    myThread.start();

    System.out.println(
      Thread.currentThread().getName() +
        ": back in main");
    Thread.dumpStack();
  }
}
```

Pass the new Runnable instance into the new Thread constructor. This tells the thread what job to run. In other words, the Runnable's run() method will be the first method that the new thread will run.

1

This thread's stack is below

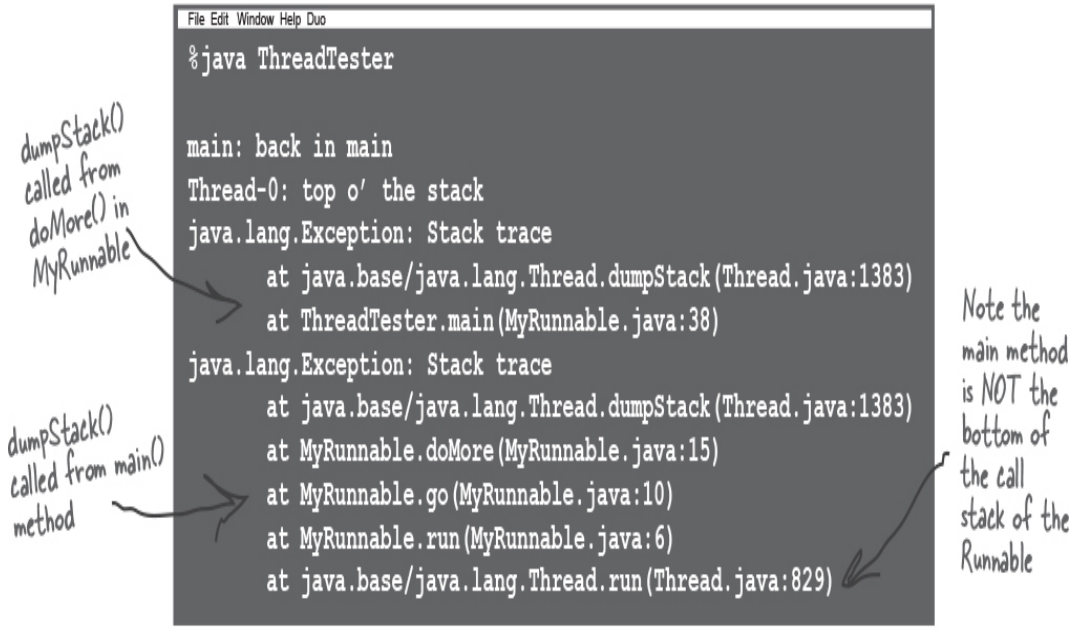
You won't get a new thread of execution until you call start() on the Thread instance. A thread is not really a thread until you start it. Before that, it's just a Thread instance, like any other object, but it won't have any real 'threadness'.




Now we have two independent call stacks

Call stack for the "main" thread of the application (started by the "public static void main" method).

Call stack for the new thread we started with the MyRunnable job.




## A better alternative: don't manage the Threads at all

Creating and starting a new Thread gives you a lot of control over that Thread, but the downside is you *have* to control it. You have to keep track of all the Threads and make sure they're shut down at the end. Wouldn't it be better to have something else that starts, stops, and even reuses the Threads, so you don't have to?

Allow us to introduce an interface in java.util.concurrent, **`ExecutorService`**. Implementations of this interface will *execute* jobs (Runnables). Behind the scenes the ExecutorService will create, reuse and kill threads in order to run these jobs.

The **`java.util.concurrent.Executors`** class has *factory methods* to create the ExecutorService instances we'll need.

Executors have been around since Java 5 and so should be available to you even if you're working with quite an old version of Java. There's no real need to use Thread directly at all these days.

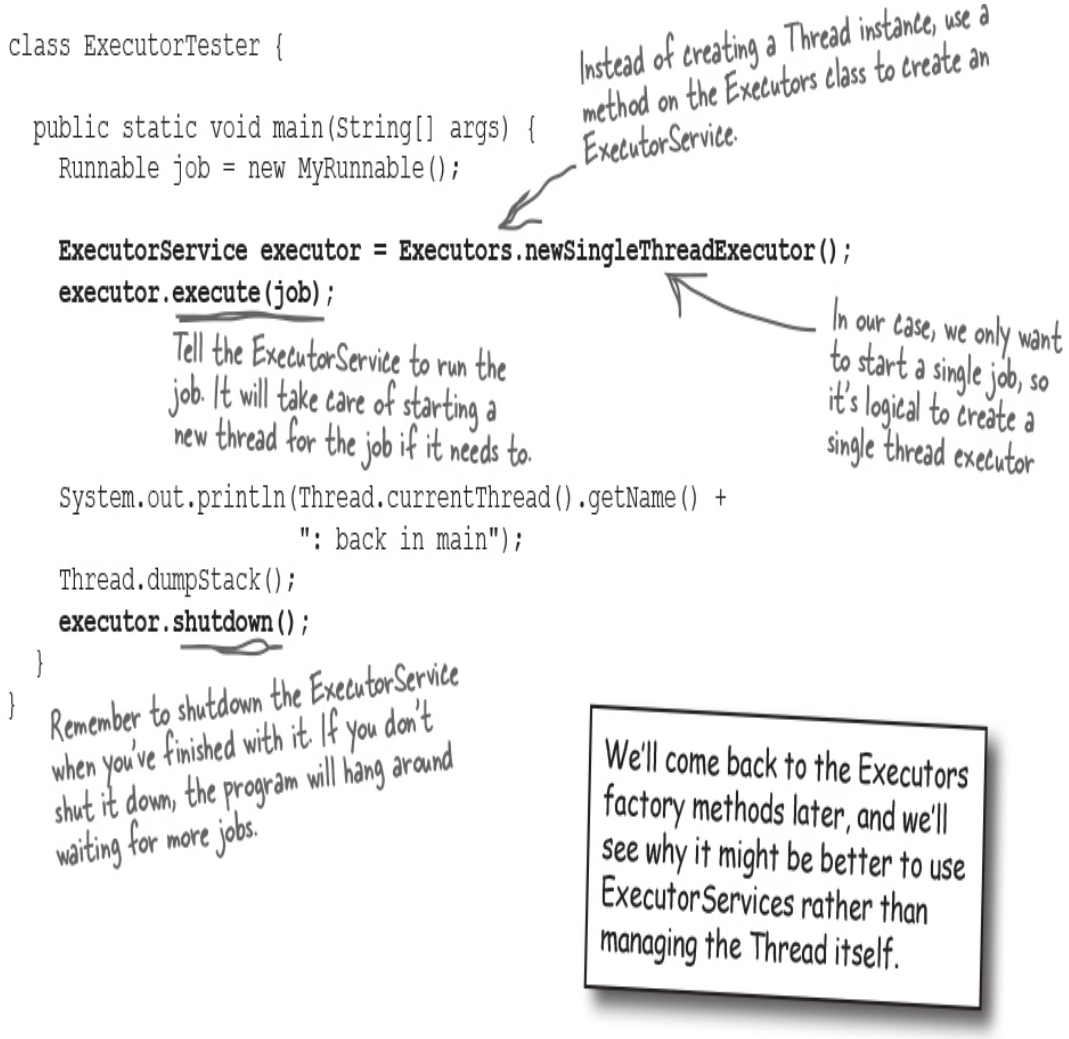
NOTE

**Static factory methods can be used instead of constructors.**

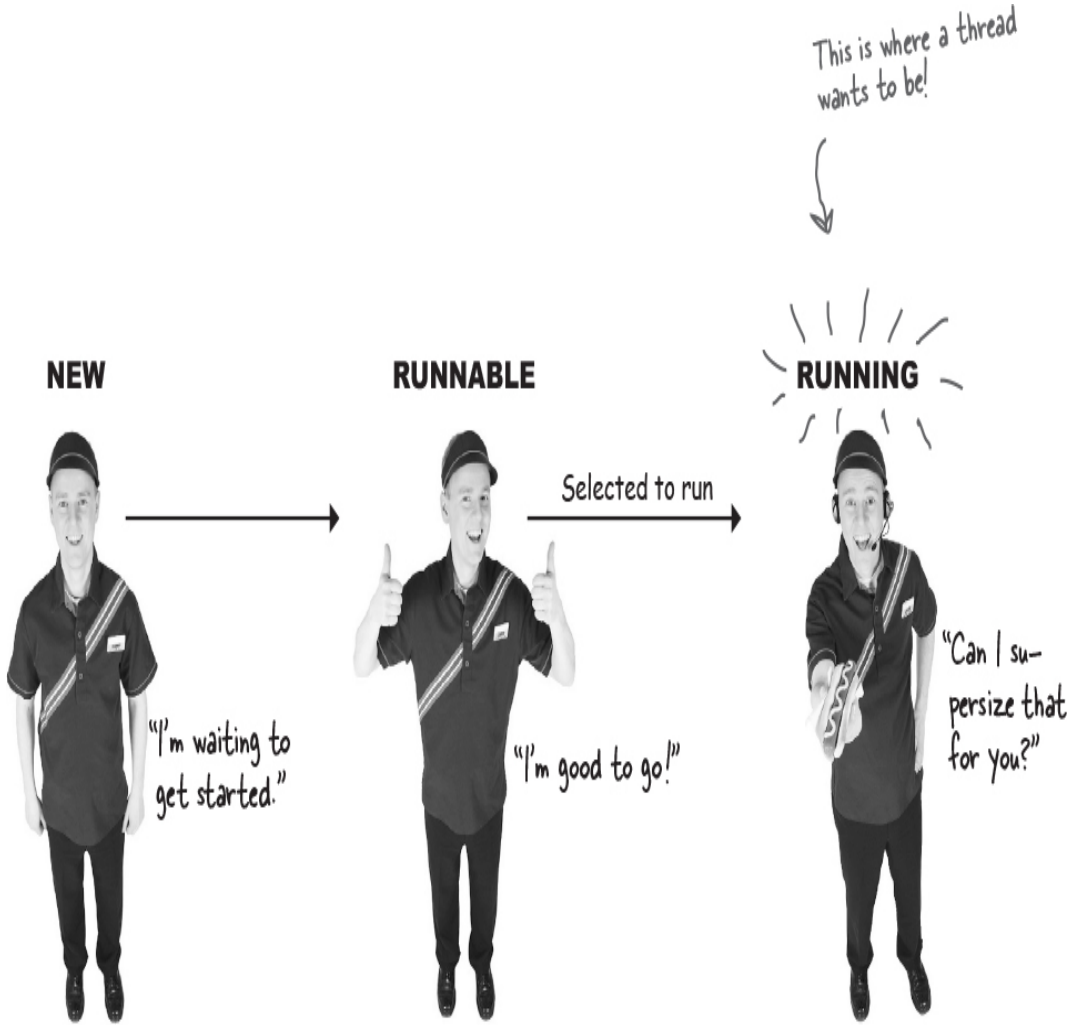
**Factory methods return exactly the implementation of an interface that we need. We don't need to know the concrete classes or how to create them.**

## Running one job

For the simple cases we're going to get started with, we'll only want to run one job in addition to our main class. There's a *single thread executor* which we can use to do this.




## The three states of a new thread




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

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 Up And Running. 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.***

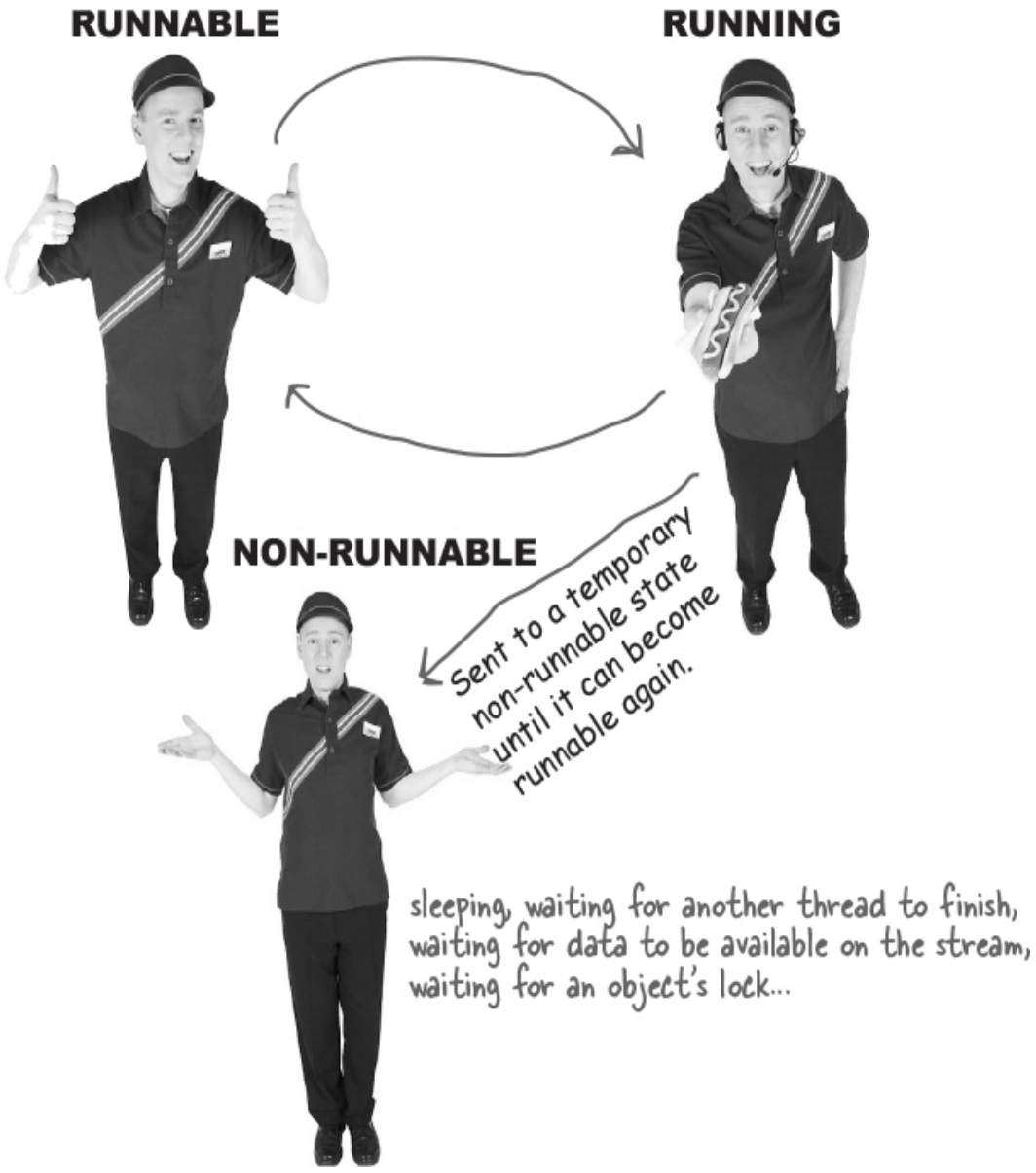
## Typical runnable/running loop

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 an 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 it 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 multithreaded 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 multithreaded 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."


Number four, you've had enough time. Back to runnable. Number two, looks like you're up!
Oh, now it looks like you're gonna have to sleep. Number five, come take his place. Number two, you're still sleeping...

> **NOTE**
>
> **The thread scheduler makes all the decisions about who runs and who doesn't. It usually makes the threads take turns, nicely. But there's no guarantee about that. It 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:

```
class ExecutorTestDrive {
  public static void main (String[] args) {
    ExecutorService executor =
      Executors.newSingleThreadExecutor();

    executor.execute(() ->
      System.out.println("top o' the stack"));

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

Runnable is a Functional Interface and can be represented as a lambda expression. Our job is just a single line of code, so a lambda expression makes sense here

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

It doesn't matter if you run this using an ExecutorService, like the code above, or with Threads directly, like the code below, both show the same symptoms.

```
class ThreadTestDrive {
  public static void main (String[] args) {
    Thread myThread = new Thread(() ->
      System.out.println("top o' the stack"));
    myThread.start();
    System.out.println("back in main");
  }
}
```

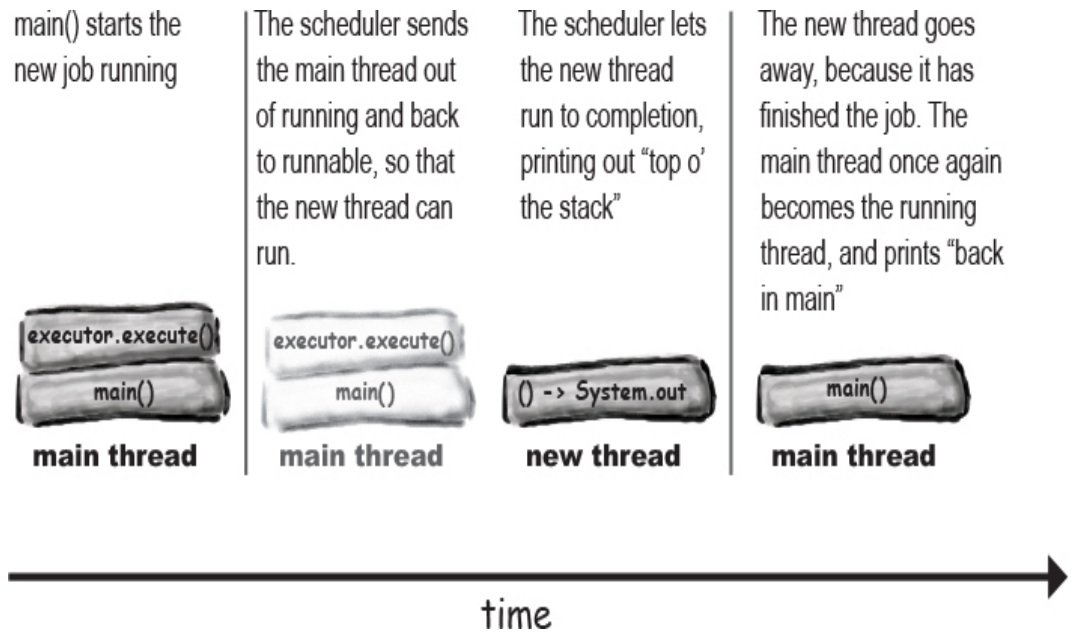
## Produced this output:

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

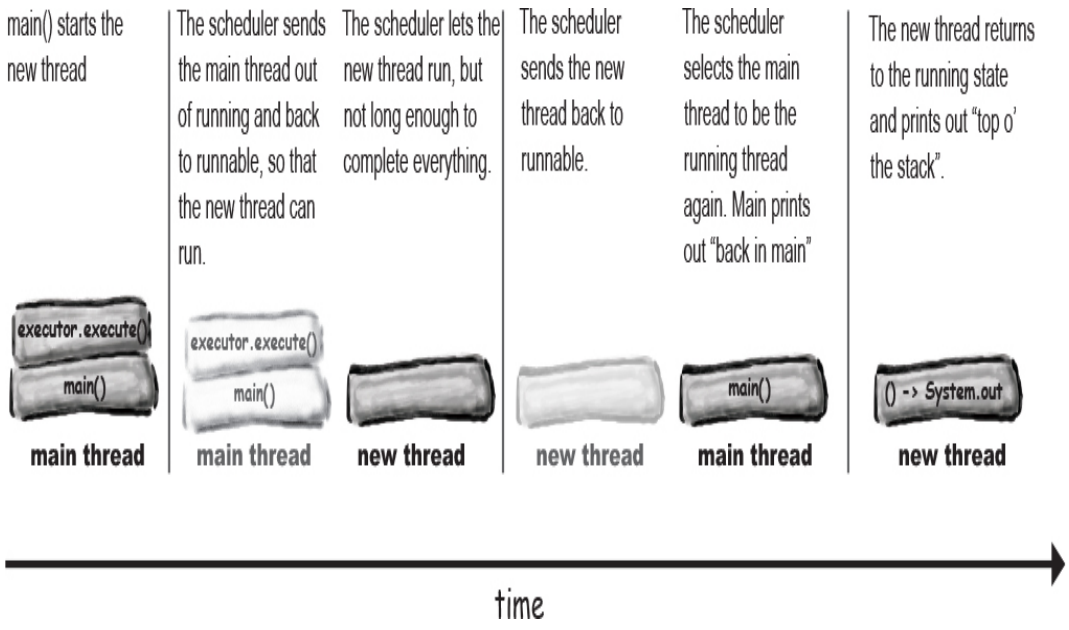
## How did we end up with different results?

Multi-threaded programs are *not deterministic*, they don't run the same way every time. The thread scheduler can schedule each thread differently each time the program runs.

**Sometimes it runs like this:**




**And sometimes it runs like this:**




## NOTE

**Even if the new thread is tiny, if it only has one line of code to run like our lambda expression, it can still be interrupted by the thread scheduler**

## THERE ARE NO DUMB QUESTIONS

**Q: Should I use a lambda expression for my Runnable, or create a new class that implements Runnable?**

**A:** It depends upon how complicated your job is, and also on whether you think it's easier to understand as a lambda expression or a class. Lambda expressions are great for when the job is really tiny, like our single-line "print" example. Lambda expressions (or method references) may also work if you have a few lines of code in another method that you want to turn into a job:

```
executor.execute(() -> printMsg());
```

You'll most likely want to use a full Runnable class if your job needs to store things in fields, and/or if your job is made up of a number of methods. This is more likely when your jobs are more complex.

**Q: What's the advantage of using an ExecutorService? So far, it works the same as creating a Thread and starting it.**

**A:** It's true that for these simple examples, where we're starting just one thread, letting it run, and then stopping our application, the two approaches seem similar. ExecutorServices become really helpful when we're starting lots of independent jobs. We don't necessarily want to create a new Thread for each of these jobs, and we don't want to keep track of all these Threads. There are different ExecutorService implementations depending upon how many threads we'll want to start (or especially if we don't know how many Threads we'll need), including ExecutorServices that create Thread pools. Thread pools let us reuse Thread instances, so we don't have to pay the cost of starting up new Threads for every job. We'll explore this in more detail later.

## 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. The job can be 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.
- Because the Runnable interface has just a single method, you can use lambda expressions where a Runnable is expected.
- Using the Thread class to run separate jobs is no longer the preferred way to create multithreaded applications in Java. Instead, use an Executor or an ExecutorService.
- The Executors class has helper methods that can create standard ExecutorServices to use to start new jobs.
- A thread is in the NEW state when it has not yet started.
- When a thread has been started, 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 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. We'll see locks in the next chapter.

- Thread scheduling is not guaranteed to work in any particular way, so you cannot be certain that threads will take turns nicely.

## Putting a thread to sleep

One way 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 amount of time you want the thread to sleep for, 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();
}
```

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? Effectively, yes. 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, therefore there are no guarantees about how long the thread will be out of action.

**NOTE**

**Putting a thread to sleep gives the other threads 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.**

It can be hard to understand how much time a number of milliseconds represents. There's a convenience method on java.util.concurrent.TimeUnit that we can use to make a more readable sleep time:

```
TimeUnit.MINUTES.sleep(2);
```

which may be easier to understand than:

```
Thread.sleep(120000);
```

(You still need to wrap both in a try-catch though)

## 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"?

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

File Edit Window Help SnoozeButton

```
% java PredictableSleep
back in main
top o' the stack
% java PredictableSleep
back in main
top o' the stack
% java PredictableSleep
back in main
top o' the stack
% java PredictableSleep
back in main
top o' the stack
% java PredictableSleep
back in main
top o' the stack
```

```
class PredictableSleep {
  public static void main (String[] args) {
    ExecutorService executor =
      Executors.newSingleThreadExecutor();
    executor.execute(() -> sleepThenPrint());
    System.out.println("back in main");
    executor.shutdown();
  }

  private static void sleepThenPrint() {
    try {
      TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("top o' the stack");
  }
}
```

Instead of putting a lambda with an ugly try-catch inside, we've put the job code inside a method. We're calling the method from this lambda expression

Calling sleep here will force the new thread to leave the currently-running state. The main thread will get a chance to print out "back in main".

Thread.sleep() throws a checked Exception that we need to catch or declare. Because catching the Exception makes the job's code a bit longer, we've put it into its own method.

There will be a pause (for about two seconds) before we get to this line, which prints out "top o' the stack"

## BRAIN BARBELL

Can you think of any problems with forcing your threads to sleep for a set amount of time? How long will it take to run this code 10 times?

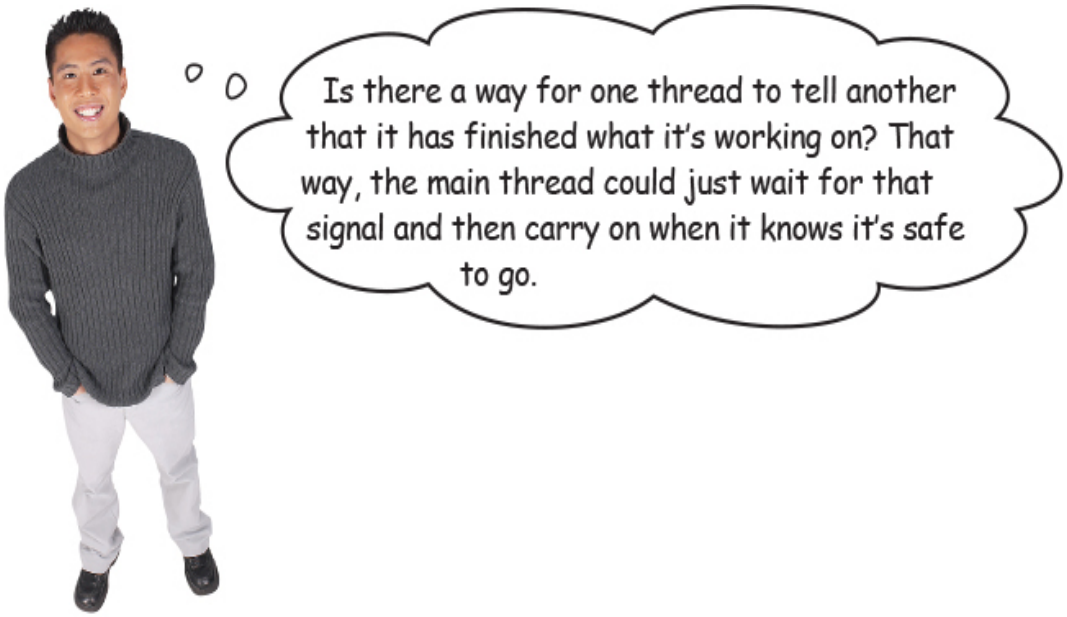
## There are downsides to forcing the thread to sleep

1. ① **The program has to wait for at least that amount of time.**

   If we put the thread to sleep for two seconds, the thread will be non-runnable for that time. When it wakes up, it won't automatically become the currently-running thread. When a thread wakes up, the thread is once again at the mercy of the thread scheduler. Our application is going to be hanging around for at least those two seconds, probably more. This might not sound like a big deal, but imagine a bigger program full of these pauses *intentionally* slowing down the application.

2. ② **How do you know the other job will finish in that time?**

   We put the new thread to sleep for two seconds, assuming that the main thread would be the running thread, and complete its work in that time. But what if the main thread took longer to finish than that? What if another thread, running a longer job, was scheduled instead? One of the ways people deal with this is to set sleep times that are much longer than they'd expect a job to take, but then our first problem becomes even more of a problem.




**A better alternative: wait for the perfect time.**

What we really wanted in our example was to wait until a specific thing had happened in our main thread before carrying on with our new thread. Java supports a number of different mechanisms to do this, like Future, CyclicBarrier, Semaphore, and CountDownLatch.

> **NOTE**
>
> **To coordinate events happening on multiple threads, one thread may need to wait for a specific signal from another thread before it can continue.**

## Counting down until ready

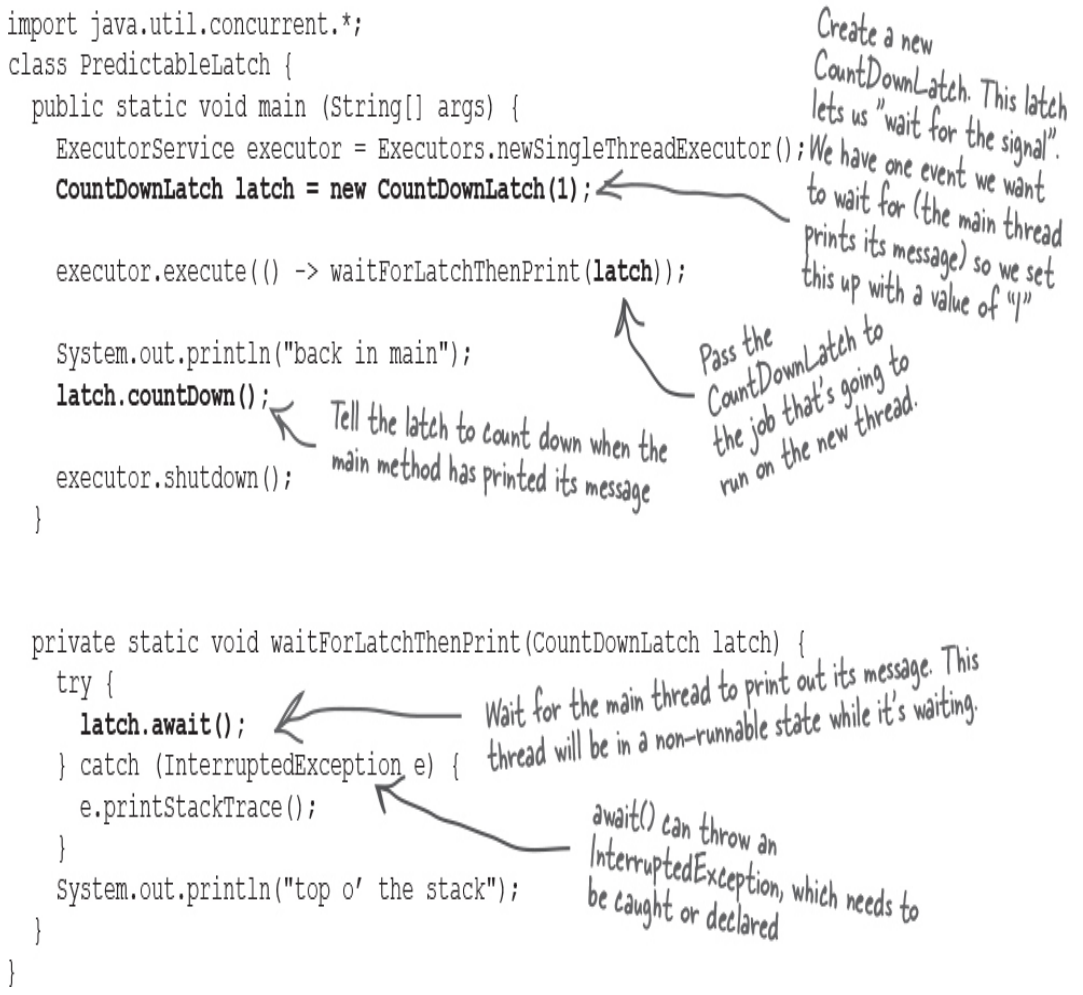
You can make threads *count down* when significant events have happened. A thread (or threads) can wait for all these events to complete before continuing. You might be counting down until a minimum number of clients have connected, or a number of services have been started.

**CountDownLatch is a barrier synchronizer. Barriers are mechanisms to allow threads to coordinate with each other.**

**Other examples are CyclicBarrier and Phaser.**

This is what **`java.util.concurrent.CountDownLatch`** is for. You set a number to count down from. Then any thread can tell the latch to count down when a relevant event has happened.

In our example, we only have one thing we want to count - our new thread should wait until the main thread has printed “back in main” before it can continue.




The code is really similar to the code that performs a sleep, the main difference is the latch.countDown in the main method. The performance difference is significant though. Instead of having to wait *at least* two seconds to make sure main has printed its message, the new thread only waits until the main method has printed its "back in main" message.

To get an idea of the performance difference this might make on a real system, when this latch code was run on a MacBook 100 times, it took around 50 milliseconds to finish *all* one hundred runs, and the output was in the correct order *every time*. If running the sleep() version just one time takes over 2 seconds (2000 milliseconds), imagine how long it took to run 100 times*.....

NOTE

* 200331 milliseconds. That's over 4000x slower

## Making and starting *two* threads (or more!)

What happens if we want to start more than one job in addition to our main thread? Clearly, we can't use Executors.newSingleThreadExecutor() if we want to run more than one thread. What else is available?

## java.util.concurrent.Executors

**ExecutorService newCachedThreadPool()**

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

**ExecutorService newFixedThreadPool(int nThreads)**

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

**ScheduledExecutorService newScheduledThreadPool(int corePoolSize)**

Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

**ExecutorService newSingleThreadExecutor()**

Creates an Executor that uses a single worker thread operating off an unbounded queue.

**ScheduledExecutorService newSingleThreadScheduledExecutor()**

Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically.

**ExecutorService newWorkStealingPool()**

Creates a work-stealing thread pool using the number of available processors as its target parallelism level.

(Just a few of the factory methods)

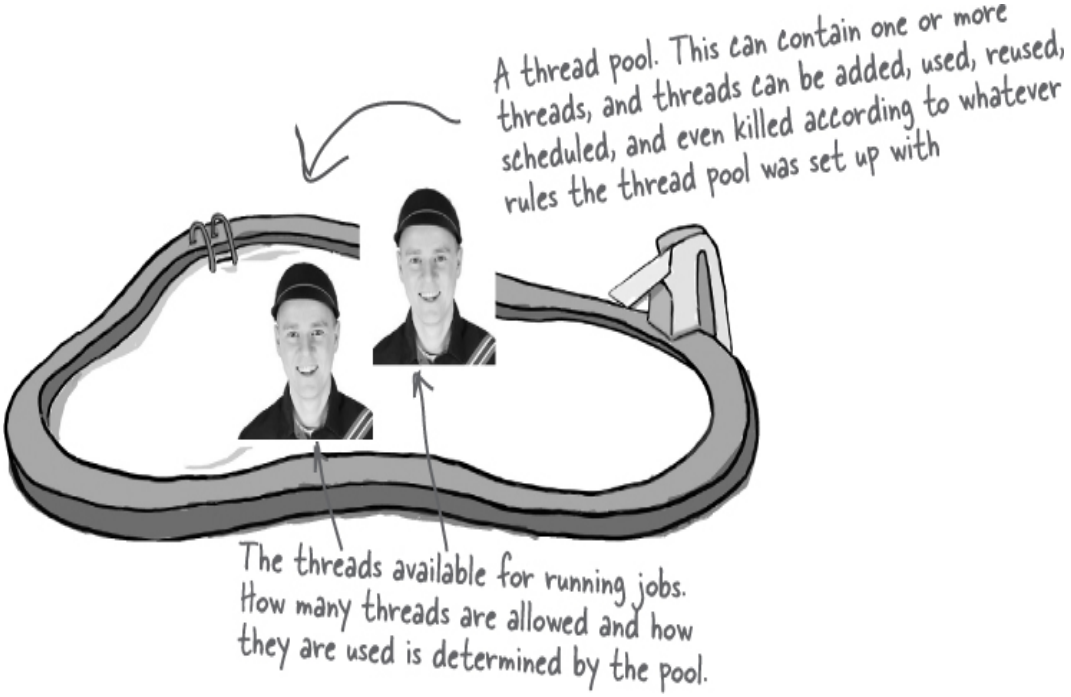
NOTE

**These ExecutorServices use some form of Thread Pool. This is a collection of Thread instances that can be used (and reused) to perform jobs.**

**How many threads are in the pool, and what to do if there are more jobs to run than threads available, depends on the ExecutorService implementation.**

## Pooling Threads

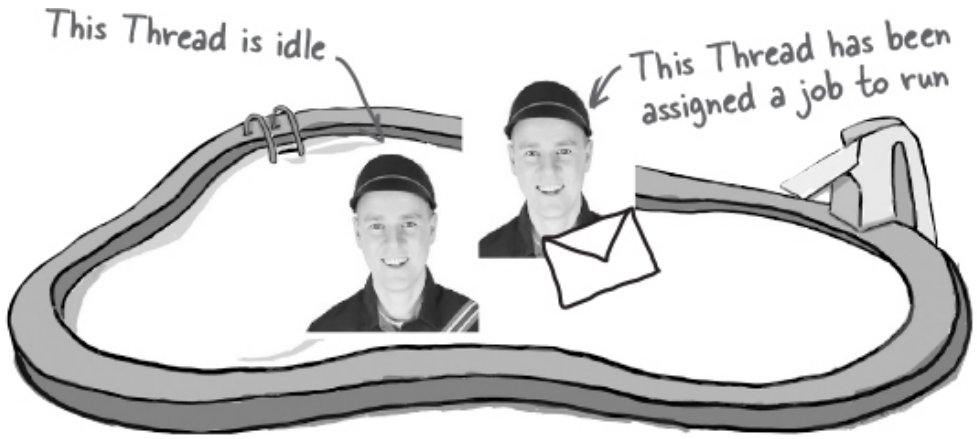
Using a pool of resources, especially ones that are expensive to create like Threads or database connections, is a common pattern in application code.




When you create a new ExecutorService, its pool may be started with some threads to begin with, or the pool may be empty.

You can create an ExecutorService with a thread pool using one of the helper methods from the Executors class:

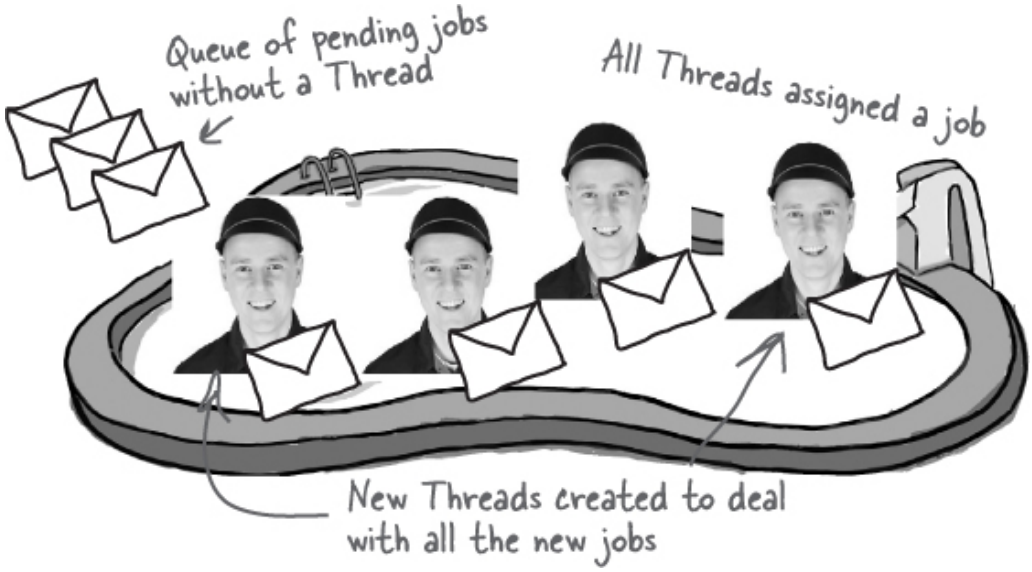
```
ExecutorService threadPool =
   Executors.newCachedThreadPool();
```




You can use the pool's threads to run your job by giving the job to the ExecutorService. The ExecutorService can then figure out if there's a free Thread to run the job.

```
threadPool.execute(() -> run("Job 1"));
```

This means an ExecutorService can **reuse** threads, it doesn't just create and destroy them.



As you give the ExecutorService more jobs to run, it *may* create and start new Threads to handle the jobs. It *may* store the jobs in a queue if there are more jobs than Threads.

How an ExecutorService deals with additional jobs depends on how it is set up

```
threadPool.execute(() -> run("Job 324"));
```

The ExecutorService may also **terminate** Threads that have been idle for some period of time. This can help to minimise the amount of hardware resources (CPU, memory) your application needs.

## Running multiple threads

The following example runs two jobs, and uses a fixed-sized thread pool to create two threads to run the jobs. Each thread has the same job: run in a loop, printing the currently-running thread's name with each iteration.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class RunThreads {

  public static void main(String[] args) {
    ExecutorService threadPool = Executors.newFixedThreadPool(2);
    threadPool.execute(() -> runJob("Job 1"));
    threadPool.execute(() -> runJob("Job 2"));
    threadPool.shutdown();
  }

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

Create an ExecutorService with a fixed-sized thread pool (we know we're only going to run two jobs)

A lambda expression that represents our Runnable job. If you don't want to use lambdas, here you'd pass in a new instance of your Runnable class, like we did when we created MyRunnable earlier in the chapter

The job is to run through this loop, printing the thread's name each time.

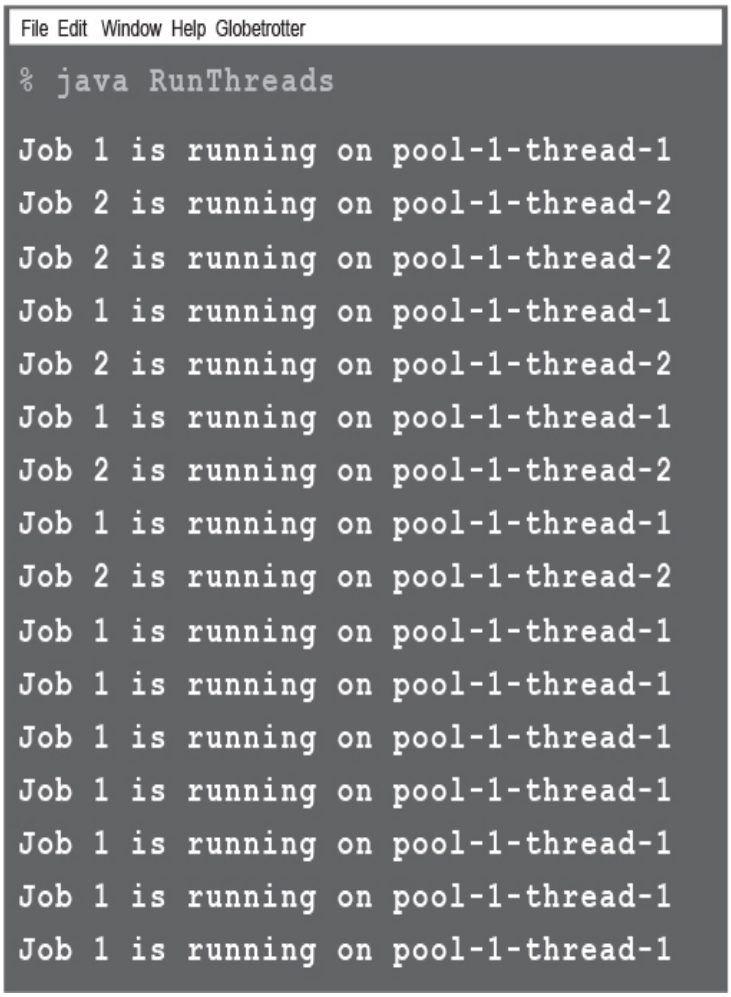
## What will happen?

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 this on a modern multi-core system, the two jobs will likely run in parallel, but there's no guarantee that this means they will complete in the same amount of time, or output values at the same rate.

Part of the output when the loop iterates 25 times.




## Closing time at the thread pool

You may have noticed that our examples have a **`threadPool.shutdown()`** at the end of the main methods. Although the thread pools will take care of our individual Threads, we do need to be responsible adults and close the pool when we're finished with it. That way, the pool can empty its job queue and shut down all of its threads to free up system resources.

ExecutorService has two shutdown methods. You can use either, but to be safe we'd use both:

1. ① **ExecutorService.shutdown()**

   Calling shutdown() asks the ExecutorService nicely if it wouldn't mind awfully wrapping things up so everyone can go home.

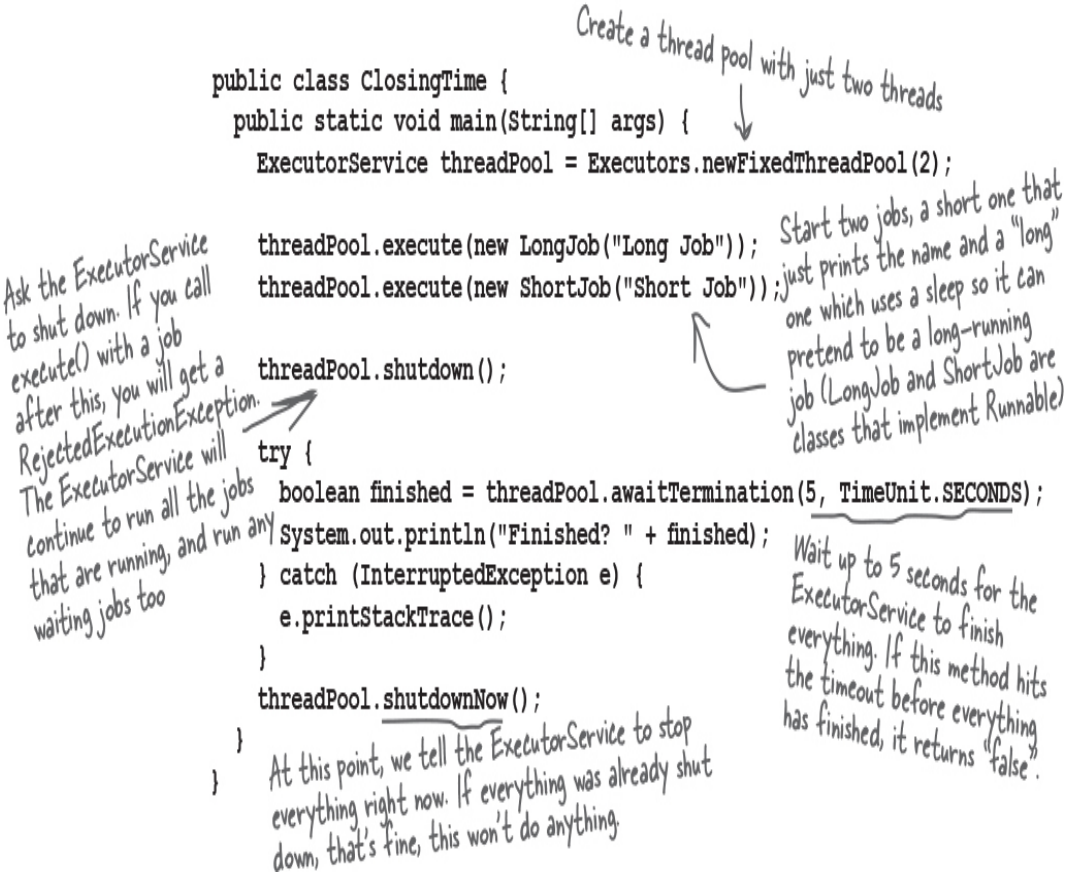
   All of the Threads which are currently running jobs are allowed to finish those jobs, and any jobs waiting in the queue will also be finished off. The ExecutorService will reject any new jobs too.

   If you need your code to wait until all of those things are finished, you can use **`awaitTermination`** to sit and wait until it's finished. You give awaitTermination a maximum amount of time to wait for everything to end, so awaitTermination will hang around until either the ExecutorService has finished everything, or the timeout has been reached, whichever is earlier.

2. ② **ExecutorService.shutdownNow()**

   Everybody out! When this is called, the ExecutorService will try to stop any Threads that are running, will not run any waiting jobs, and definitely won't let anyone else into the pool.

   Use this if you need to put a stop to everything. This is sometimes used after first calling shutdown() to give the jobs a chance to finish before pulling the plug entirely.


Create a thread pool with just two threads
public class ClosingTime {
public static void main(String[] args) {
ExecutorService threadPool = Executors.newFixedThreadPool(2);
threadPool.execute(new LongJob("Long Job"));
threadPool.execute(new ShortJob("Short Job"));
threadPool.shutdown();
try {
boolean finished = threadPool.awaitTermination(5, TimeUnit.SECONDS);
System.out.println("Finished? " + finished);
} catch (InterruptedException e) {
e.printStackTrace();
}
threadPool.shutdownNow();
}
}
Start two jobs, a short one that just prints the name and a "long" one which uses a sleep so it can pretend to be a long-running job (LongJob and ShortJob are classes that implement Runnable)
Ask the ExecutorService to shut down. If you call execute() with a job after this, you will get a RejectedExecutionException. The ExecutorService will continue to run all the jobs that are running, and run any waiting jobs too
Wait up to 5 seconds for the ExecutorService to finish everything. If this method hits the timeout before everything has finished, it returns "false".
At this point, we tell the ExecutorService to stop everything right now. If everything was already shut down, that's fine, this won't do anything.


## Um, yes. There IS a dark side.

### Multithreading 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.

## 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.
- You can also use the sleep method on java.util.concurrent.TimeUnit, for example TimeUnit.SECONDS.sleep(2);
- The sleep() method throws a checked exception (InterruptedException), so all calls to sleep() must be wrapped in a try/ catch, or declared.
- There are different mechanisms to give threads a chance to wait for each other. You can use sleep(), but you can also use CountDownLatch to wait for the right number of events to have happened before continuing.
- Managing threads directly can be a lot of work. Use the factory methods in Executors to create an ExecutorService, and use this service to run Runnable jobs.
- Thread pools can manage creation, reuse, and destruction of threads so you don’t have to.
- ExecutorServices should be shut down correctly so the jobs are finished and threads terminated. Use shutdown() for graceful shutdown, and shutdownNow() to kill everything.

**IT'S A CLIFF-HANGER!**

A dark side? Race conditions?? Data corruption?! But what can we DO about those things? Don't leave us hanging!

Stay tuned for the next chapter, where we address these issues and more...

# Exercise

### Who am I?

A bunch of Java and network terms, in full costume, are playing a party game, “Who am I?” They’ll give you a clue — you try to guess who they are based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one attendee, then write down all for whom that sentence applies. Fill in the blanks next to the sentence with the names of one or more attendees.

**Tonight’s attendees:**

**InetSocketAddress, SocketChannel, IP address, host name, port, Socket, ServerSocketChannel, Thread, thread pool, Executors, ExecutorService, CountDownLatch, Runnable, InterruptedException, Thread.sleep()**

| | |
|---|---|
| I need to be shut down or I might live forever | _____ |
| I let you talk to a remote machine | _____ |
| I might be thrown by sleep() and await() | _____ |
| If you want to reuse Threads, you should use me | _____ |
| You need to know me if you want to connect to another machine | _____ |
| I’m like a separate process running on the machine | _____ |
| I can give you the ExecutorService you need | _____ |
| You need one of me if you want clients to connect to me | _____ |
| I can help you make your multithreaded code more predictable | _____ |
| I represent a job to run | _____ |
| I store the IP address and port of the server | _____ |

## New and improved SimpleChatClient

Way back near the beginning of the 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.

This is the New Improved chat client which can both send and receive messages, thanks to the power of multi-threading! Remember, you need to run the chat server first to run this code.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.util.concurrent.*;

import static java.nio.charset.StandardCharsets.UTF_8;

public class SimpleChatClient {
  private JTextArea incoming;
  private JTextField outgoing;
  private BufferedReader reader;
  private PrintWriter writer;

  public void go() {
    setUpNetworking();

    JScrollPane scroller = createScrollableTextArea();

    outgoing = new JTextField(20);

    JButton sendButton = new JButton("Send");
    sendButton.addActionListener(e -> sendMessage());

    JPanel mainPanel = new JPanel();
    mainPanel.add(scroller);
    mainPanel.add(outgoing);
    mainPanel.add(sendButton);

    ExecutorService executor = Executors.newSingleThreadExecutor();
    executor.execute(new IncomingReader());

    JFrame frame = new JFrame("Ludicrously Simple Chat Client");
    frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
    frame.setSize(400, 350);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
  }
```

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

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

We've got a new job, an inner class which is a Runnable. The job is to read from the server's socket stream, displaying any incoming messages in the scrolling text area. We start this job using a single thread executor since we know we only want to run this one job

```
  private JScrollPane createScrollableTextArea() {
    incoming = new JTextArea(15, 30);
    incoming.setLineWrap(true);
    incoming.setWrapStyleWord(true);
    incoming.setEditable(false);
    JScrollPane scroller = new JScrollPane(incoming);
    scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
    return scroller;
  }

  private void setUpNetworking() {
    try {
      InetSocketAddress serverAddress = new InetSocketAddress("127.0.0.1", 5000);
      SocketChannel socketChannel = SocketChannel.open(serverAddress);

      reader = new BufferedReader(Channels.newReader(socketChannel, UTF_8));
      writer = new PrintWriter(Channels.newWriter(socketChannel, UTF_8));

      System.out.println("Networking established.");
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  private void sendMessage() {
    writer.println(outgoing.getText());
    writer.flush();
    outgoing.setText("");
    outgoing.requestFocus();
  }

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

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

A helper method, like we saw back in Chapter 16, to create a scrolling text area

We're using Channels to create a new reader and writer for the SocketChannel that's connected to the server. The writer sends messages to the server, and now we're using a reader so that the reader job can get messages from the server.

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

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).

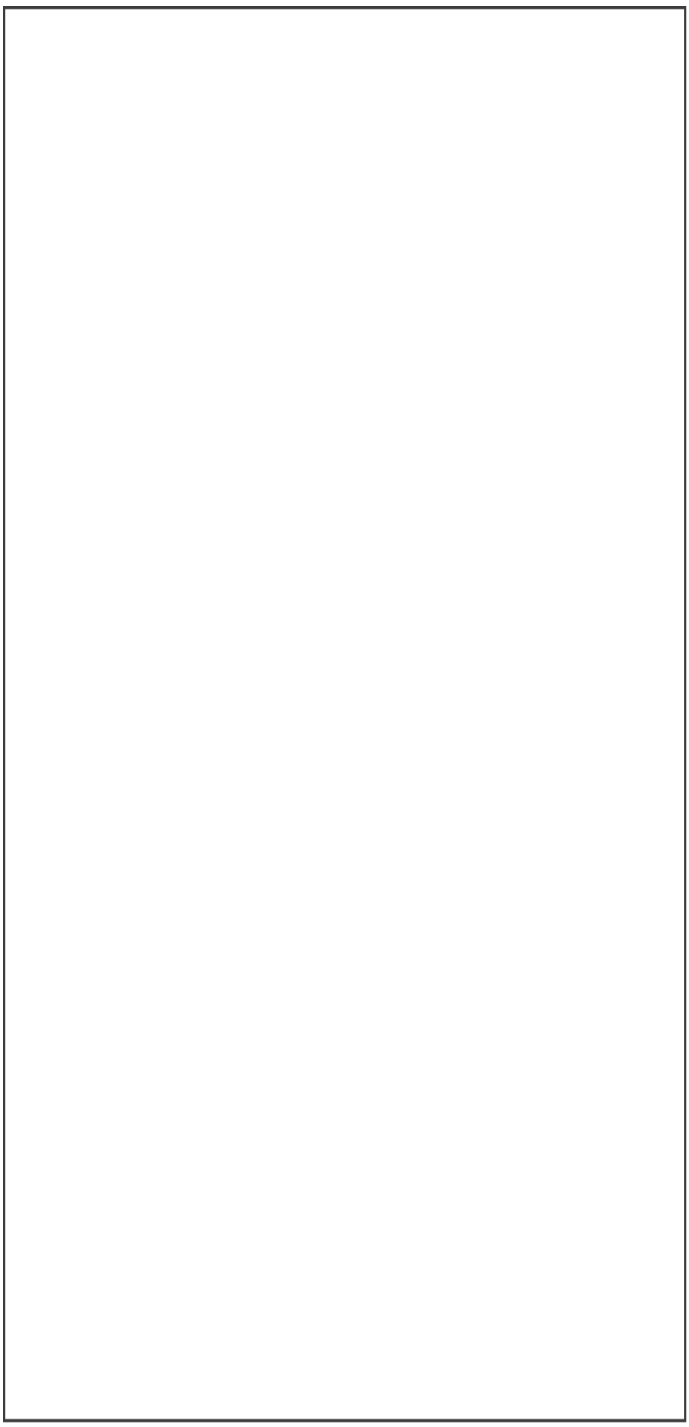
**Remember, the Chat Server code was the Ready-bake code from p490**

# Exercise

## Code Magnets

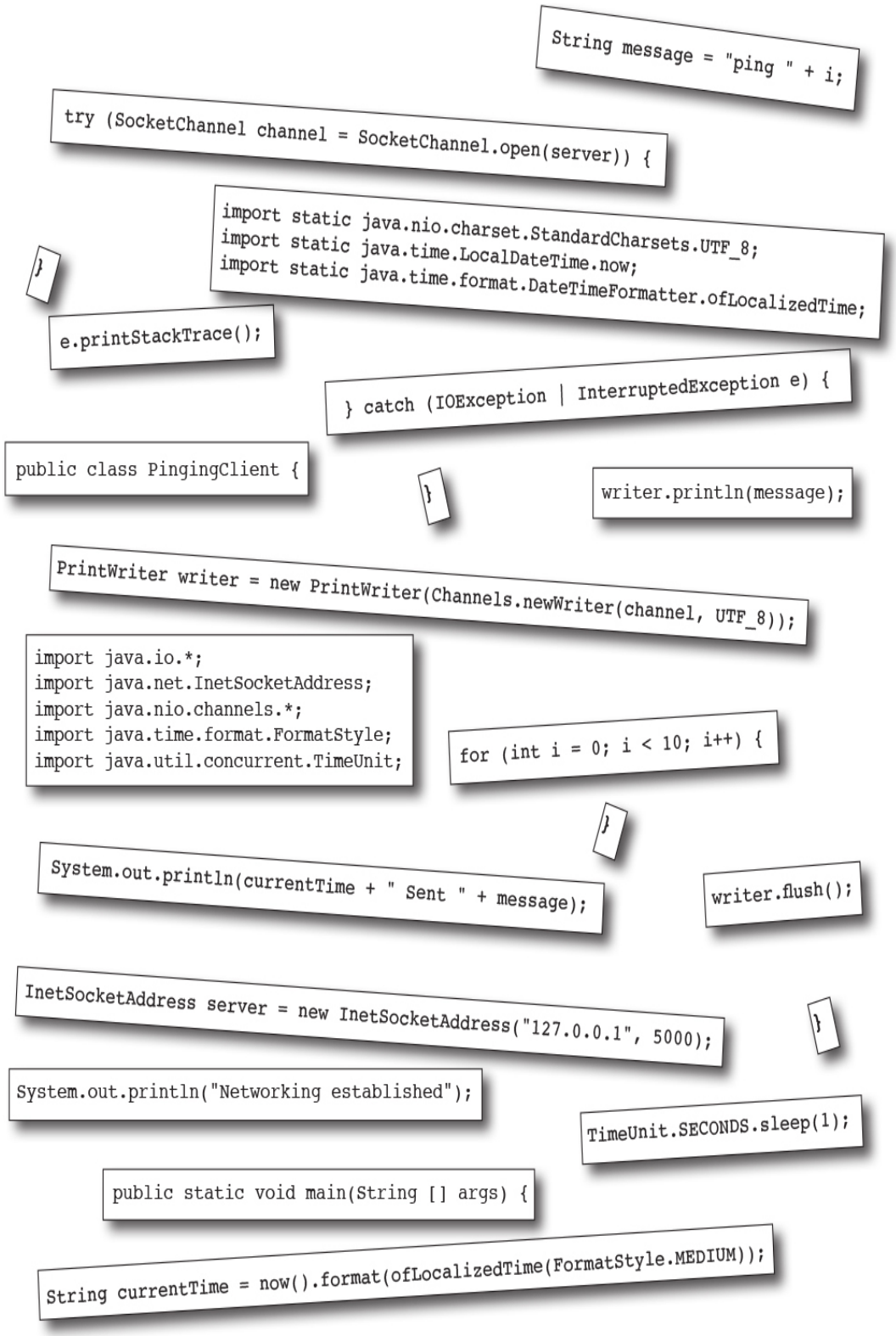
A working Java program is scrambled up on the fridge (see the next page). Can you reconstruct the code snippets on the next page to make a working Java program that produces the output listed below?

To get it to work, you will need to be running the **`SimpleChatServer`** from p490



File Edit Window Help StillThere

```
% java PingingClient
Networking established
09:27:06 Sent ping 0
09:27:07 Sent ping 1
09:27:08 Sent ping 2
09:27:09 Sent ping 3
09:27:10 Sent ping 4
09:27:11 Sent ping 5
09:27:12 Sent ping 6
09:27:13 Sent ping 7
09:27:14 Sent ping 8
09:27:15 Sent ping 9
```


String message = "ping " + i;
try (SocketChannel channel = SocketChannel.open(server)) {
import static java.nio.charset.StandardCharsets.UTF_8;
import static java.time.LocalDateTime.now;
import static java.time.format.DateTimeFormatter.ofLocalizedTime;
}
e.printStackTrace();
} catch (IOException | InterruptedException e) {
public class PingingClient {
}
writer.println(message);
PrintWriter writer = new PrintWriter(Channels.newWriter(channel, UTF_8));
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.time.format.FormatStyle;
import java.util.concurrent.TimeUnit;
for (int i = 0; i < 10; i++) {
}
System.out.println(currentTime + " Sent " + message);
writer.flush();
InetSocketAddress server = new InetSocketAddress("127.0.0.1", 5000);
}
System.out.println("Networking established");
TimeUnit.SECONDS.sleep(1);
public static void main(String [] args) {
String currentTime = now().format(ofLocalizedTime(FormatStyle.MEDIUM));

# Exercise Solutions

## Who am I?

| | |
|---|---|
| I need to be shut down or I might live forever | ExecutorService |
| I let you talk to a remote machine | SocketChannel, Socket |
| I might be thrown by sleep() and await() | InterruptedException |
| If you want to reuse Threads, you should use me | Thread pool, ExecutorService |
| You need to know me if you want to connect to another machine | IP Address, Host name, port |
| I'm like a separate process running on the machine | Thread |
| I can give you the ExecutorService you need | Executors |
| You need one of me if you want clients to connect to me | ServerSocketChannel |
| I can help you make your multithreaded code more predictable | Thread.sleep(), CountDownLatch |
| I represent a job to run | Runnable |
| I store the IP address and port of the server | InetSocketAddress |

## Code Magnets

```
import java.io.*;
import java.net.InetSocketAddress;
import java.nio.channels.*;
import java.time.format.FormatStyle;
import java.util.concurrent.TimeUnit;

import static java.nio.charset.StandardCharsets.UTF_8;
import static java.time.LocalDateTime.now;
import static java.time.format.DateTimeFormatter.ofLocalizedTime;

public class PingingClient {

  public static void main(String[] args) {
    InetSocketAddress server = new InetSocketAddress("127.0.0.1", 5000);
    try (SocketChannel channel = SocketChannel.open(server)) {
      PrintWriter writer = new PrintWriter(Channels.newWriter(channel, UTF_8));
      System.out.println("Networking established");

      for (int i = 0; i < 10; i++) {
        String message = "ping " + i;
        writer.println(message);
        writer.flush();
        String currentTime = now().format(ofLocalizedTime(FormatStyle.MEDIUM));
        System.out.println(currentTime + " Sent " + message);
        TimeUnit.SECONDS.sleep(1);
      }
    } catch (IOException | InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

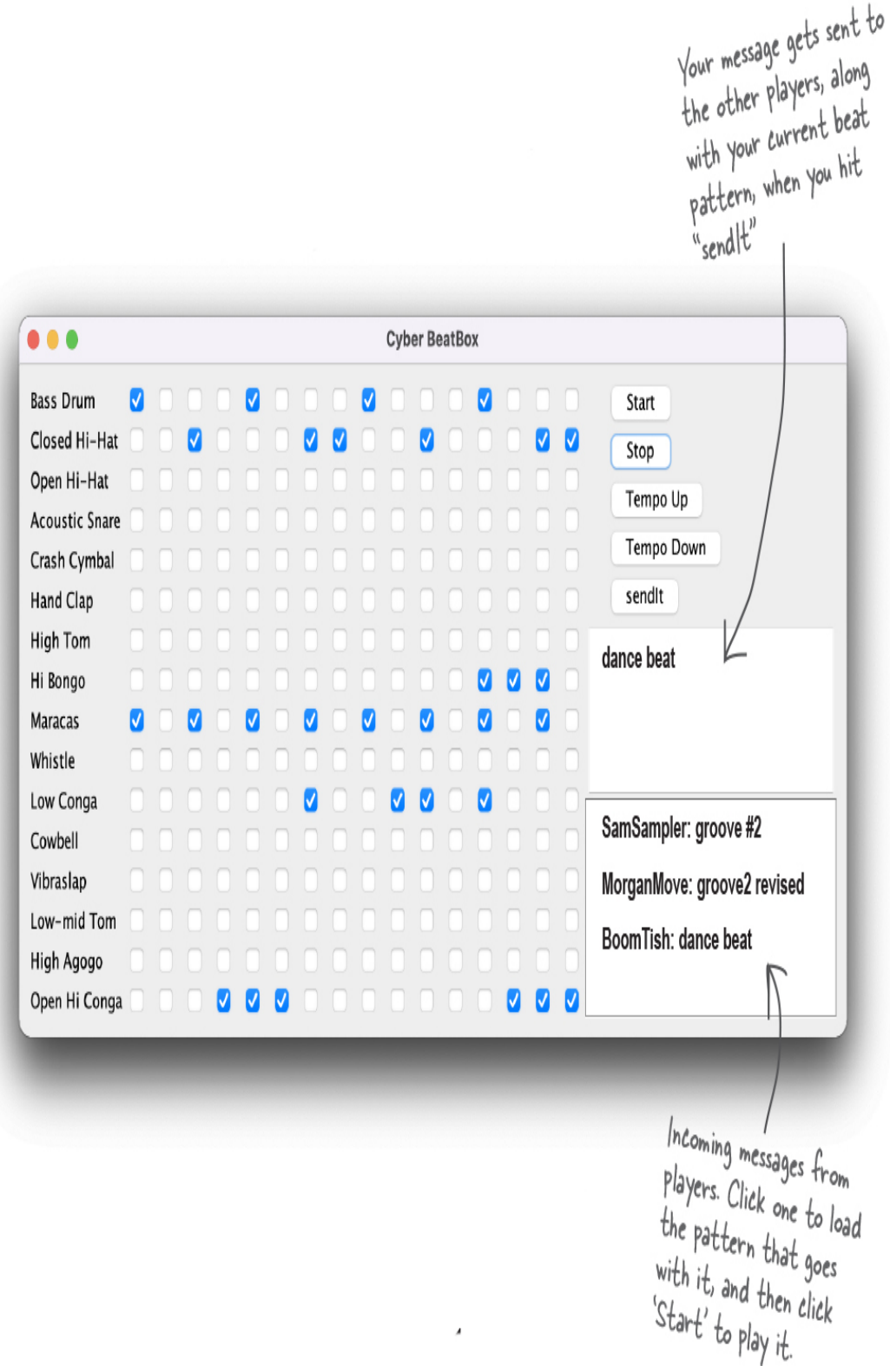
You should get the same output even if you move the sleep() to somewhere else inside this for loop.

This is one way of getting the current time and turning it into a String in the format of Hours:Minutes:Seconds

You can catch the InterruptedException thrown by sleep() inside the for loop, or you can catch all the Exceptions at the end of the method.

Catching all Exceptions at the end because we do the same thing with them all

## Code Kitchen


Your message gets sent to the other players, along with your current beat pattern, when you hit "sendIt"
Cyber BeatBox
Bass Drum
Closed Hi-Hat
Open Hi-Hat
Acoustic Snare
Crash Cymbal
Hand Clap
High Tom
Hi Bongo
Maracas
Whistle
Low Conga
Cowbell
Vibraslap
Low-mid Tom
High Agogo
Open Hi Conga
Start
Stop
Tempo Up
Tempo Down
sendIt
dance beat
SamSampler: groove #2
MorganMove: groove2 revised
BoomTish: dance beat
Incoming messages from players. Click one to load the pattern that goes with it, and then click 'Start' to play it.

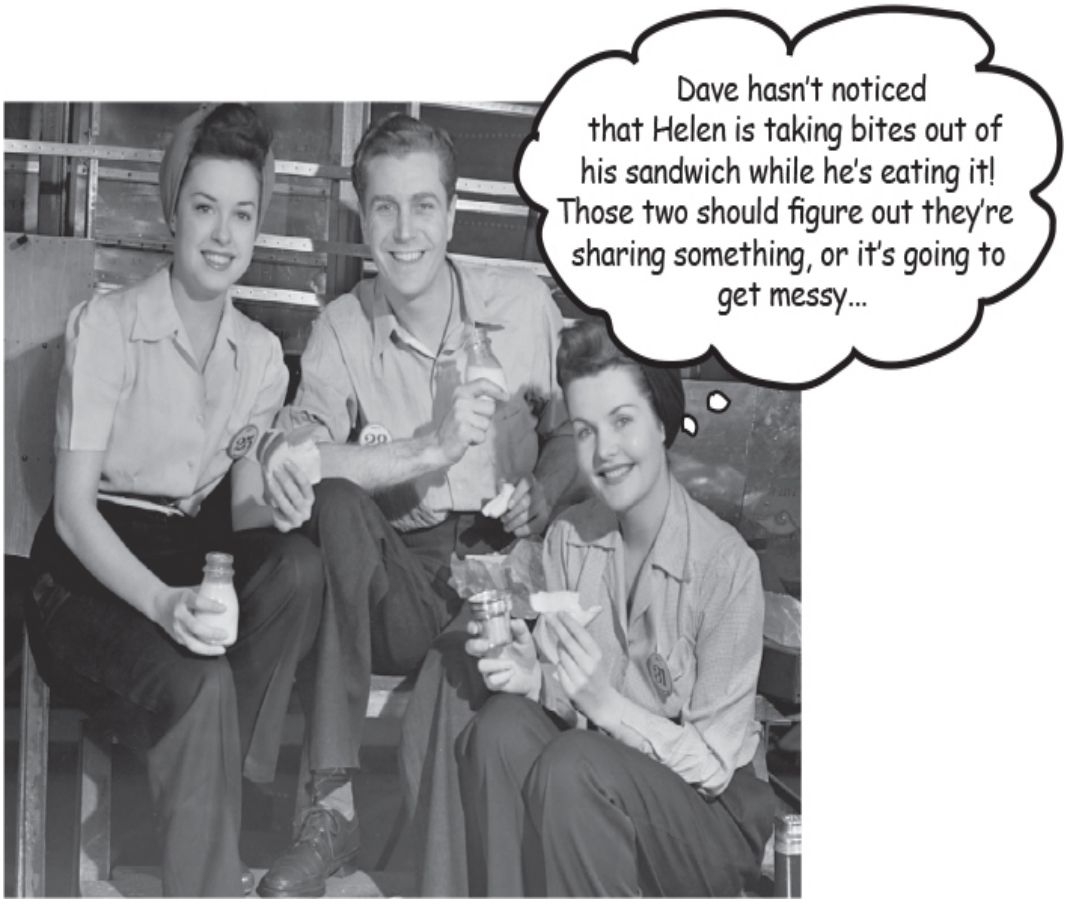
## NOTE

Now you've seen how to build a chat client, we have 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.

# Chapter 18. Writing Multithreaded Programs: Dealing with Concurrency Issues



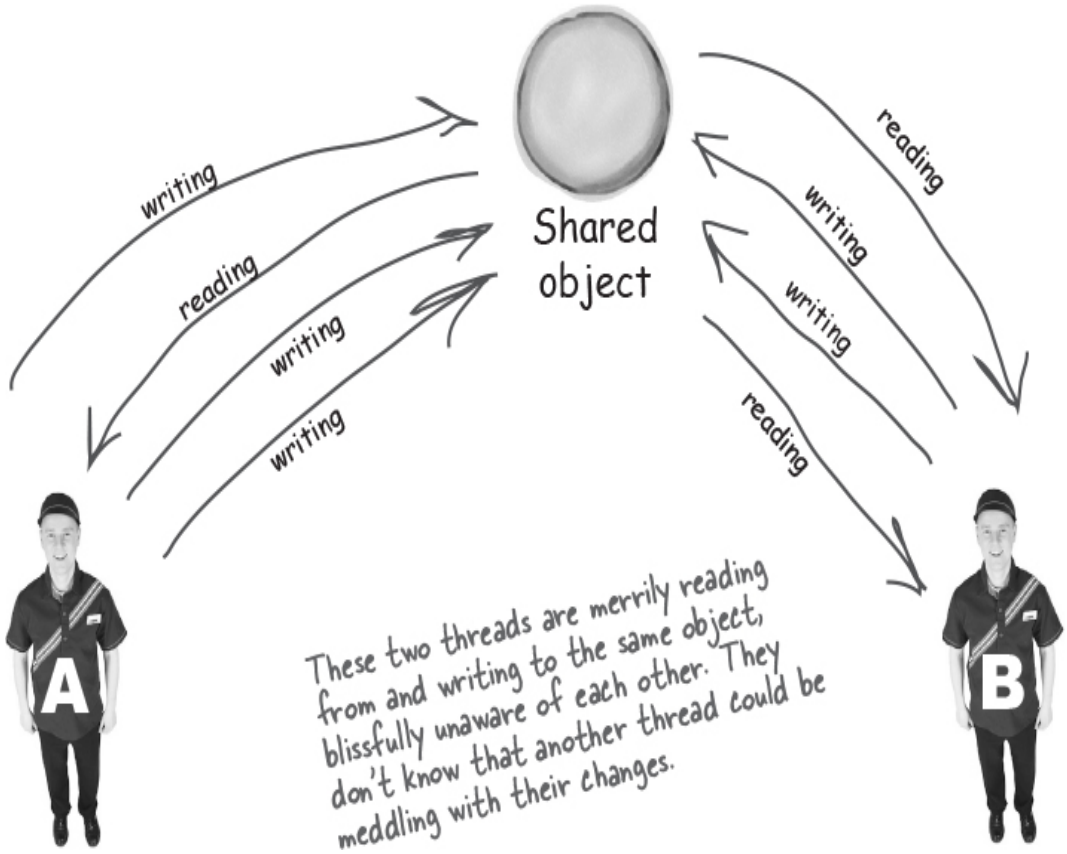

**Doing two or more things at once is hard.** Writing multithreaded code is easy. Writing multithreaded code that works the way you expect, can be much harder. In this final chapter, we're going to show you some of the things that

can go wrong when two or more threads are working at the same time. You'll learn about some of the tools in java.util. concurrent that can help you to write multithreaded code that works correctly. You'll learn how to create immutable objects (objects that don't change) that are safe for multiple threads to use. By the end of the chapter, you'll have a lot of different tools in your toolkit for working with concurrency.

## What could possibly go wrong?

At the end of the last chapter we hinted that things may not all be rainbows and sunshine when you're working with multithreaded code. Well, actually, we did more than hint! We outright said:

> **NOTE**
>
> **"It all comes down to one potentially deadly scenario: two or more threads have access to a single object's data."**




**BRAIN BARBELL**

***Why is it a problem if two threads are both reading and writing?***

If a thread reads the object's data before changing it, why is it a problem that another thread might also be writing at the same time?

## 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 one of the top reasons why couples split up—finances.

Today's troubled pair, Ryan and Monica, share 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 spend money must check the balance in the account *before* withdrawing cash or spending on a card.

It all seemed so simple. But suddenly we're getting hit with overdraft fees!

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

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

Ryan had a full online shopping cart totalling $50. He checked the balance in the account, and saw that it was $100. No problem. So he started the checkout procedure.

And that's where *I* come in, while Ryan was filling in the shipping details, *I* wanted to spend $100. I checked the balance, and it's $100 (because Ryan hasn't clicked the "Pay" button yet), so I think, no problem. So I spend the money, and again no problem. But then Ryan finally pays, and we're suddenly overdrawn! He didn't know that I was spending money at the same time, 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 help them with their online shopping addiction, but can we make sure one of them can't start spending while the other one is shopping?

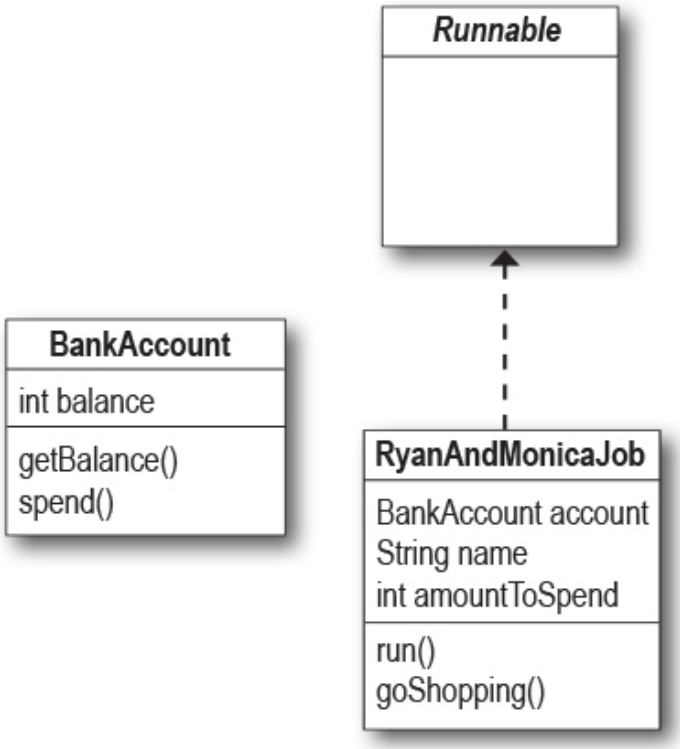
Take a moment and think about that while we go to a commercial break.

# 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 RyanAndMonicaJob. There's also a RyanAndMonicaTest with a main method to run everything. The RyanAndMonicaJob class implements Runnable, and represents the behavior that Ryan and Monica both have— checking the balance and spending money.

The RyanAndMonicaJob class has instance variables for the shared BankAccount, the person's name and for the amount they want to spend. The code works like this:




1. ① **Make an instance of the shared bank account**

   Creating a new one will set up all the defaults correctly

```
BankAccount account = new BankAccount();
```

2. ② **Make one instance of RyanAndMonicaJob for each person.**

   We need one job for each person. We also need to give them access to the BankAccount, and tell them how much to spend.

```
RyanAndMonicaJob ryan = new RyanAndMonicaJob("Ryan", account,
50);
RyanAndMonicaJob monica = new RyanAndMonicaJob("Monica",
account, 100);
```

3. ③ **Create an ExecutorService and give it the two jobs**

   Since we know we have two jobs, one for Ryan and one for Monica, we can create a fixed-sized thread pool with two threads

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.execute(ryan);
executor.execute(monica);
```

4. ④ **Watch both jobs run**

   One thread represents Ryan, the other represents Monica. Both threads check the balance before spending money. Remember that when more than one thread is running at a time, you can't assume that your thread is the only one making changes to a shared object (e.g. the BankAccount). Even if there's only two lines of code related to the shared object, and they're right next to each other.

```
if (account.getBalance() >= amount) {
  account.spend(amount);
} else {
  System.out.println("Sorry, not enough money");
}
```

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

*This should protect against overdrawing the account.*

*Except... If Ryan and Monica are spending money at the same time, the money in the bank account might be gone before the other one can spend it!*

## The Ryan and Monica example

```
import java.util.concurrent.*;

public class RyanAndMonicaTest {
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    RyanAndMonicaJob ryan = new RyanAndMonicaJob("Ryan", account, 50);
    RyanAndMonicaJob monica = new RyanAndMonicaJob("Monica", account, 100);
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(ryan);
    executor.execute(monica);
    executor.shutdown();
  }
}

class RyanAndMonicaJob implements Runnable {
  private final String name;
  private final BankAccount account;
  private final int amountToSpend;
  RyanAndMonicaJob(String name, BankAccount account, int amountToSpend) {
    this.name = name;
    this.account = account;
    this.amountToSpend = amountToSpend;
  }

  public void run() {
    goShopping(amountToSpend);
  }

  private void goShopping(int amount) {
    if (account.getBalance() >= amount) {
      System.out.println(name + " is about to spend");
      account.spend(amount);
      System.out.println(name + " finishes spending");
    } else {
      System.out.println("Sorry, not enough for " + name);
    }
  }
}

class BankAccount {
  private int balance = 100;
  public int getBalance() {
    return balance;
  }
  public void spend(int amount) {
    balance = balance - amount;
    if (balance < 0) {
      System.out.println("Overdrawn!");
    }
  }
}
```

There will be only ONE instance of the BankAccount. That means both threads will access this one account.

Make two jobs that will do the withdrawal from the shared bank account, one for Monica and one for Ryan, passing in the amount they're going to spend

Start both jobs running

Don't forget to shut the pool down

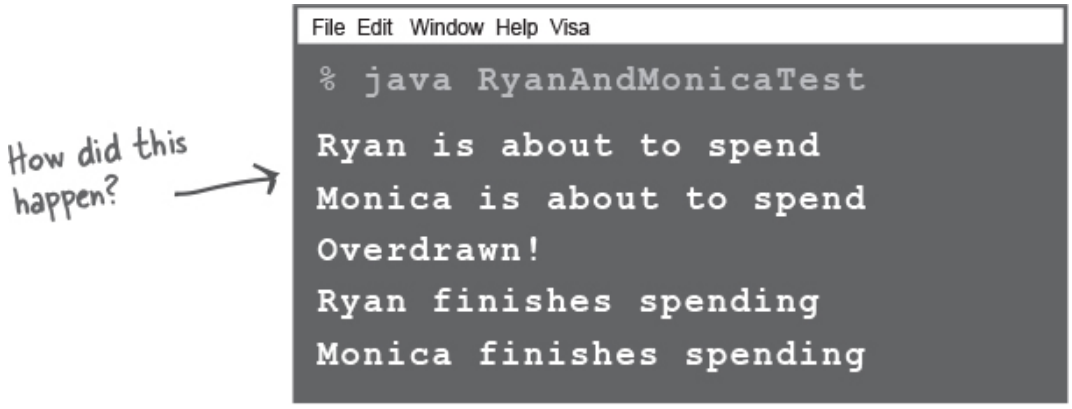
Create a new thread pool with two threads for our two jobs

The run() method just calls goShopping() with the amount they need to spend

Check the account balance, and if there's enough money, we go ahead and spend the money, just like Ryan and Monica did.

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

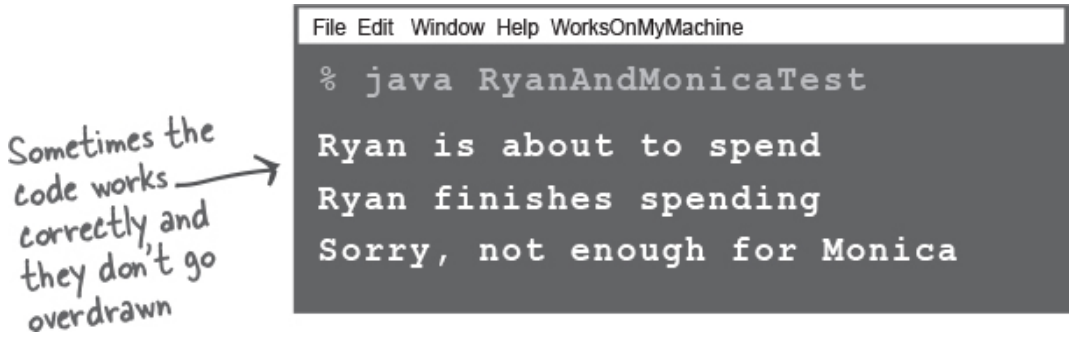
The account starts with a balance of $100.




**NOTE**

**This code is not deterministic, it doesn't always produce the same result every time. You may need to run it a few times before you see the problem.**

**This is common with multithreaded code, since it depends upon which threads start first, and when each thread gets its time on a CPU core.**




**The goShopping() method always checks the balance before making a withdrawal, but still we went overdrawn.**

**Here's one scenario:**

Ryan checks the balance, sees that there's enough money, and goes to check out.

Meanwhile, Monica checks the balance. She, too, sees that there's enough money. She has no idea that Ryan is about to pay for something.

Ryan completes his purchase.

Monica completes her purchase. Big Problem! In between the time when she checked the balance and spent the money, Ryan had already spent money!

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

Monica must be stopped from getting into the account until Ryan has finished, and vice-versa.

**They need a lock for account access!**

**The lock works like this:**

1. 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. 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. 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 until Ryan unlocks the account and returns the key.

   Now, even if Ryan gets distracted after he checks the balance, he has a guarantee that the balance will be the same when he spends the money, because he kept the key while he was doing something else!

**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 to check the balance and spend the money as one atomic thing.

We need to make sure that once a thread starts a shopping transaction, *it must be allowed to finish* before any other thread changes the bank account.

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 spend the money *before any other thread can check the account balance!*

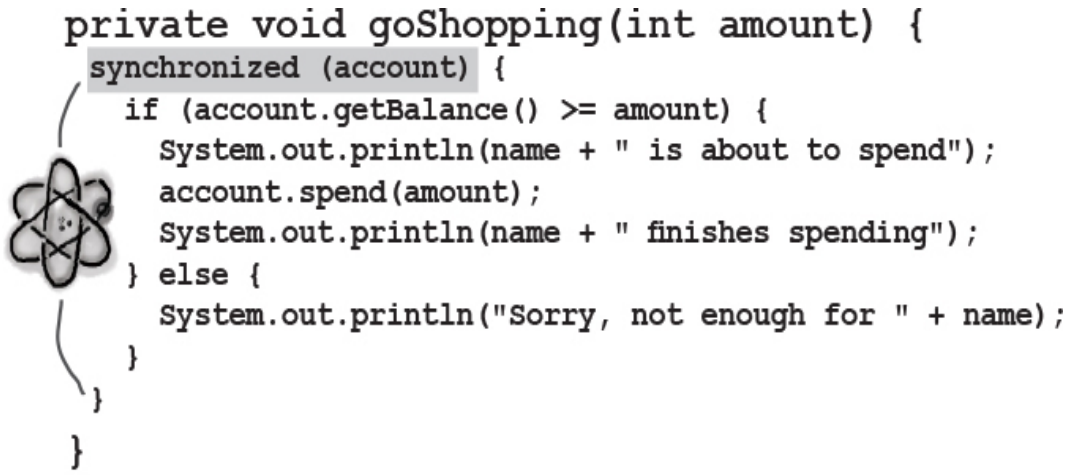
Use the **`synchronized`** keyword on a method, or with an object, to lock an object so only one thread can use it at a time.

That's how you protect the bank account! We can put a lock on the bank account inside the method that does the banking transaction. That way, one thread gets to complete the whole transaction, start to finish, even if that thread is taken out of the "running" state by the thread scheduler, or another thread is trying to make changes at exactly the same time.

On the next couple of pages we'll look at the different things that we can lock. With the Ryan and Monica example, it's quite simple—we want to wrap our shopping transaction in a block that locks the bank account:

**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 code that acts on that data.**



```
private void goShopping(int amount) {
  synchronized (account) {
    if (account.getBalance() >= amount) {
      System.out.println(name + " is about to spend");
      account.spend(amount);
      System.out.println(name + " finishes spending");
    } else {
      System.out.println("Sorry, not enough for " + name);
    }
  }
}
```

### NOTE

(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.)

### THERE ARE NO DUMB QUESTIONS

**Q: Why don't you just synchronize all the getters and setters from the class with the data you're trying to protect?**

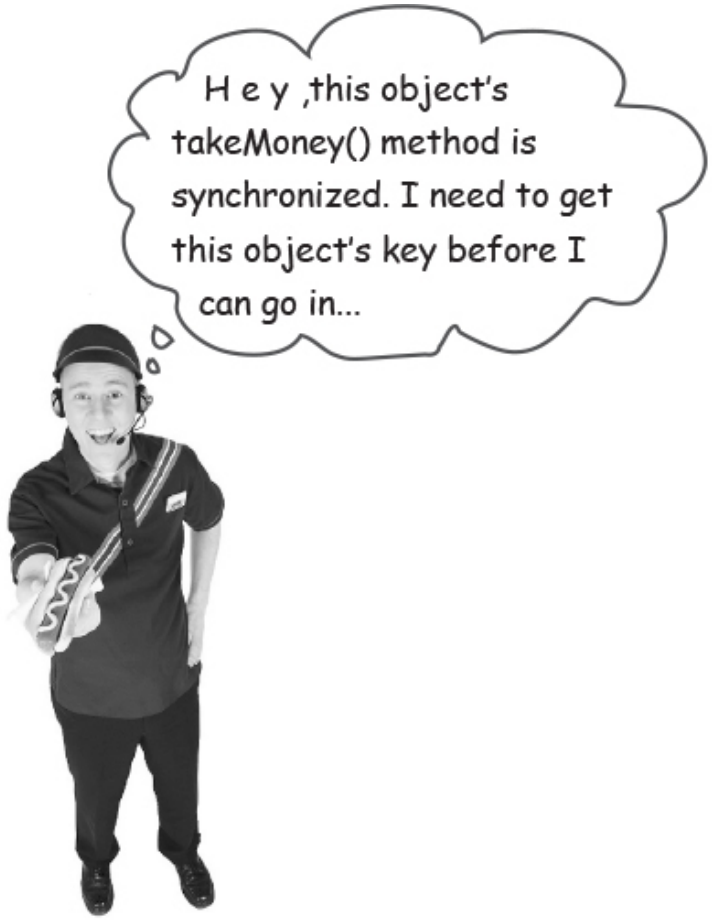
**A:** Synchronizing the getters and setters isn't enough. Remember, the point of synchronization is to make a specific section of code work ATOMICALLY. In other words, it's not just the individual methods we care about, it's methods that require ***more than one step to complete***!

Think about it. We added a synchronized block inside the goShopping() method. If getBalance() and spend() were both synchronized instead, it wouldn't help - Ryan (or Monica) would have checked the balance returned the key! The whole point is to keep the key until ***both*** operations are completed.

## 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 is a **synchronized block** for an object (like in the last page), or a class has **synchronized methods**. A method is synchronized if it has the synchronized keyword in the method declaration.

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 doesn't *only* mean two threads can't enter the same

method. It means you can't have two threads entering *any* of the synchronized methods. If you have two synchronized methods on the same object, method1() and method2(), if one thread is in method1(), a second thread can't enter method1(), obviously, but it *also can't enter method2()*, or any other synchronized method on that object.

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 or block. 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.

**NOTE**

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

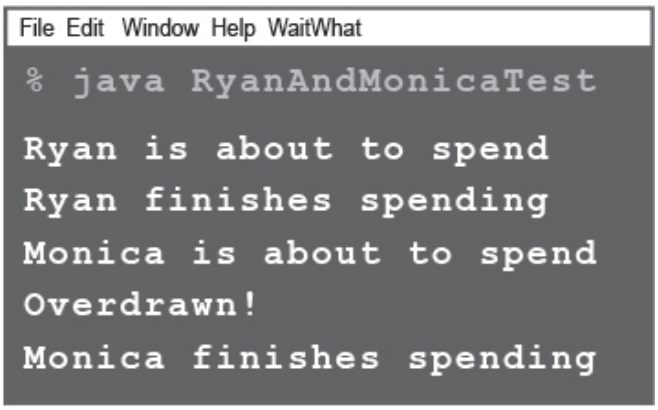
**Most of the time, the lock is unlocked and nobody cares.**

**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.**

## Using synchronized methods

Can we synchronize the goShopping() method to fix Ryan and Monica's problem?

```
private synchronized void goShopping(int amount) {
  if (account.getBalance() >= amount) {
    System.out.println(name + " is about to spend");
    account.spend(amount);
    System.out.println(name + " finishes spending");
  } else {
    System.out.println("Sorry, not enough for " + name);
  }
}
```



NOTE

**It does NOT work!**

The synchronized keyword locks an object. The goShopping() method is in RyanAndMonicaJob. Synchronizing an instance method means "lock *this* RyanAndMonicaJob instance". However, there are *two* instances of RyanAndMonicaJob, one is "ryan" and the other is "monica". If "ryan" is locked, "monica" can still make changes to the bank account, she doesn't care that the "ryan" job is locked.

The object that needs locking, the object these two threads are fighting over, is the BankAccount. Putting synchronized on a method in RyanAndMonicaJob (and locking a RyanAndMonicaJob instance) isn't going to solve anything.

## It's important to lock the correct object

Since it's the BankAccount object that's shared, you could argue it should be the BankAccount that's in charge of making sure it is safe for multiple threads to use. The spend() method on BankAccount could make sure there's enough money *and* debit the account in a single transaction.

```
class RyanAndMonicaJob implements Runnable {
  // ...rest of the RyanAndMonicaJob class
  // the same as before...

  private void goShopping(int amount) {
    System.out.println(name + " is about to spend");
    account.spend(name, amount);
    System.out.println(name + " finishes spending");
  }
}

class BankAccount {
  // other methods in BankAccount...

  public synchronized void spend(String name, int amount) {
    if (balance >= amount) {
      balance = balance - amount;
      if (balance < 0) {
        System.out.println("Overdrawn!");
      }
    } else {
      System.out.println("Sorry, not enough for " + name);
    }
  }
}
```

This would no longer need to check the balance before spending if we know the BankAccount spend() method checks for us

Locks the BankAccount instance the two threads are using

Ryan and Monica SHOULDN'T go overdrawn now, this should never be the case

Do the balance check and balance decrease in the BankAccount itself. If this method is synchronized, it becomes an atomic transaction that can only be done in full by one thread at a time

### 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.

## The dreaded "Lost Update" problem

Here's another classic concurrency problem, Sometimes you'll hear them called "race conditions", where two or more threads are changing the same data at the same time. It's closely related to the Ryan and Monica story, so 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


Even if we used the more common syntax: **`balance++`** there is no guarantee that the compiled bytecode will be an "atomic process". In fact, it probably won't - it's actually multiple operations: a read of the current value, then adding one to that value and setting it back into the original variable.

In the "Lost Update" problem, we have many threads trying to increment the balance. Take a look at the code, and then we'll look at the real problem:

```
public class LostUpdate {
  public static void main(String[] args) throws InterruptedException {
    ExecutorService pool = Executors.newFixedThreadPool(6);

    Balance balance = new Balance();
    for (int i = 0; i < 1000; i++) {
      pool.execute(() -> balance.increment());
    }
    pool.shutdown();
    if (pool.awaitTermination(1, TimeUnit.MINUTES)) {
      System.out.println("balance = " + balance.balance);
    }
  }
}

class Balance {
  int balance = 0;

  public void increment() {
    balance++;
  }
}
```

Create a thread pool to run all the jobs. If you add more threads here, you may see even more missing updates

Run 1000 attempts to update the balance, on different threads

Make sure the pool has finished running all the updates before printing the final balance. In theory, this should be 1000. If it's any less than that, we've lost an update!

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). You might think that "++" is an atomic operation, but it is not.

## Let's run this code...

1. Thread A runs for awhile

Reads balance: 0
Set the value of balance to 0 + 1.
Now balance is 1

Reads balance: 1
Set the value of balance to 1 + 1.
Now balance is 2


2. Thread B runs for awhile


Reads balance: 2
Set the value of balance to 2 + 1.
Now balance is 3

Reads balance: 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

Reads balance: 3
Set the value of balance to 3 + 1.
Now balance is 4

Reads balance: 4
Set the value of balance to 4 + 1.
Now balance is 5


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


Set the value of balance to 3 + 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 steps in the method (read of balance and increment of balance) as one unbreakable unit.

```
public synchronized void increment() {
  balance++;
}
```

Classic concurrency gotcha: this looks like a single operation, but it's actually more than one – it's a read of the balance, an increment, and an update to the balance.

***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

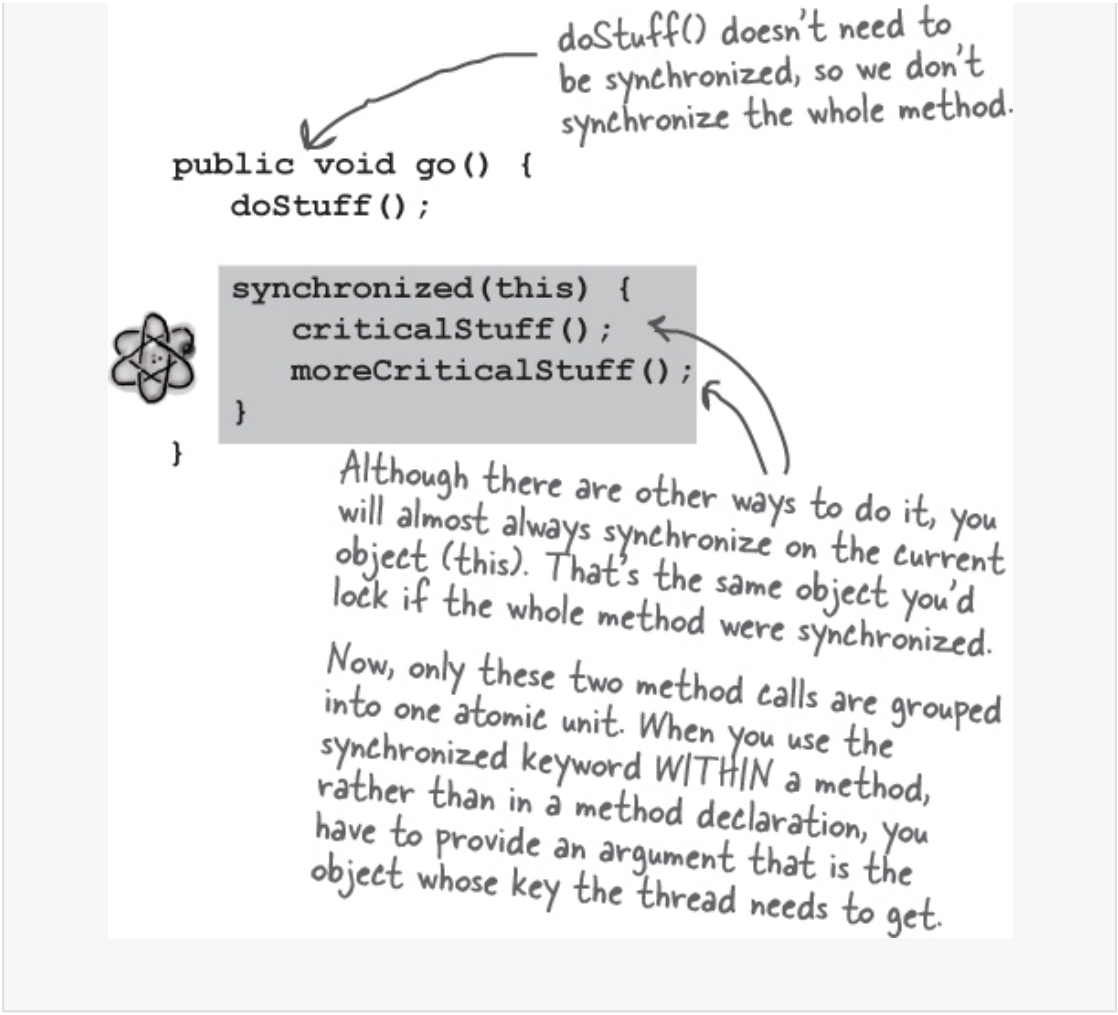
**Q: Sounds like it's a good idea to synchronize everything, just to be thread-safe.**

**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! (Which we'll see in a couple of pages).

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. Remember, 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 (we used this in our first solution to Ryan and Monica's problem).




1. ① Thread A runs for awhile

   Attempt to enter the increment() method.

   The method is synchronized, so **get the key** for this object

   Reads balance: 0

   Set the value of balance to 0 + 1.

   Now balance is 1

   **Return the key** (it completed the increment() method). Re-enter the increment() method and **get the key**.

Reads balance: 1

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


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

   Set the value of balance 1 + 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. 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.

   Reads balance: 2

   [continues to run...]


## Deadlock, a deadly side of synchronization

Synchronization saved Ryan and Monica from using their bank account at the same time, and has saved us from losing updates. But we also mentioned that we shouldn't synchronize everything. one reason being that synchronization can slow your program down.

There's another important consideration: we need to be careful using 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.

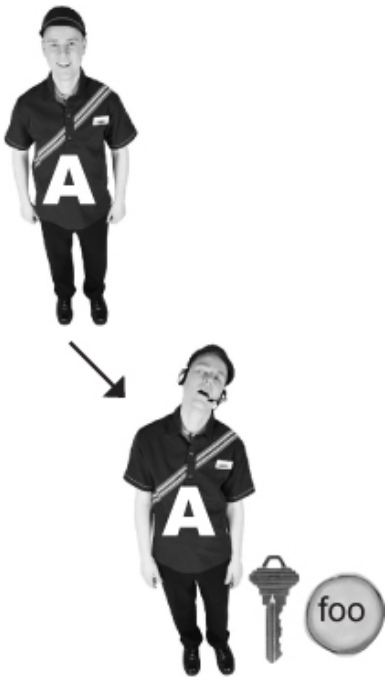
If you're familiar with databases or other application servers, you might recognize the problem; databases often have a locking mechanism somewhat like synchronization. But a real transaction management system can sometimes deal with deadlock. It might assume, for example, that deadlock might have occurred 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. We're not going to go into more detail about deadlock than you see on this page, so if you find yourself writing multithreaded code, you might want to study "Java Concurrency in Practice" by Brian Goetz et al. It goes into a lot of detail about the sorts of problems you can face with concurrency (like deadlock), and approaches to address these problems.

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

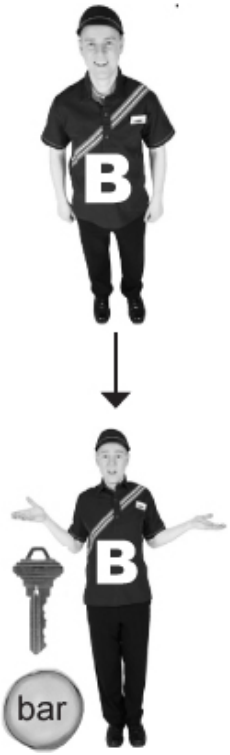

**A simple deadlock scenario:**

1. ① Thread A enters a synchronized method of object *foo*, and gets the key. 


Thread A goes to sleep, holding the *foo* key.

2. ② Thread B enters a synchronized method of object *bar*, and gets the key. 




Thread B tries to enter a synchronized method of object *foo*, but can't get ***that*** key (because A has it). B goes to the waiting lounge, until the *foo* key is available. B keeps the *bar* key.

3. ③ Thread A wakes up (still holding the *foo* key) and tries to enter a synchronized method on object *bar*, but can't get ***that*** key because B has it. A goes to the waiting lounge, until the *bar* key is available (it never will be!)


Thread A can't run until it can get the *bar* key, but B is holding the *bar* key and B can't run until it gets the *foo* key that A is holding and...

## You don't always have to use synchronized

Since synchronization can come with some costs (like performance and potential deadlocks), you should know about other ways to manage data that's shared between threads. The java.util. concurrent package has lots of classes and utilities for working with multithreaded code.

### Atomic variables

If the shared data is an int, long or boolean, we might be able to replace it with an *atomic variable*. These classes provide methods that are atomic, i.e. can safely be used by a thread without worrying about another thread changing the object's values at the same time.

There are few types of atomic variable, e.g. **AtomicInteger**, **AtomicLong**, **AtomicBoolean**, and **AtomicReference**.

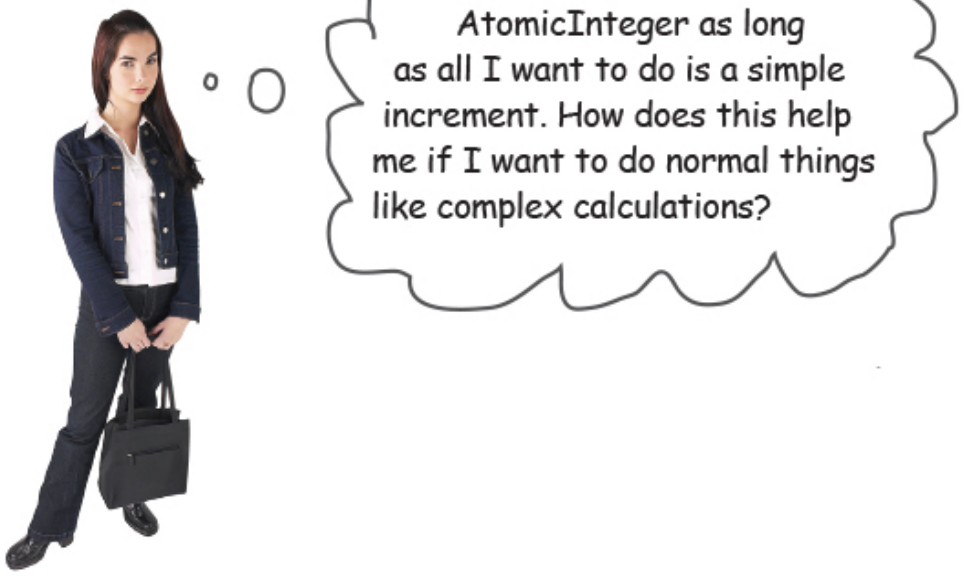
We can fix our Lost Update problem with an AtomicInteger, instead of synchronizing the increment method.

```
class Balance {
  AtomicInteger balance = new AtomicInteger(0);

  public void increment() {
    balance.incrementAndGet();
  }
}
```

Use an AtomicInteger initialized to zero, instead of an int value

incrementAndGet atomically adds one to the value, i.e. even if it's used by multiple threads it will safely increase the value by one in a single operation. The incrementAndGet method returns the new, updated value, but we don't need that for our example, we're not going to use the returned value.

No need to add "synchronized" when you're using atomic operations



Atomic variables get more interesting when you use their *compare-and-swap* **(CAS)** operations. CAS is yet another way to make an atomic change to a value. You can use CAS on atomic variables by using the **`compareAndSet`** method. Yes, it's a slightly different name! Gotta love programming, where naming is always the hardest problem to solve.

The compareAndSet method takes a value which is what you *expect* the atomic variable to be, compares it to the *current* value, and if that matches, *then* the operation will complete.

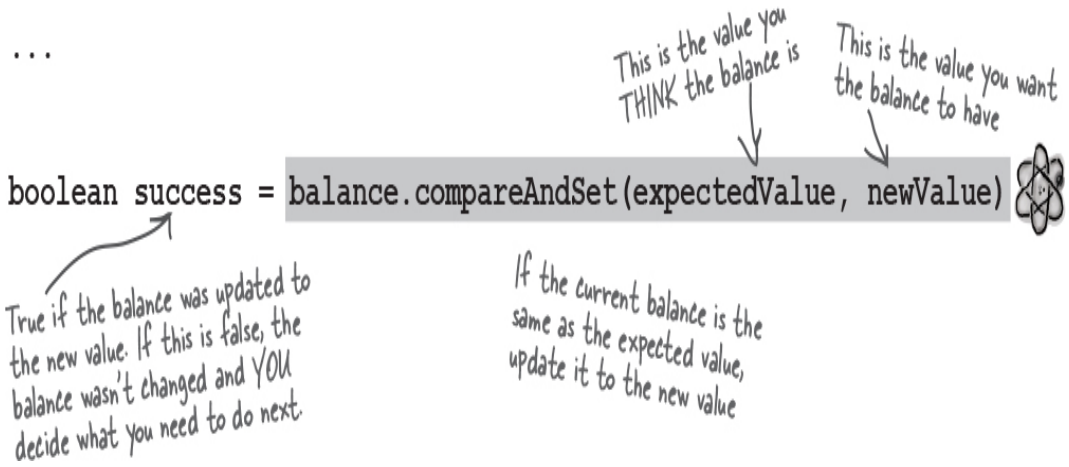
In fact, we can use this to fix our Ryan and Monica problem, instead of locking the whole bank account with **`synchronized`**.

## Compare-and-swap with atomic variables

How could we make use of atomic variables, and CAS (via **`compareAndSet`**), to solve Ryan and Monica's problem?

Since Ryan and Monica were both trying to access an int value, the account balance, we could use an AtomicInteger to store that balance. We could then use **`compareAndSet`** to update the balance when someone wants to spend money.

```
private AtomicInteger balance = new AtomicInteger(100);

...

boolean success = balance.compareAndSet(expectedValue, newValue);
```



In plain English:

> **NOTE**
>
> **"Set the balance to this new value only if the current balance is the same as this expected value, and tell me if the balance was actually changed."**

Compare-and-swap uses *optimistic locking*. Optimistic locking means you don't stop all threads from getting to the object, you *try* to make the change but you embrace the fact that the change ***might not happen***. If it doesn't succeed, you decide what to do. You might decide to try again, or to send a message letting the user know it didn't work.

This may be more work than simply locking all other threads out from the object, but it can be faster than locking everything. For example, when the chances of multiple writes happening at the same time are very low, or if you have a lot of threads reading and not so many writing, then you may not want to pay the price of a lock on every write.

> **When you're using CAS operations, you have to deal with the times when the operation does NOT succeed**

## Ryan and Monica, going atomic

Let's see the whole thing in action in Ryan and Monica's bank account. We'll put the balance in an AtomicInteger, and use compareAndSet to make an *atomic* change to the balance.

```
import java.util.concurrent.atomic.AtomicInteger;

class BankAccount {
  private final AtomicInteger balance = new AtomicInteger(100);

  public int getBalance() {
    return balance.get();
  }

  public void spend(String name, int amount) {

    int initialBalance = balance.get();
    if (initialBalance >= amount) {

      boolean success = balance.compareAndSet(initialBalance, initialBalance - amount);

      if (!success) {
        System.out.println("Sorry " + name + ", you haven't spent the money.");
      }
    } else {
      System.out.println("Sorry, not enough for " + name);
    }
  }
}
```

Store the balance in an AtomicInteger, with the same initial value ($100) as before

Use the get() method to get the int value of the AtomicInteger

Not synchronized

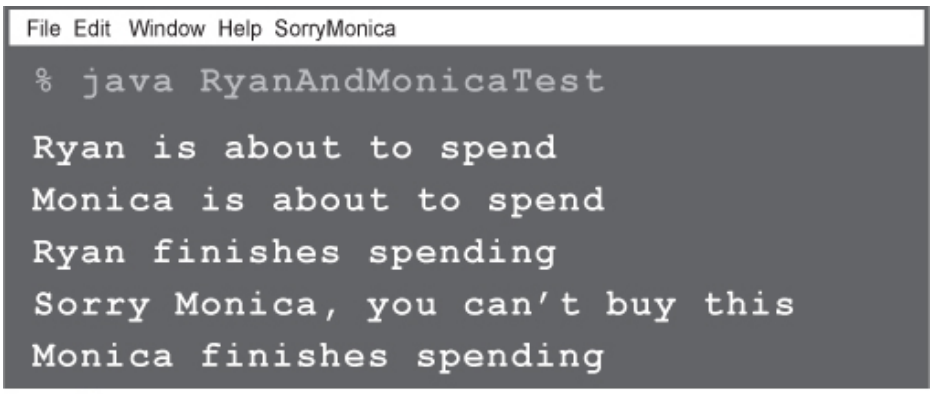
Like before, check if there's enough money. This time, keep a record of the balance

The balance will NOT be changed if the initial balance does not match the actual balance right now

Pass in the balance from when we checked if there was enough money

This is the "spend", subtracting the amount spent from the account balance

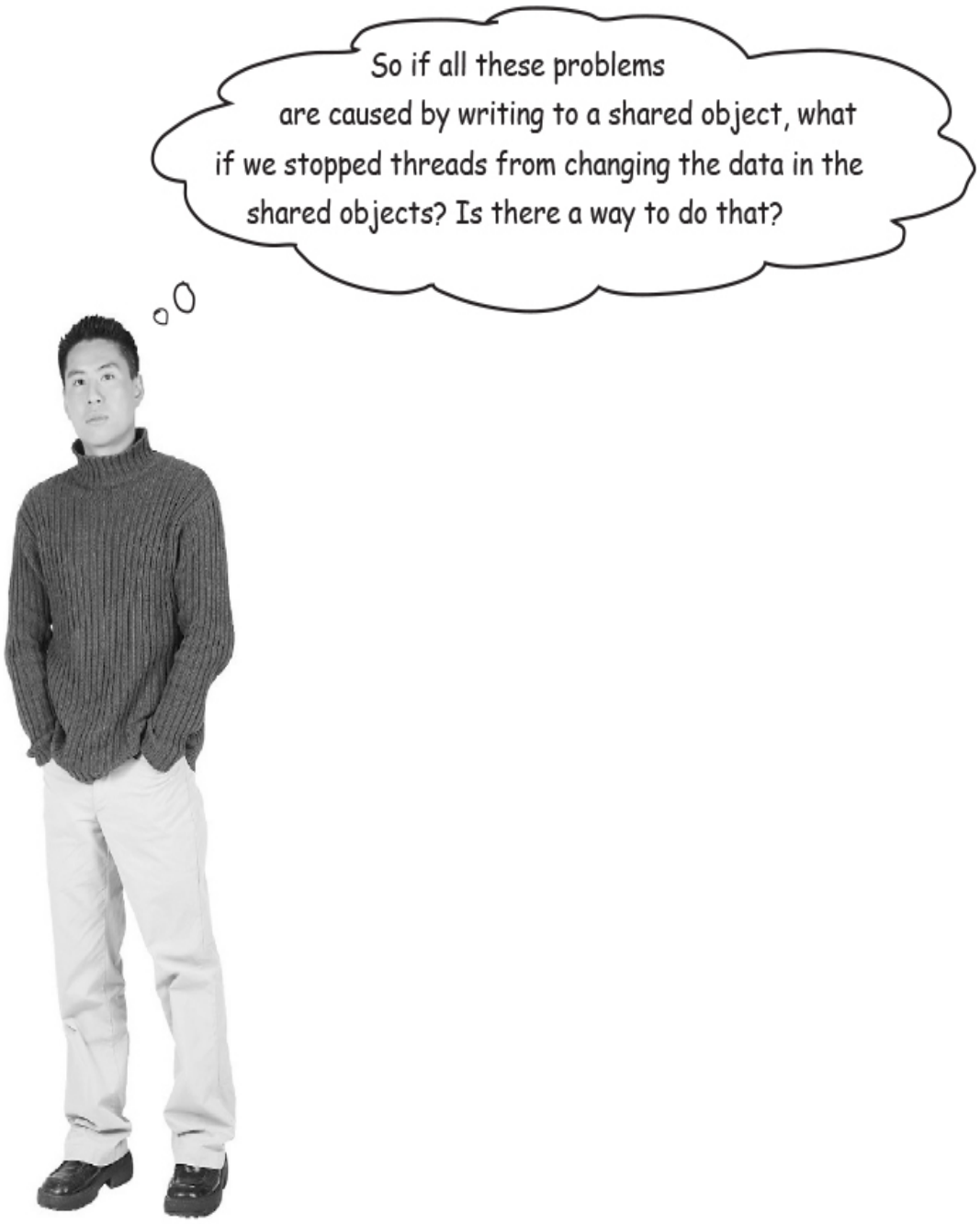
If success was false, the money was NOT spent. Tell Ryan or Monica it didn't work and they can decide what to do




Monica was able to start her shopping, but by the time she came to pay, the bank said no. At least they didn't go overdrawn!

## NOTE

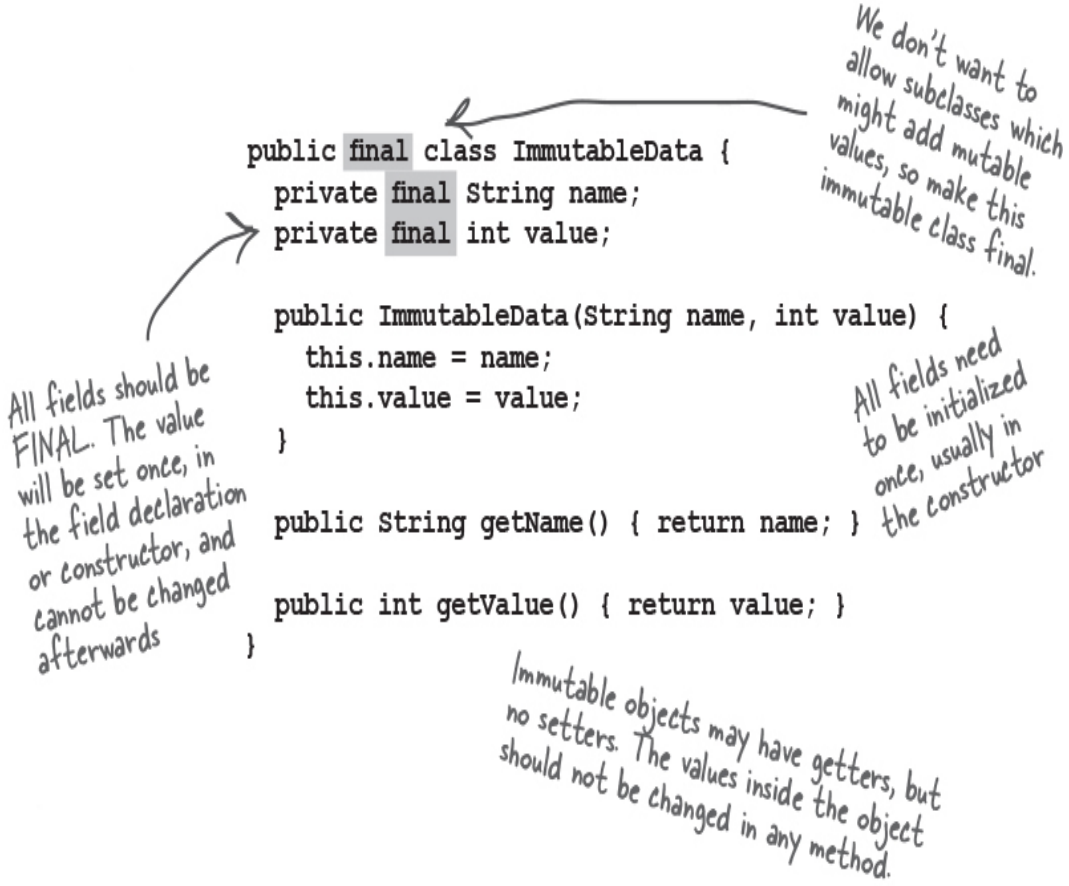
**java.util.concurrent has lots of useful classes and utilities for working with multithreaded code. Take a look at what's there!**




**Make an object <u>immutable</u> if you're going to share it between threads and you don't want the threads to change its data.**

The very best way to know *for sure* that another thread isn't changing your data is to make it impossible to change the data in the object. When an object's data cannot be changed, we call it an **immutable object**.

## Writing a class for immutable data



**BRAIN BARBELL**

There are times when adding the final keyword isn't enough to prevent changes. When do you think that might be the case? We'll give you a clue...

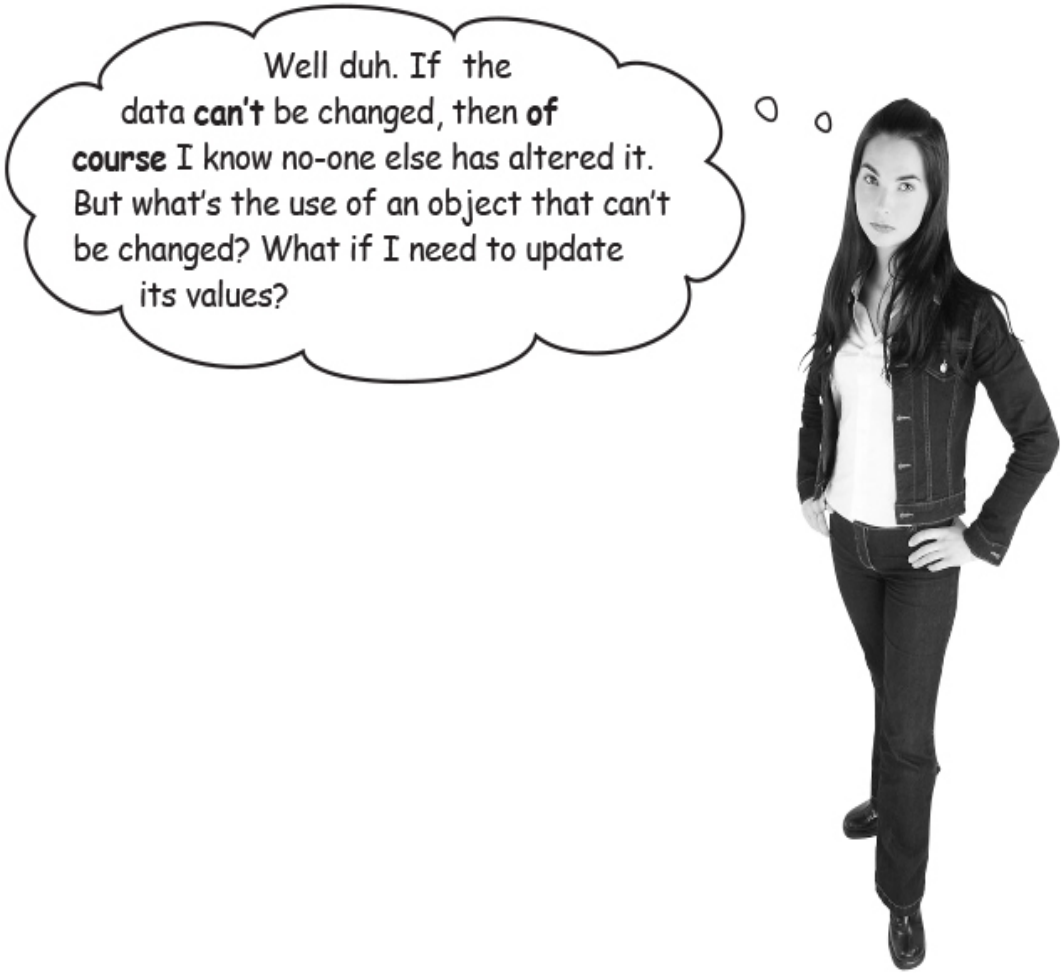
## Using immutable objects

It is terribly convenient to be able to change data on a shared object and assume that all the other threads will be able to see these changes.

However, we've also seen that while it's *convenient*, it's not very *safe*.

On the other hand, when a thread is working with an object that cannot be changed, it can make assumptions about the data in that object, e.g. once the thread has read a value from the object, it knows that data can't change.

We don't need to use synchronization or other mechanisms to control who changes the data because it can't change.




**Working with immutable objects means thinking in a different way.**

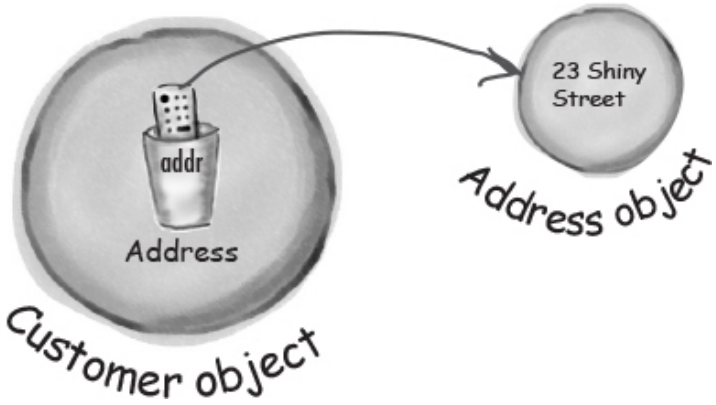
Instead of making changes to the *same* object, we *replace* the old object with a new one. The new object has the updated values, and any threads which need the new values need to use the new object.

What happens to the old object? Well, if it's still being used by something (and it might be - it's perfectly valid sometimes to work with older data) it will hang around on the heap. If it's not being used, it'll be garbage collected, and we don't have to worry about it any more.

## Changing immutable data

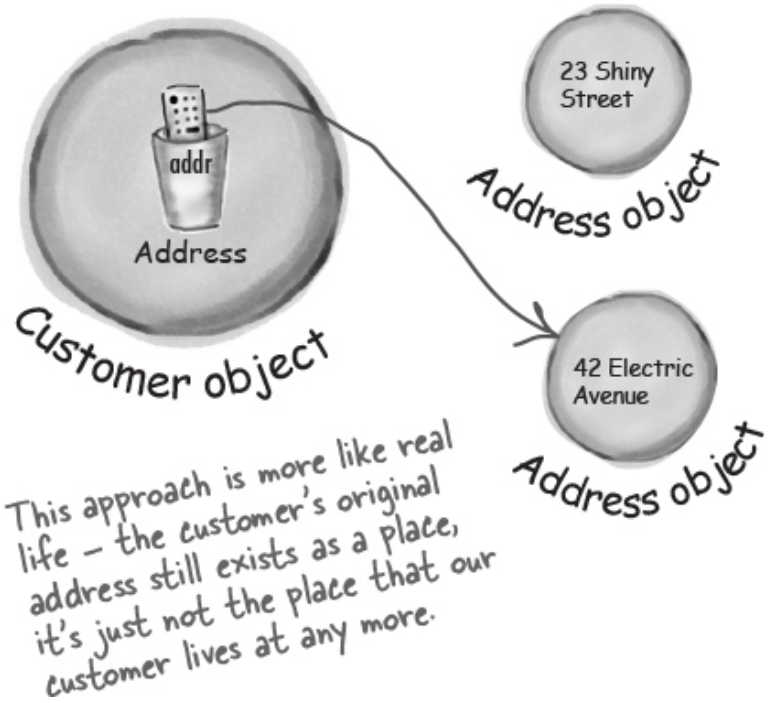
Imagine a system that has customers, and that each Customer object has an Address which represents the street address of a customer. If the customer's Address is an immutable object (all its fields are final and the data cannot be changed), how do you change the customer's address when they move?

1. ❶ The Customer has a reference to the original Address object containing the customer's street address data.




2. ❷ When the customer moves, a brand new Address object is created with the new street address for the customer


3. ❸ The Customer object's reference to their address is changed to point to the new Address object




### THERE ARE NO DUMB QUESTIONS

**Q: What happens if other parts of the program have a reference to the old Address?**

**A:** Actually sometimes we want this. Imagine the customer placed an order to be delivered to their original address. We still want the details of that order to have the original address, we don't want it changed to contain the details of the new address.

Once the customer changes their address (and the Customer contains a reference to the new address object), *then* we want new orders to use the new Address object.


Absolutely right. If your system has data that changes, those changes do have to happen somewhere. The key idea to take away from this discussion is that not all of the classes in your application have to have data that changes. In fact, we'd argue for minimizing the places where things change. Then, there are far fewer places where you have to think about what happens if multiple threads are making changes at the same time.

There are a number of techniques for working effectively with immutable data classes, we've just scratched the surface here. It is interesting to note that Java 16 introduced *records*, which are immutable data classes provided directly by the language.

NOTE

**Use immutable data classes where you can.**

**Limit the number of places where data can be changed by multiple threads.**

## More problems with shared data


We're nearly there, we promise! Just one last thing to look at.

So far we've seen all sorts of problems that can come from many threads writing to the same data. This applies to data in Collections too.

We can even have problems when we have lots of threads *reading* the same data, even if only one thread is making changes to it.

This code has just one thread writing to a collection, but two threads reading it.

```
public class ConcurrentReaders {
  public static void main(String[] args) {
    List<Chat> chatHistory = new ArrayList<>();
    ExecutorService executor = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 5; i++) {
      executor.execute(() -> chatHistory.add(new Chat("Hi there!")));
      executor.execute(() -> System.out.println(chatHistory));
      executor.execute(() -> System.out.println(chatHistory));
    }
    executor.shutdown();
  }
}
```

Stores the Chat objects in an ArrayList, which is NOT thread safe

Create a writing thread that adds to the List, and two threads that read from the list. Loop a few times to try to provoke the problem.

```
final class Chat {
  private final String message;
  private final LocalDateTime timestamp;

  public Chat(String message) {
    this.message = message;
    timestamp = LocalDateTime.now();
  }

  public String toString() {
    String time = timestamp.format(ofLocalizedTime(MEDIUM));
    return time + " " + message;
  }
}
```

Making an Object field "final" doesn't guarantee the data inside that object won't change, just that the reference won't change. String and LocalDateTime are immutable so this is safe.

Instances of Chat are immutable

# Reading from a changing data structure causes an Exception

Running the code on the last page causes an Exception to be thrown, sometimes. By now you know these sorts of issues depend a lot on the whims of the hardware, the operating system and the JVM.

```
File Edit Window Help PoorBrunonono
% java ConcurrentReaders

[]
[]
[18:43:59 Hi there!, 18:43:59 Hi there!]
[18:43:59 Hi there!, 18:43:59 Hi there!]
[18:43:59 Hi there!, 18:43:59 Hi there!, 18:43:59 Hi there!]
[18:43:59 Hi there!, 18:43:59 Hi there!, 18:43:59 Hi there!]
Exception in thread "pool-1-thread-2" Exception in thread "pool-1-thread-1" java.util.
ConcurrentModificationException
        at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1043)
        at java.base/java.util.ArrayList$Itr.next(ArrayList.java:997)
        at java.base/java.util.AbstractCollection.toString(AbstractCollection.java:472)
        at java.base/java.lang.String.valueOf(String.java:2951)
        at java.base/java.io.PrintStream.println(PrintStream.java:897)
        at ConcurrentReaders.lambda$main$2(ConcurrentReaders.java:17)
```

A ConcurrentModificationException is thrown by the reading thread when the List it is reading is changed WHILE this thread is reading it.

**NOTE**

**If a collection is changed by one thread while another thread is reading that collection, you can get a ConcurrentModificationException**

## Use a thread-safe data structure

Clearly good ol' ArrayList just isn't going to cut it if you have threads reading data that's being changed at the same time. Luckily for us, there are other options. We want a thread-safe data structure, one that can be written to, and read from, by multiple threads at the same time.

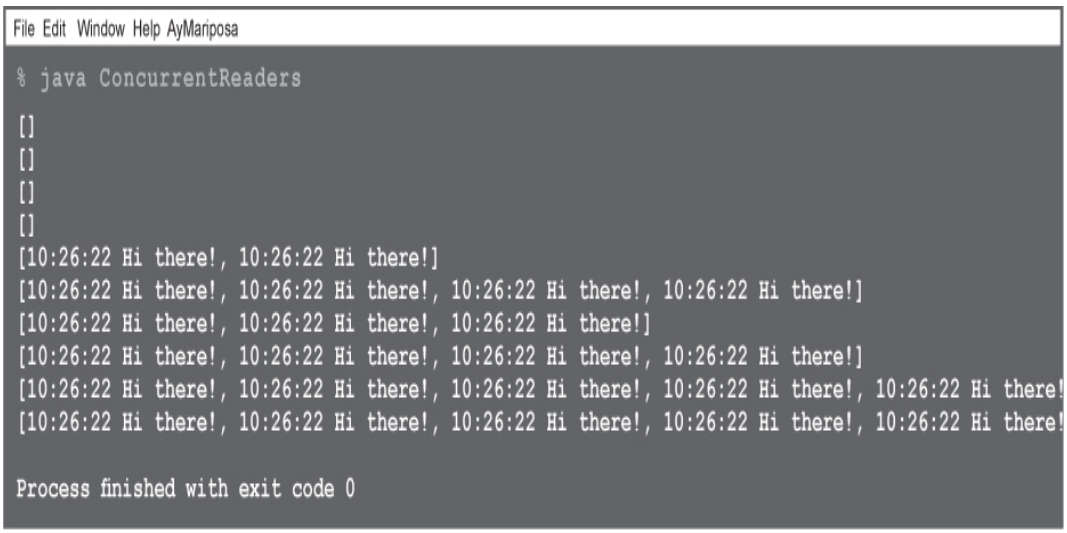
The java.util.concurrent package has a number of thread-safe data structures, and we're going to look at **`CopyOnWriteArrayList`** to solve this specific problem.

CopyOnWriteArrayList is a reasonable choice when you have a List that is being **read a lot, but not changed very often**. We'll see why later.

CopyOnWriteArrayList implements the List interface, so we can use it as a drop-in replacement for any List.

```
public class ConcurrentReaders {
  public static void main(String[] args) {
    List<Chat> chatHistory = new CopyOnWriteArrayList<>();
    ExecutorService executor = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 5; i++) {
      executor.execute(() -> chatHistory.add(new Chat("Hi there!")));
      executor.execute(() -> System.out.println(chatHistory));
      executor.execute(() -> System.out.println(chatHistory));
    }
    executor.shutdown();
  }
}
```

The rest of the code is exactly the same as before




No Exception!

## CopyOnWriteArrayList

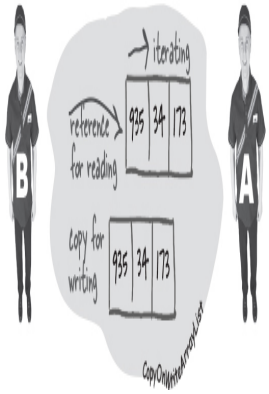
CopyOnWriteArrayList uses immutability to provide safe access for reading threads while other threads are writing.

How does it work? Well, it does what it says on the tin: when a thread is writing to the list, it's actually writing to a *copy* of the list. When the changes have been made, then the new copy replaces the original. In the meantime, any threads that were reading the list before the change are happily (and safely!) reading the original.

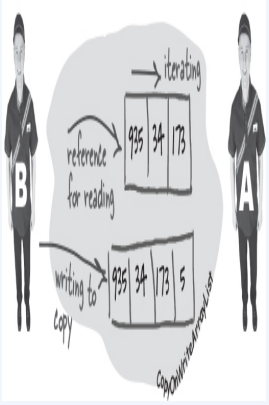
1 An instance of CopyOnWriteArrayList contains an ordered set of data, like an array.


2 When Thread A reads the CopyOnWriteArrayList, it gets an Iterator that allows it to read a **snapshot** of the list data at that point in time.

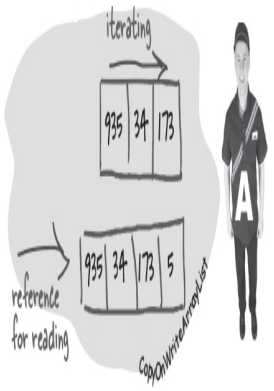

3 Thread B writes data to the CopyOnWrite-ArrayList by adding a new element, and the CopyOnWriteArray creates a **copy** of the list data before any changes are made. This is invisible to any of the reading or writing threads.




4 When Thread B makes changes to “the list”, it’s actually making changes to this copy. It’s happy knowing the changes are being made. The reading threads like Thread A are not affected at all, they’re iterating over the snapshot of the original data.




5 Once Thread B has finished its updates, then the original data is replaced with the new data.
If Thread A is still reading, it’s safely reading the old data. If any other threads start reading after the change, the get the new data.


iterating
935
34
173
935
34
173
5
reference
for reading
copyOnWriteArrayList
A

## THERE ARE NO DUMB QUESTIONS

**Q: Doesn't using CopyOnWriteArrayList mean that some reading threads will be reading old data?**

**A:** Yes, the reading threads will always be working off data that is a snapshot of the data from when they first started reading. This means that potentially the data might be out of date at some point, but at least it's not going to throw a ConcurrentModificationException.

**Q: Isn't it bad to be using out of date data?**

**A:** Not necessarily. In many systems this is "good enough". Think, for example, about a website that shows live news. Yes, you want it to be pretty up to date, but it doesn't have to be up to date to the latest millisecond, it's probably fine if some news is a few seconds old.

**Q: But I don't want my bank statement to be even slightly out of date! How can I make sure that critical shared data is always correct?**

**A:** CopyOnWriteArrayList is probably not the right choice if all threads need to be working off exactly the same data. Other data structures, like Vector, provide thread-safety by using locks to ensure only one thread at a time has access to the data. This is safe, but can be slow - you're not getting any benefits of multithreading if your threads need to wait their turn to get to their data.

**Q: So CopyOnWriteArrayList is a fast thread-safe data structure?**

**A:** Well... it depends! It's fast (compared to a locking Collection) if you have lots of reading threads and not many writes. But if there's a lot of writing going on, it might not be the best data structure. The cost of creating a new copy of the data every time a single write is made might be too high for some applications.

**Q: Why isn't there an easy answer to the best way to do this concurrency stuff?**

**A:** Concurrent programming is all about trade-offs. You need a good understanding of what your application is doing, how it should work, and the hardware and environment that's running it.

If you find yourself wondering which approach is better for your application, it's probably a good time to learn about performance testing, so you can measure exactly how each approach impacts the performance in your system.

**Q: You never told us about the case where adding final to a field declaration is not enough to make sure that value is never changed. What's the deal?**

**A:** Good catch! The "deal" is that if your field is a reference to another object, like a Collection or one of your own objects, using final does not prevent another thread from changing the values inside that object. The only way to make sure that doesn't happen is to make sure all your fields that are references only refer to immutable objects themselves. Otherwise, your immutable object can have data that changes.

See the LocalDateTime case on p538.

**NOTE**

**Thread-safe collections in early versions of Java were made safe via locking. For example, java.util. Vector.**

**Java 5 introduced concurrent data structures in java.util. concurrent. These do NOT use locking.**

## BULLET POINTS

- You can have serious problems with threads if two or more threads are trying to change the same data.
- 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 or use the synchronized keyword with a specified object.
- 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.
- The synchronized keyword isn't the only way to safeguard your data from multithreaded changes. Atomic variables, with CAS

operations, may be suitable if it's just one value which is being changed by multiple threads.

- It's *writing* data from multiple threads that causes the most problems, not *reading*, so consider if your data needs to be changed at all, or if it can be immutable.
- Make a class immutable by: making the class final, making all the fields final, setting the values just once in the constructor or field declaration; no setters or other methods that can change the data.
- Having immutable objects in your application doesn't mean nothing ever changes, it means that you limit the parts of your application where you have to worry about multiple threads changing the data.
- There are thread-safe data structures that let you have multiple threads reading the data while one (or more) threads change the data. Some of these are in java. util.concurrent.
- Concurrent programming is difficult! But there are plenty of tools to help you.

## Exercise

### BE the JVM

**NOTE**

**The Java file on this page represents a complete source file. Your job is to play JVM and determine what would be the output when the program runs?**

**How might you fix it, making sure the output is correct every time?**

```
import java.util.*;
import java.util.concurrent.*;

public class TwoThreadsWriting {
  public static void main(String[] args) {
    ExecutorService threadPool = Executors.newFixedThreadPool(2);
    Data data = new Data();
    threadPool.execute(() -> addLetterToData('a', data));
    threadPool.execute(() -> addLetterToData('A', data));
    threadPool.shutdown();
  }

  private static void addLetterToData(char letter, Data data) {
    for (int i = 0; i < 26; i++) {
      data.addLetter(letter++);
      try {
        Thread.sleep(50);
      } catch (InterruptedException ignored) {}
    }
```

```
    System.out.println(Thread.currentThread().getName() +
data.getLetters());
    System.out.println(Thread.currentThread().getName()
                          + " size = " +
data.getLetters().size());
  }
}

final class Data {
  private final List<String> letters = new ArrayList<>();

  public List<String> getLetters() {return letters;}

  public void addLetter(char letter) {
    letters.add(String.valueOf(letter));
  }
}
```

## Five-Minute Mystery

### 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.

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 and asked, "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?*

# Exercise Solutions

## BE the JVM

The answer is the output won't be the same every time. In theory, one might expect the size to always be 52 (2x 26 letters in the alphabet), but in fact this is one of those lost-update problems.

```
File Edit Window Help ZeCount
% java TwoThreadsWriting
pool-1-thread-2[a, A, b, B, c, C, d, D, E, F, g, G, h, H, i, j, K, k,
pool-1-thread-1[a, A, b, B, c, C, d, D, E, F, g, G, h, H, i, j, K, k,
pool-1-thread-1 size = 40
pool-1-thread-2 size = 40
```

**NOTE**

Example output. Your output probably won't look exactly the same as this, but if you predicted that the size would be less than 52, you win a cookie.

It can be solved in two different ways, both are valid.

### Synchronize the write method

```
public synchronized void addLetter(char letter) {
  letters.add(String.valueOf(letter));
}
```

**NOTE**

If this method is synchronized, only one thread at a time can write to the data, and therefore no updates will be lost. This will not work if there's a DIFFERENT thread reading at the same time as one of these threads are writing

### Use a thread-safe collection

```
private final List<String> letters = new CopyOnWriteArrayList<>
();
```

**NOTE**

Using CopyOnWriteArrayList will allow the threads to both safely write to the letters List

**NOTE**

**Use either solution, you do NOT have to do both!!**

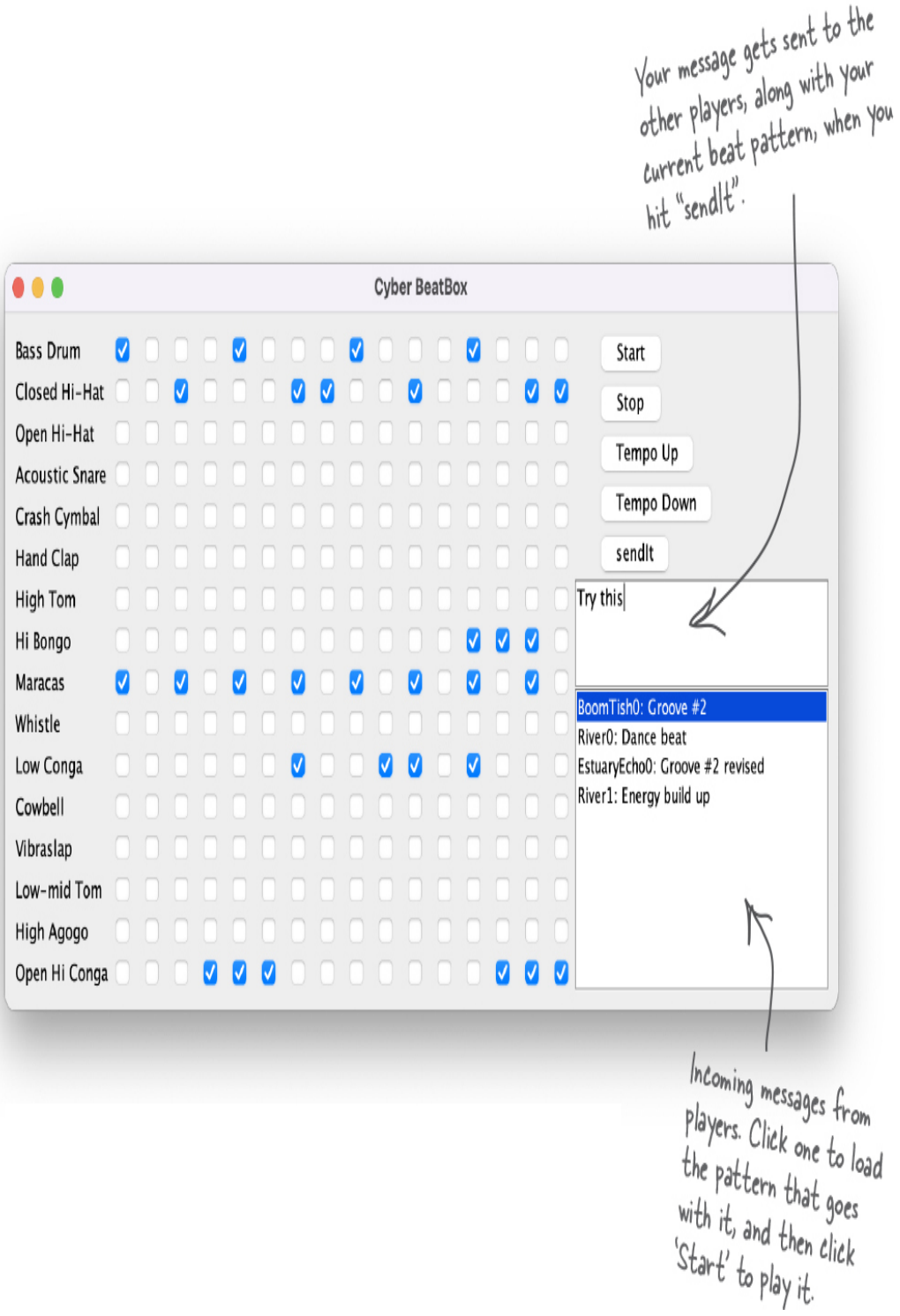
**With a thread-safe collection, you don't have to synchronize the writing method**

### 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.

# Appendix A. Final Code Kitchen


Your message gets sent to the other players, along with your current beat pattern, when you hit "sendIt".
Cyber BeatBox
Bass Drum
Closed Hi-Hat
Open Hi-Hat
Acoustic Snare
Crash Cymbal
Hand Clap
High Tom
Hi Bongo
Maracas
Whistle
Low Conga
Cowbell
Vibraslap
Low-mid Tom
High Agogo
Open Hi Conga
Start
Stop
Tempo Up
Tempo Down
sendIt
Try this
BoomTish0: Groove #2
River0: Dance beat
EstuaryEcho0: Groove #2 revised
River1: Energy build up
Incoming messages from players. Click one to load the pattern that goes with it, and then click 'Start' to play it.

NOTE

Finally, the complete version of the BeatBox!

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

# Final BeatBox client program

Most of this code is the same as the code from the Code Kitchens in the previous chapters, so we don't annotate the whole thing again. The new parts include:

GUI - two new components are added for the text area that displays incoming messages (actually a scrolling list) and the text field.

NETWORKING - just like the SimpleChatClient in this chapter, the BeatBox now connects to the server and gets an input and output stream.

MULTITHREADED - again, just like the SimpleChatClient, we start a 'reader' job that keeps looking for incoming messages from the server. But instead of just text, the messages coming in include TWO objects: the String message and the serialized array (the thing that holds the state of all the check boxes.)

All the code is available at https://oreil.ly/hfJava_3e_examples

```
import javax.sound.midi.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.io.*;
import java.net.Socket;
import java.util.*;
import java.util.concurrent.*;

import static javax.sound.midi.ShortMessage.*;

public class BeatBoxFinal {
  private JList<String> incomingList;
  private JTextArea userMessage;
  private ArrayList<JCheckBox> checkboxList;

  private Vector<String> listVector = new Vector<>();
  private HashMap<String, boolean[]> otherSeqsMap = new HashMap<>();

  private String userName;
  private int nextNum;

  private ObjectOutputStream out;
  private ObjectInputStream in;

  private Sequencer sequencer;
  private Sequence sequence;
  private Track track;

  String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat",
          "Open Hi-Hat", "Acoustic Snare", "Crash Cymbal", "Hand Clap",
          "High Tom", "Hi Bongo", "Maracas", "Whistle", "Low Conga",
          "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo",
          "Open Hi Conga"};
  int[] instruments = {35, 42, 46, 38, 49, 39, 50, 60, 70, 72, 64, 56, 58, 47, 67, 63};
```

These are the names of the instruments, as a String array, for building the GUI labels (on each row)

These represent the actual drum 'keys'. The drum channel is like a piano, except each 'key' on the piano is a different drum. So the number '35' is the key for the Bass drum, 42 is Closed Hi-Hat, etc.

```
public static void main(String[] args) {
  new BeatBoxFinal().startUp(args[0]);
}
```

Add a command-line argument for your screen name.

Example: % java BeatBoxFinal theFlash

```
public void startUp(String name) {
  userName = name;
  // open connection to the server
  try {
    Socket socket = new Socket("127.0.0.1", 4242);
    out = new ObjectOutputStream(socket.getOutputStream());
    in = new ObjectInputStream(socket.getInputStream());
    ExecutorService executor = Executors.newSingleThreadExecutor();
    executor.submit(new RemoteReader());
  } catch (Exception ex) {
    System.out.println("Couldn't connect-you'll have to play alone.");
  }
  setUpMidi();
  buildGUI();
}
```

Set up the networking, I/O, and make (and start) the reader thread. We're using Sockets here instead of Channels because they work better with Object Input/Output streams

You've seen this GUI code before, in Chapter 15

```
public void buildGUI() {
  JFrame frame = new JFrame("Cyber BeatBox");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  BorderLayout layout = new BorderLayout();
  JPanel background = new JPanel(layout);
  background.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

  Box buttonBox = new Box(BoxLayout.Y_AXIS);
  JButton start = new JButton("Start");
  start.addActionListener(e -> buildTrackAndStart());
  buttonBox.add(start);

  JButton stop = new JButton("Stop");
  stop.addActionListener(e -> sequencer.stop());
  buttonBox.add(stop);

  JButton upTempo = new JButton("Tempo Up");
  upTempo.addActionListener(e -> changeTempo(1.03f));
  buttonBox.add(upTempo);

  JButton downTempo = new JButton("Tempo Down");
  downTempo.addActionListener(e -> changeTempo(0.97f));
  buttonBox.add(downTempo);
```

An 'empty border' gives us a margin between the edges of the panel and where the components are placed. Purely aesthetic.

Lambda expressions call a specific method on this class when the button is pressed

The default tempo is 1.0, so we're adjusting +/- 3% per click.

```
JButton sendIt = new JButton("sendIt");
sendIt.addActionListener(e -> sendMessageAndTracks());
buttonBox.add(sendIt);
```

This is new, send the message and the current beat sequence to the music server

```
userMessage = new JTextArea();
userMessage.setLineWrap(true);
userMessage.setWrapStyleWord(true);
JScrollPane messageScroller = new JScrollPane(userMessage);
buttonBox.add(messageScroller);
```

Create a text area for the user to type their message

We saw JList briefly in Chapter 15. This is where the incoming messages are displayed. Only instead of a normal chat where you just LOOK at the messages, in this app you can SELECT a message from the list to load and play the attached beat pattern.

```
incomingList = new JList<>();
incomingList.addListSelectionListener(new MyListSelectionListener());
incomingList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane theList = new JScrollPane(incomingList);
buttonBox.add(theList);
incomingList.setListData(listVector);
```

This border on each instrument name helps them line up with the check boxes

```
  Box nameBox = new Box(BoxLayout.Y_AXIS);
  for (String instrumentName : instrumentNames) {
    JLabel instrumentLabel = new JLabel(instrumentName);
    instrumentLabel.setBorder(BorderFactory.createEmptyBorder(4, 1, 4, 1));
    nameBox.add(instrumentLabel);
  }

  background.add(BorderLayout.EAST, buttonBox);
  background.add(BorderLayout.WEST, nameBox);

  frame.getContentPane().add(background);
  GridLayout grid = new GridLayout(16, 16);
  grid.setVgap(1);
  grid.setHgap(2);

  JPanel mainPanel = new JPanel(grid);
  background.add(BorderLayout.CENTER, mainPanel);

  checkboxList = new ArrayList<>();
  for (int i = 0; i < 256; i++) {
    JCheckBox c = new JCheckBox();
    c.setSelected(false);
    checkboxList.add(c);
    mainPanel.add(c);
  }

  frame.setBounds(50, 50, 300, 300);
  frame.pack();
  frame.setVisible(true);
}

private void setUpMidi() {
  try {
    sequencer = MidiSystem.getSequencer();
    sequencer.open();
    sequence = new Sequence(Sequence.PPQ, 4);
    track = sequence.createTrack();
    sequencer.setTempoInBPM(120);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

This layout manager one lets you put the components in a grid with rows and columns

Make the check boxes, set them to 'false' (so they aren't checked) and add them to the ArrayList AND to the GUI panel.

Get the Sequencer, make a Sequence, and make a Track

```
private void buildTrackAndStart() {
  ArrayList<Integer> trackList; // this will hold the instruments for each
  sequence.deleteTrack(track);
  track = sequence.createTrack();
  for (int i = 0; i < 16; i++) {
    trackList = new ArrayList<>();
    int key = instruments[i];
    for (int j = 0; j < 16; j++) {
      JCheckBox jc = checkboxList.get(j + (16 * i));
      if (jc.isSelected()) {
        trackList.add(key);
      } else {
        trackList.add(null);  // because this slot should be empty in the track
      }
    }
    makeTracks(trackList);
    track.add(makeEvent(CONTROL_CHANGE, 1, 127, 0, 16));
  }
  track.add(makeEvent(PROGRAM_CHANGE, 9, 1, 0, 15)); // - so we always go to 16 beats
  try {
    sequencer.setSequence(sequence);
    sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
    sequencer.setTempoInBPM(120);
    sequencer.start();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Build a track by walking through the check boxes to get their state, and mapping that to an instrument (and making the MidiEvent for it). This is pretty complex, but it is EXACTLY as it was in the previous chapters, so refer to the Code Kitchen in Chapter 15 to get the full explanation again.

```
private void changeTempo(float tempoMultiplier) {
  float tempoFactor = sequencer.getTempoFactor();
  sequencer.setTempoFactor(tempoFactor * tempoMultiplier);
}
```

The Tempo Factor scales the sequencer's tempo by the factor provided, slowing the beat down or speeding it up.

```
private void sendMessageAndTracks() {
  boolean[] checkboxState = new boolean[256];
  for (int i = 0; i < 256; i++) {
    JCheckBox check = checkboxList.get(i);
    if (check.isSelected()) {
      checkboxState[i] = true;
    }
  }
  try {
    out.writeObject(userName + nextNum++ + ": " + userMessage.getText());
    out.writeObject(checkboxState);
  } catch (IOException e) {
    System.out.println("Terribly sorry. Could not send it to the server.");
    e.printStackTrace();
  }
  userMessage.setText("");
}
```

This is new... it's a lot like the SimpleChatClient, except instead of sending a String message, we serialize two objects (the String message and the beat pattern) and write those two objects to the socket output stream (to the server).