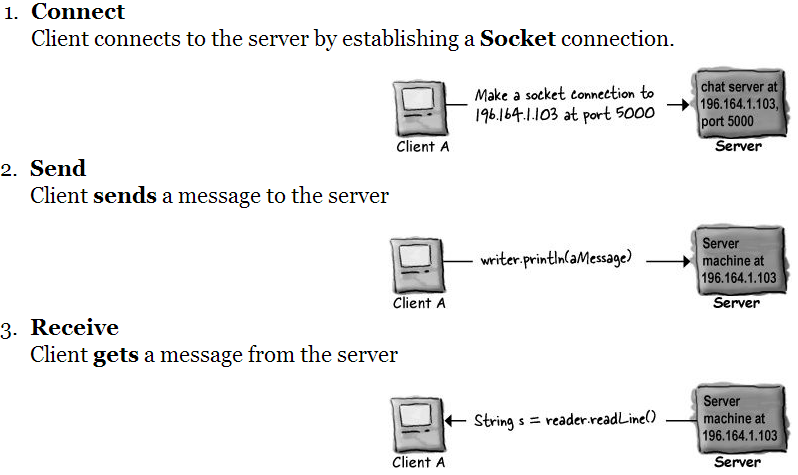
**B”H**

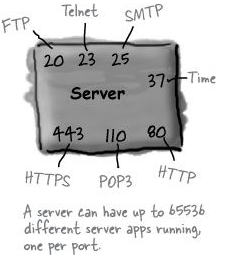
**Chapter 15**

**Networking and Threads**

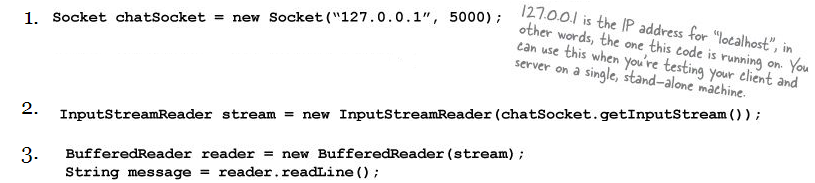
* Sending and receiving data over a network is just I/O with a slightly different connection stream at the end of the chain.



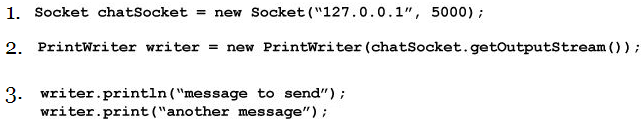
* A Socket (java.net.Socket class) is an object that represents a network connection between two machines.
  + A relationship between two machines, where two pieces of software know about each other and know how to communicate with each other.
* To make a Socket connection, you need to know two things about the server: IP address and TCP port number.
* A Socket connection means the two machines have information about each other, including network location (IP address) and TCP port.
* A TCP port is just a 16-bit number that identifies a specific program on the server.
* Some well-known TCP port numbers for common server applications:

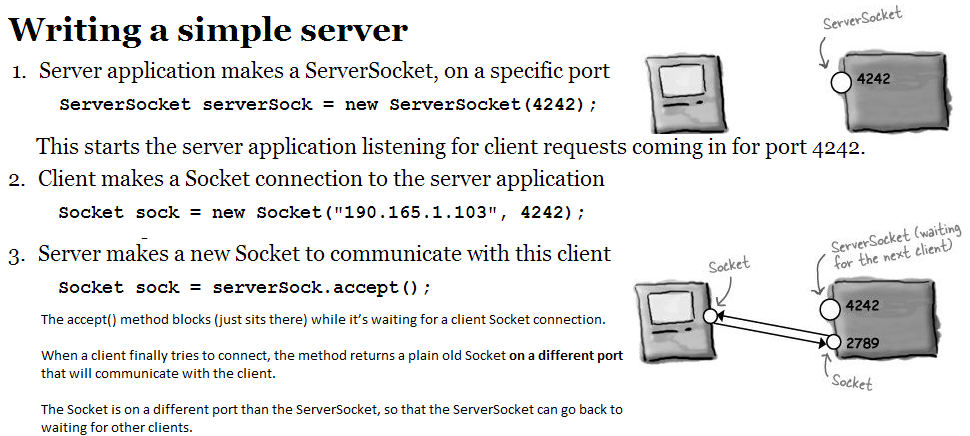


* The TCP port numbers from 0 to 1023 are reserved for well-known services. Don’t use them for your own server programs.
* The chat server we’re writing in the book uses port 5000. We just picked a number between 1024 and 65535.
* To bind a program to a port just means starting up a server application and telling it to run on a particular port. If you try to bind a program to a port that is already in use, you’ll get a BindException.
* One of the coolest features in Java is that most of your I/ O work won’t care what your high-level chain stream is actually connected to. In other words, you can use a BufferedReader just like you did when you were writing to a file, the difference is that the underlying connection stream is connected to a Socket rather than a File.
* To read data from a Socket:



* To write data to a Socket, use a PrintWriter We didn’t use PrintWriter in the last chapter, we used BufferedWriter. We have a choice here, but when you’re writing one String at a time, PrintWriter is the standard choice.

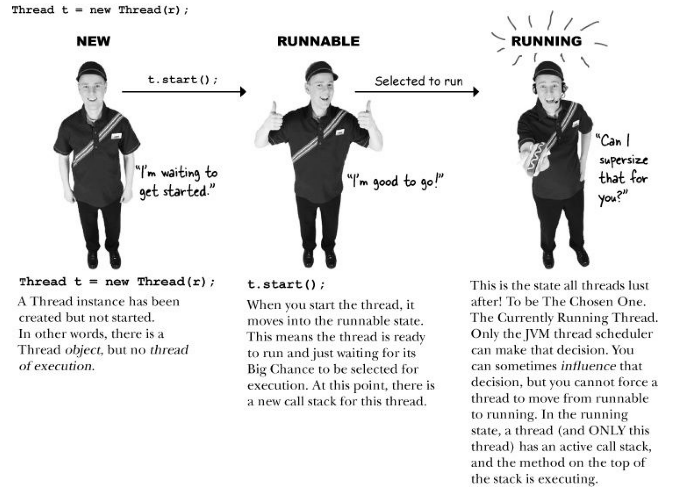




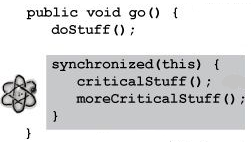
* Run java DailyAdviceServer from one command line terminal then run java DailyAdviceClient on another (or many other) command line terminals.
* Note the DailyAdviceServer has a VERY serious limitation — it can handle only one client at a time! It can’t accept a request from a client until it has finished with the current client and started the next iteration of the infinite loop.
* To make a server that can handle multiple clients concurrently give each new client Socket to a new thread.
* For the simple chat app we’re going to create in this chapter 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 need a new thread. A new, separate stack.
  + Unless you have multiple processors on your computer, each new Java thread is not actually a separate process running on the OS. But it almost feels as though it is.

**Multithreading in Java**

* By creating a new Thread object, you’ve launched a separate thread of execution, with its very own call stack.
* Multiple threading in Java means we have to look at both the thread and the job that’s run by the thread.
* 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 - an instance of class Thread. In other words, a separate call stack. When you see Thread, it refers to the Thread class in the java.lang package.
* 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).
* The Thread class has methods for starting a thread, joining one thread with another, and putting a thread to sleep. (It has more methods; these are just the crucial ones we need to use now).
* With more than one call stack, you get the appearance of having multiple things happen at the same time.
* In reality, only a true multiprocessor system can actually do more than one thing at a time, but with Java threads, it can appear that you’re doing several things simultaneously.
* In other words, execution can move back and forth between stacks so rapidly that you feel as though all stacks are executing at the same time.
* Remember, Java is just a process running on your underlying OS. So first, Java itself has to be ‘the currently executing process’ on the OS. But once Java gets its turn to execute, exactly what does the JVM run? 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.
* You’ll write a class that implements the Runnable interface, and that class is where you’ll define the work that a thread will perform.
* Runnable is to a Thread what a job is to a worker. A Runnable is the job a thread is supposed to run.
* The Runnable interface defines only one method, public void run().
* See and run MyRunnable.java
* The states of a new thread. Once the thread becomes runnable, it can move back and forth between runnable, running, and an additional state: temporarily not runnable (also known as ‘blocked’):

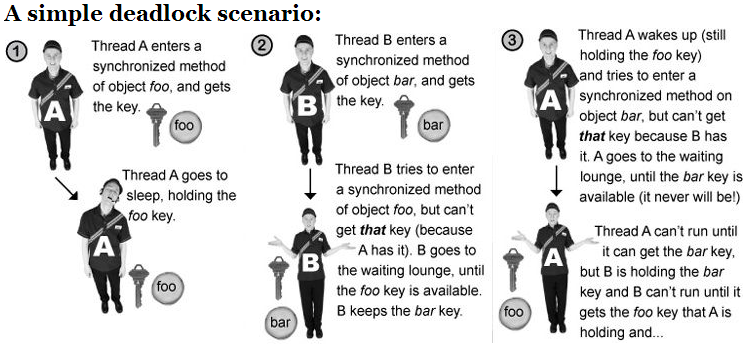


* The thread scheduler can move a running thread into a blocked state, for a variety of reasons. Examples:
  + The thread might be executing code to read from a Socket input stream, but there isn’t any data to read.
  + The executing code might have told the thread to put itself to sleep (sleep()).
  + The thread might be waiting because it tried to call a method on an object, and that object was ‘locked’.
* 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.
* 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.)
* **Do not base your program’s correctness on the scheduler working in a particular way!**
* Comment out the sleep portion in MyRunnable.java and run a few times – you’ll see that you can’t predict which output will show first.
* Putting a thread to sleep, even for a few milliseconds, forces the currently-running thread to leave the running state, thus giving another thread a chance to run. A sleeping thread will not become the currently-running thread before the length of its sleep time has expired.
* You can help influence turn-taking by putting your threads to sleep periodically. 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.
* You cannot reuse a Thread object? I.e. you cannot give it a new job to do and then restart it by calling start().
  + Once a thread’s run() method has completed, the thread can never be restarted.
  + At that point the thread moves into the dead state.
  + The Thread object might still be on the heap, as a living object that you can call other methods on (if appropriate), but the Thread object has permanently lost its ‘threadness’. In other words, there is no longer a separate call stack, and the Thread object is no longer a thread.
  + There are design patterns for making a pool of threads that you can keep using to perform different jobs. But you don’t do it by restarting() a dead thread.
* You can give your threads a name of your choosing, or you can accept their default names. The cool thing about names is that you can use them to tell which thread is running.
* See and run RunThreads.java - it starts two threads; each thread has the same job: run in a loop, printing the currently-running thread’s name with each iteration.
* Concurrency issues lead to race conditions. Race conditions lead to data corruption.
* 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.
* See and run RyanAndMonicaJob.java – that example uses bank account transactions as an example of race condition issues and how to remedy it. To demonstrate the "overdrawn" error remove the "synchronized" modifier.
  + We need the makeWithdrawal( ) method to run as one atomic thing.
  + We need to make sure that once a thread enters the makeWithdrawal() method, it must be allowed to finish the method before any other thread can enter.
  + **Use the synchronized keyword to modify a method so that only one thread at a time can access it.**
  + That’s how you protect the bank account! You don’t put a lock on the bank account itself; you lock the method that does the banking transaction.
* Every object has a lock.
* Most of the time, the lock is unlocked, and you can imagine a virtual key sitting with it.
* Object locks come into play only when there are synchronized methods.
* When an object has one or more synchronized methods, a thread can enter a synchronized method only if the thread can get the key to the object’s lock!
* **The locks are not per method, they are per object.**
  + If an object has two synchronized methods, it does not simply mean that you can’t have two threads entering the same method. It means you can’t have two threads entering any of the synchronized methods.
  + 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.
  + When a thread hits a synchronized method, it 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. The thread won’t give up the key until it completes the synchronized method. So while that thread is holding the key, no other threads can enter any of that object’s synchronized methods, because the one key for that object won’t be available.
* Don’t synchronize everything, just to be thread-safe. Because:
  + A synchronized method has a certain amount of overhead; there’s going to be a performance hit
  + A synchronized method can slow your program down because synchronization restricts concurrency.
  + Most frightening, synchronized methods can lead to deadlock (see below).
  + 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:



**Deadlocks**

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



* Run java VerySimpleChatServer in one terminal then run java SimpleChatClient on multiple other terminals.
  + Examine this code well – it creates a simple working chat app
* You can use synchronization on static methods that change the static variable state.
  + 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.
* Thread priorities are numerical values that tell the scheduler (if it cares) how important a thread is to you. But there is no guarantee. We recommend that you use priorities only if you want to influence performance, but never, ever rely on them for program correctness.