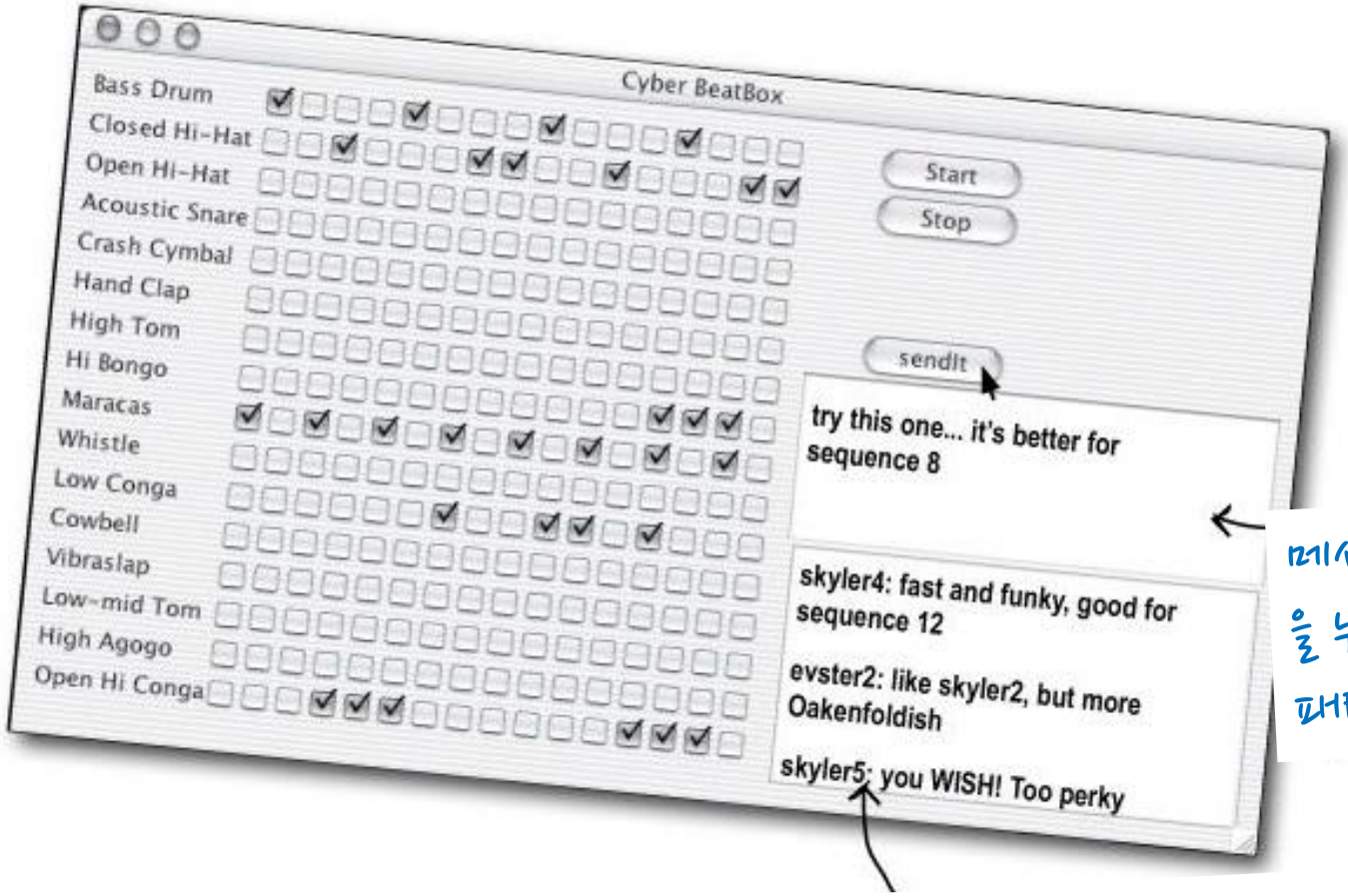
Networking and Threads: Make a Connection [1]

Samkeun Kim <skim@hknu.ac.kr>

http://cyber.hankyong.ac.kr

Real-time Beat Box Chat

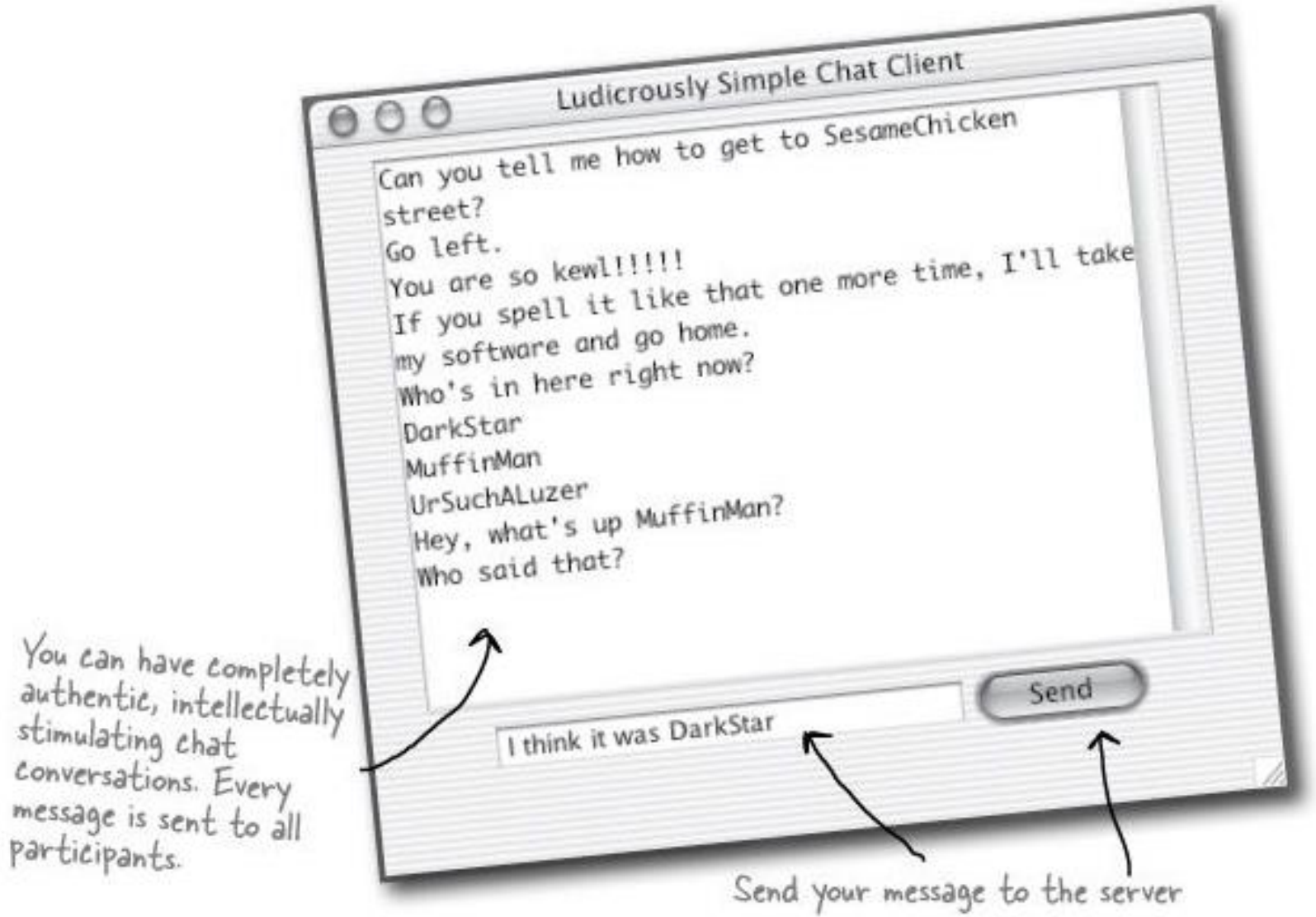


메시지를 입력하고 sendIt 버튼을 누르면 메시지와 현재 비트 패턴을 보낼 수 있다.

받은 메시지를 클릭하면 그 메시지와 함께 전달되어온 비트 패턴을 불러올 수 있다.

Chat server에 대해서도 조금 알아야 한다.

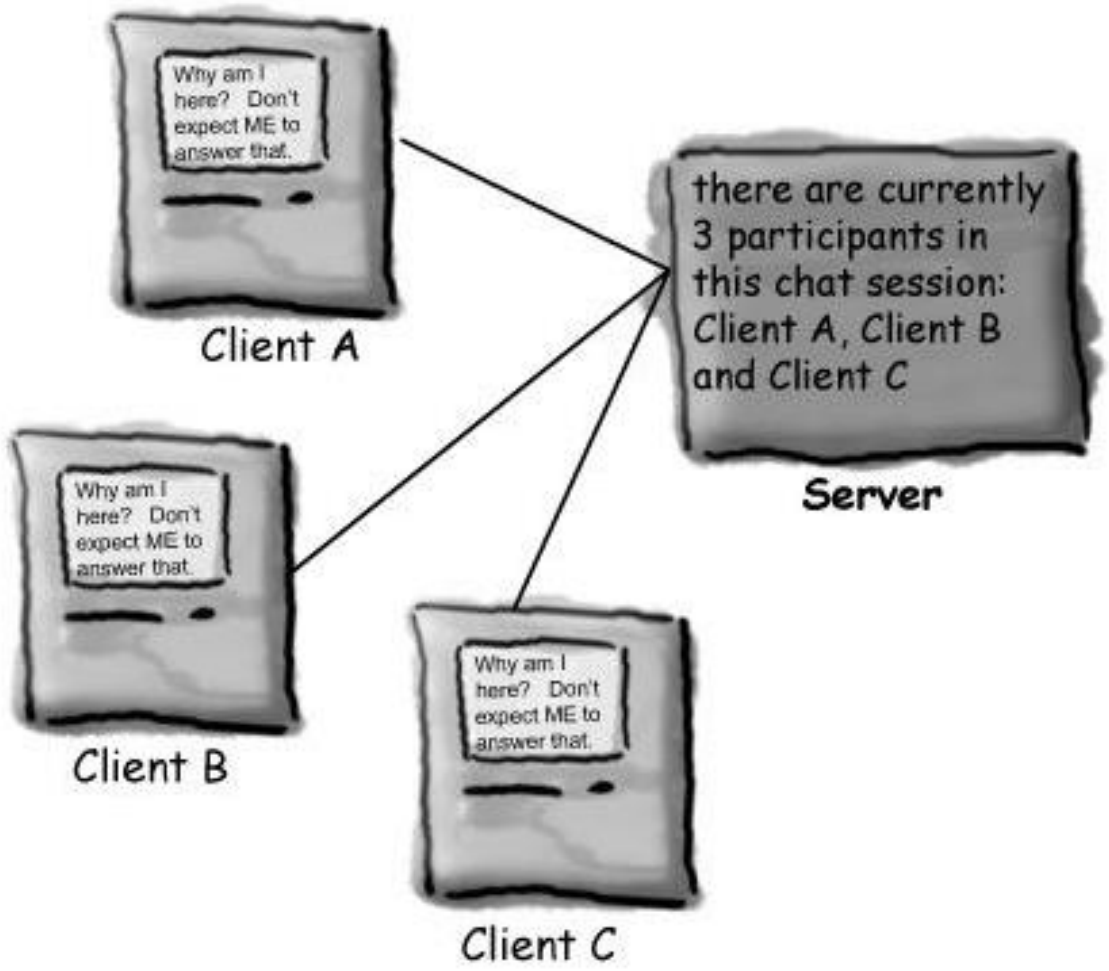
Real-time Beat Box Chat



Chat Program Overview

클라이언트는 서버에 대해 알아야 한다.

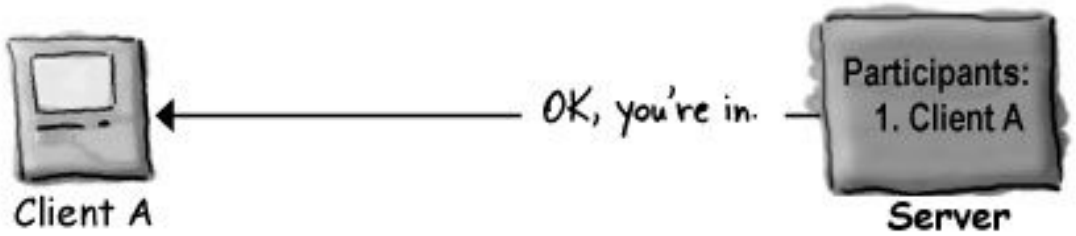
서버는 모든 클라이언트에 대해 알아야 한다.



How it works:

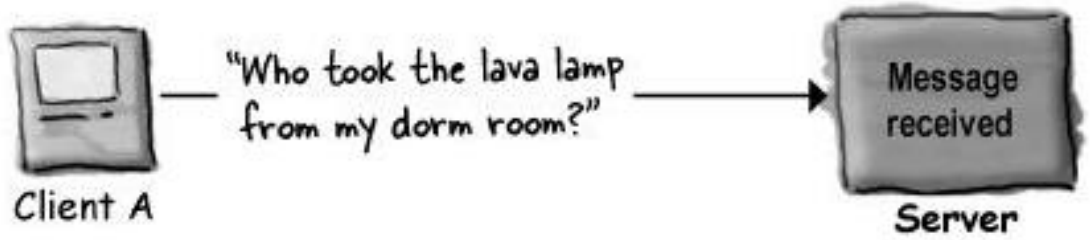
1. 클라이언트가 서버에 접속한다

2. 서버는 접속을 허용하고 그 클라이언트를 참가자 명단에 추가한다

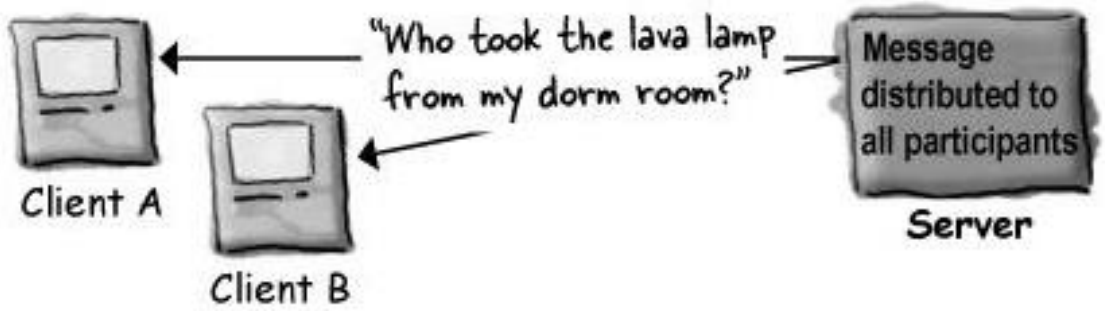


3. 다른 클라이언트가 서버에 접속한다

4. 클라이언트 A가 채팅 서비스에 메시지를 보낸다



5. 서버는 그 메시지를 **모든 참가자**에게 보낸다 (including the original sender)



Connecting, Sending, and Receiving

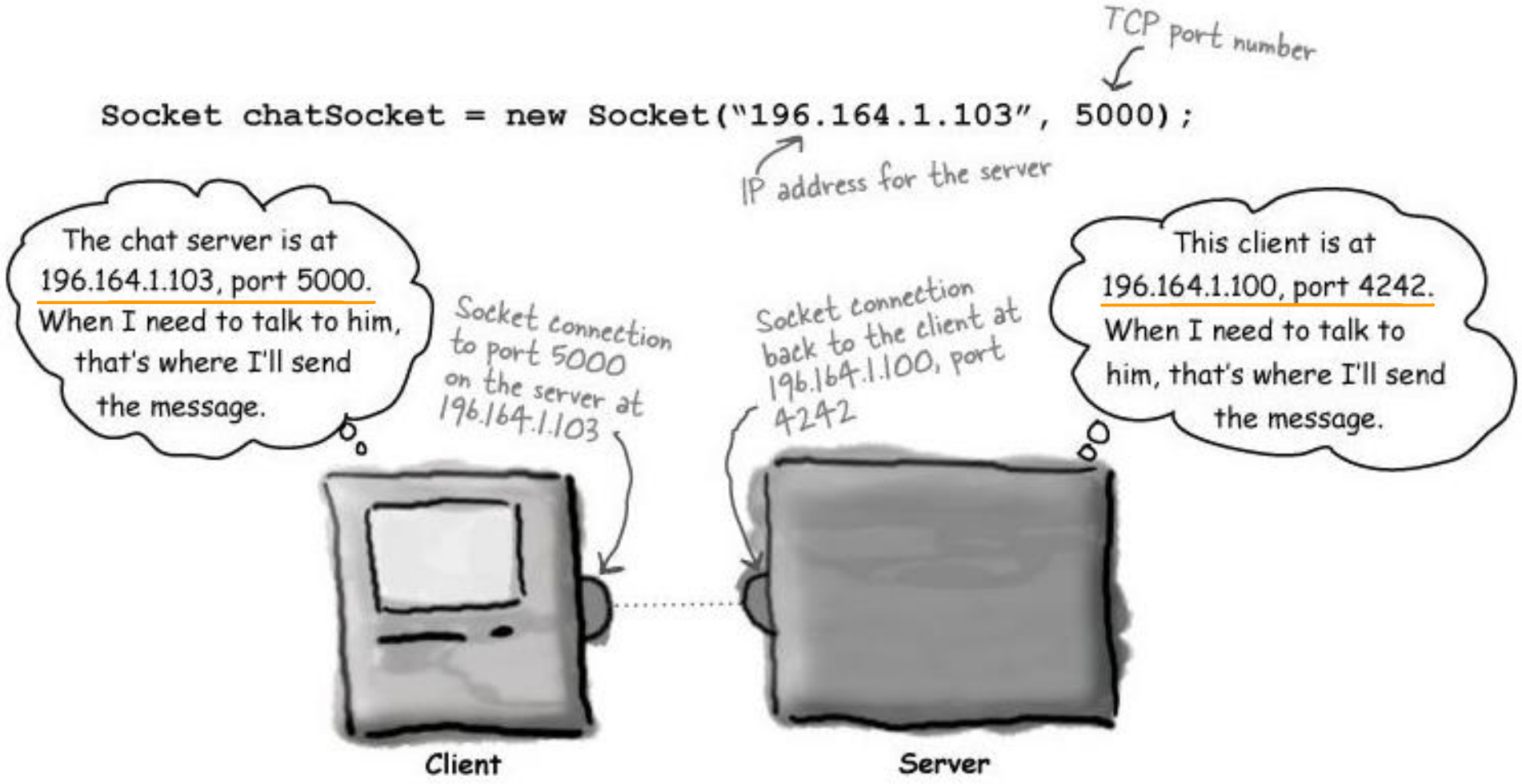
1. **Connect**: 클라이언트가 소켓 연결을 통하여 서버에 접속한다

2. **Send**: 클라이언트가 서버에 메시지를 보낸다

3. **Receive**: 클라이언트가 서버로부터 메시지를 받는다

Make a network Socket connection

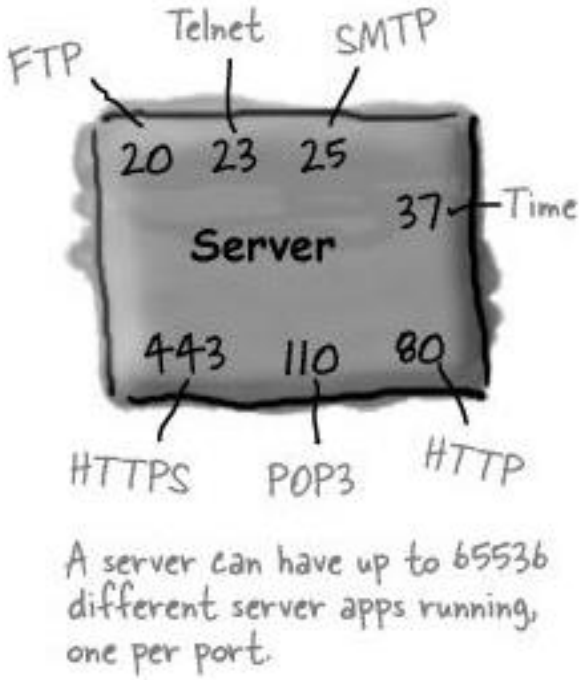
소켓 연결을 위해서 서버에 관해 알아야 할 두 가지: IP address 와 TCP port number



A TCP port is just a number

TCP 포트: 서버에 있는 특정 프로그램을 구분해주는 16비트 숫자

<< Well-known TCP port numbers for common server applications >>



포트번호 0 ~ 1,023번까지는 미리 정해진 서비스를 위해 예약되어 있다.

1024 ~ 65535 사이의 포트번호를 선택할 수 있다.

IP address is the mall

IP address is like specifying a particular shopping mall, say, "Flatirons Marketplace"

Port number is the specific store in the mall

Port number is like naming a specific store, say, "Bob's CD Shop"

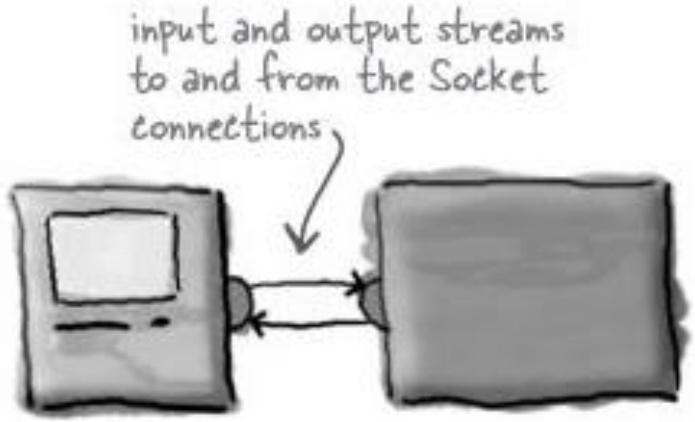
소켓으로부터 데이터 읽기: BufferedReader 사용

Socket 연결을 통해서 데이터를 주고받을 때는 스트림을 사용한다.

Java I/O의 매력중의 하나는 고수준 체인 스트림이 실제로 어디에 연결되어 있는지 신경쓰지 않아도 된다는 점이다.

단순히 파일에 쓰기 작업을 했듯이 **BufferedReader**를 사용하면 된다.

다만 파일이 아니라 **Socket**에 연결된 것이 다를 뿐!



1. 서버에 소켓 연결을 만든다

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

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

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

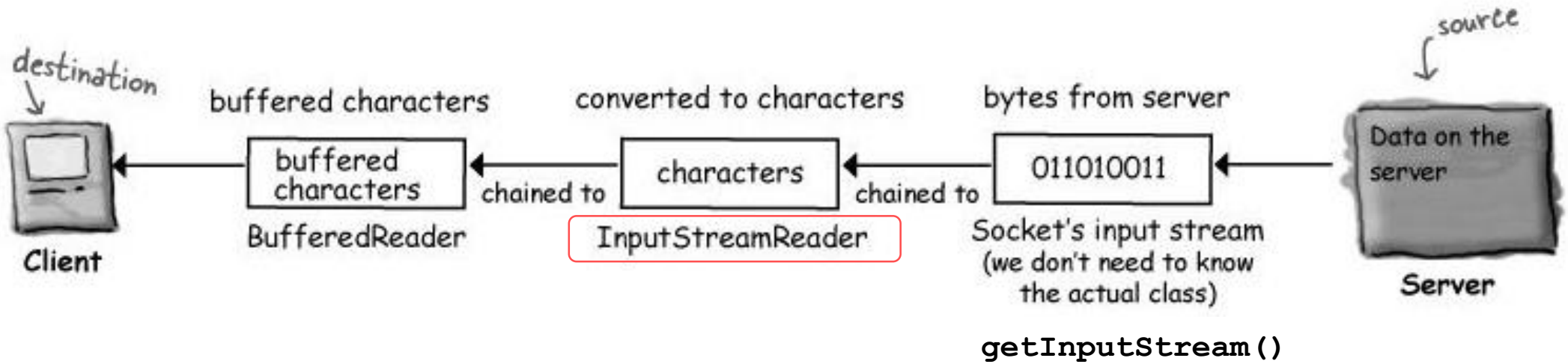
2. Socket 저수준 연결 스트림에 연결된 InputStreamReader를 만든다.
   (InputStreamReader는 저수준 Socket과 고수준 BufferedReader의 브릿지 역할을 한다.)

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

3. BufferedReader를 만들어서 읽어 들인다!

InputStreamReader를 BufferedReader에 체인으로 연결한다. (InputStreamReader는 소켓으로부터 얻은 저수준 연결 스트림으로부터 체인으로 연결된 것이다)

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



데이터를 Socket에 쓰기: PrintWriter

1. 서버에 Socket 연결을 만든다

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

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

2. Socket 저수준 연결 스트림에 고수준 체인스트림인 PrintWriter를 만든다

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

PrinterWriter는 소켓의 저수준 출력 스트림으로부터 얻어 온 문자 데이터와 바이트 데이터 사이에 브릿지로서 동작한다. PrinterWriter를 소켓의 출력 스트림에 체인으로 연결함으로서 소켓 연결에 스트링을 writing할 수 있다.

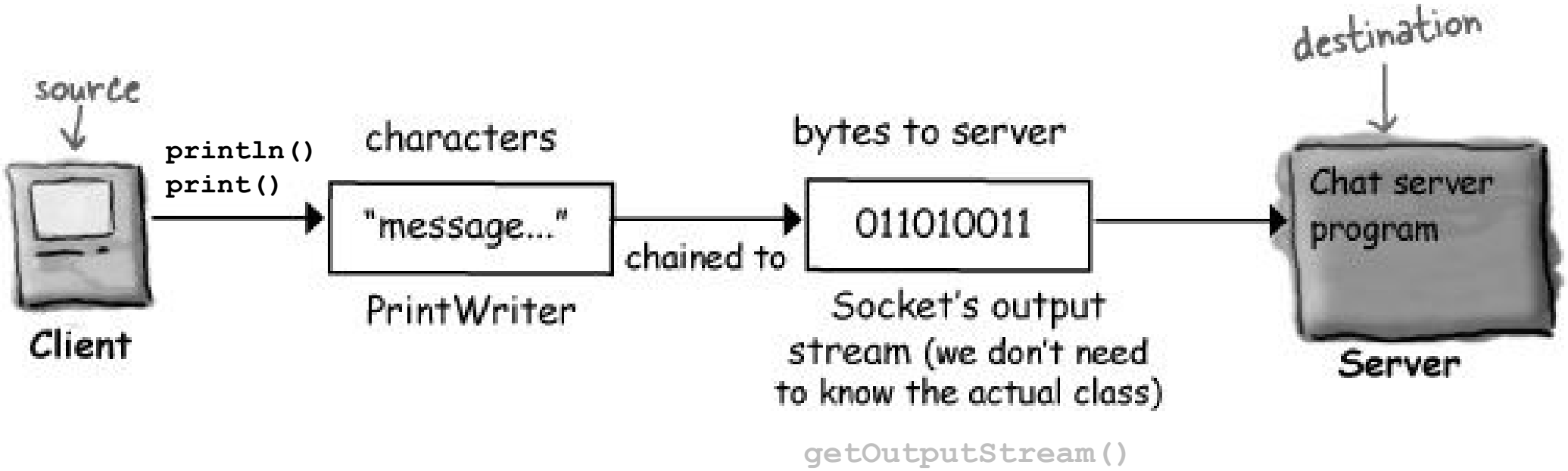
Socket은 저수준 연결 스트림을 주고 우리는 PrinterWriter 생성자에게 그 연결 스트림을 제공함으로서 PrinterWriter에 체인으로 연결한다.

3. 뭔가를 쓴다(write/print)

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

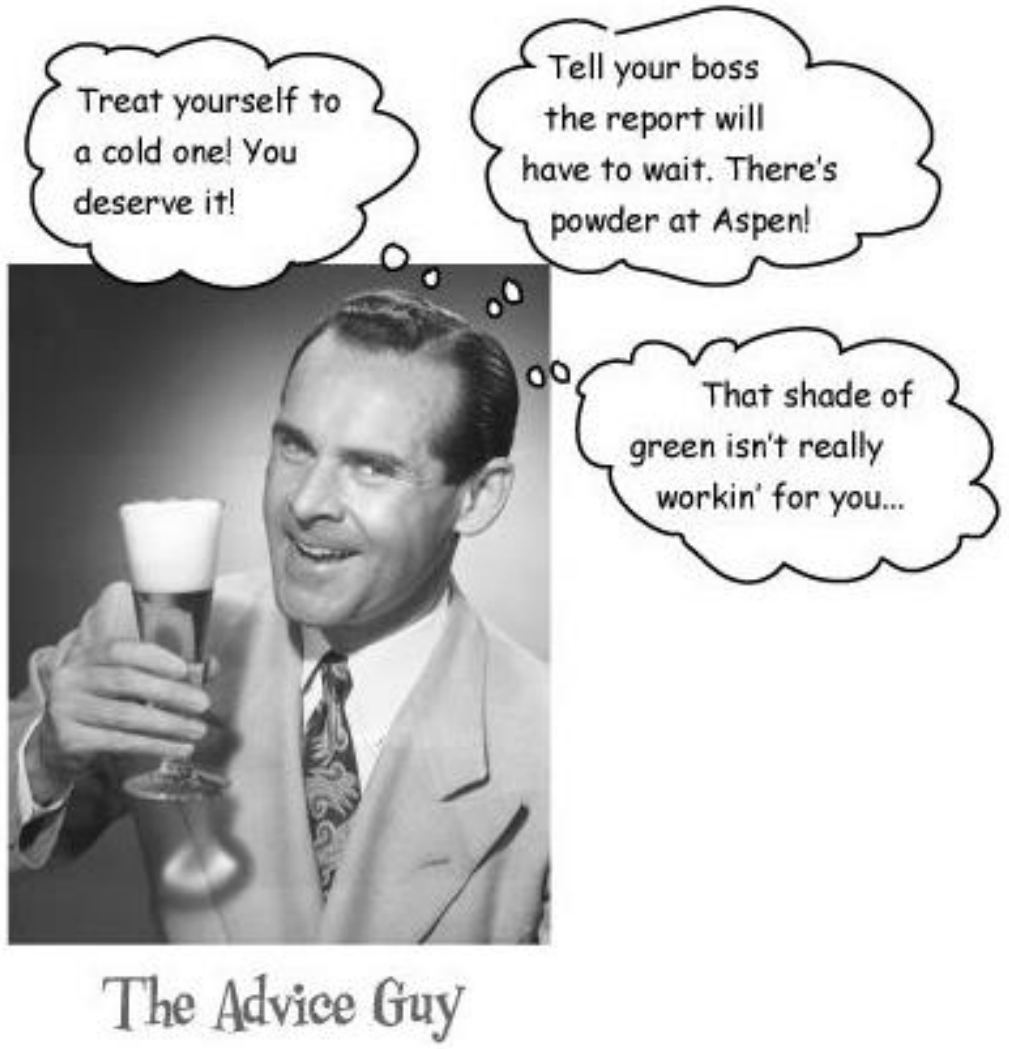
println() adds a new line at the end of what it sends.

print() doesn't add the new line.



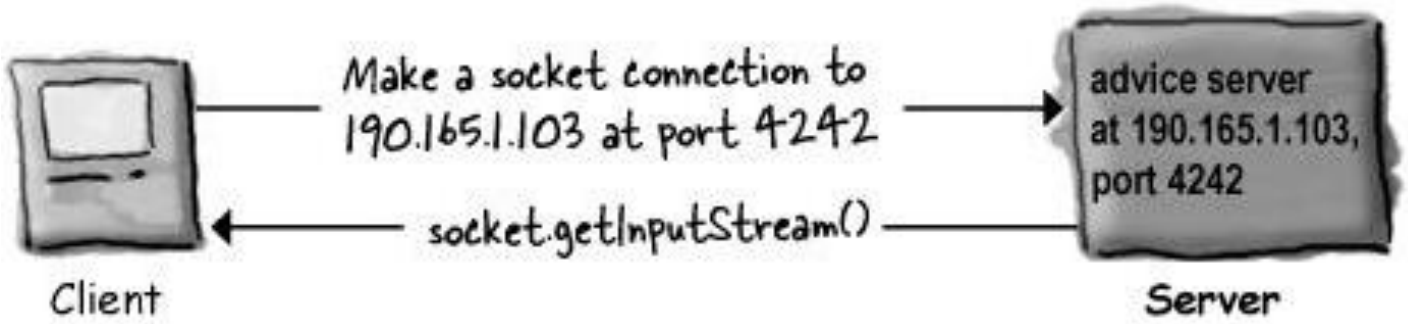
The DailyAdviceClient

채팅 애플리케이션을 만들기 전에
조금 간단한 것부터 시작해보자.

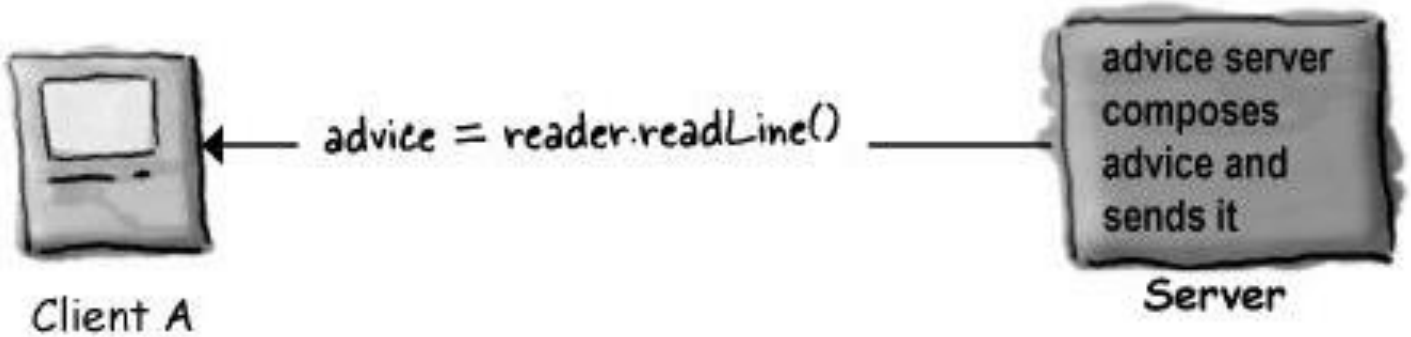


조언 맨 프로그램에 대한 클라이언트 프로그램부터 먼저 만들어보자.

1. **Connect**
   클라이언트가 서버에 연결하여 서버로부터 입력 스트림을 받는다.



2. **Read**
   클라이언트가 서버로부터 받은 메시지를 읽는다.



DailyAdviceClient code

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

public class DailyAdviceClient {

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

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

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

            reader.close();

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

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

class Socket is in java.net

a lot can go wrong here

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

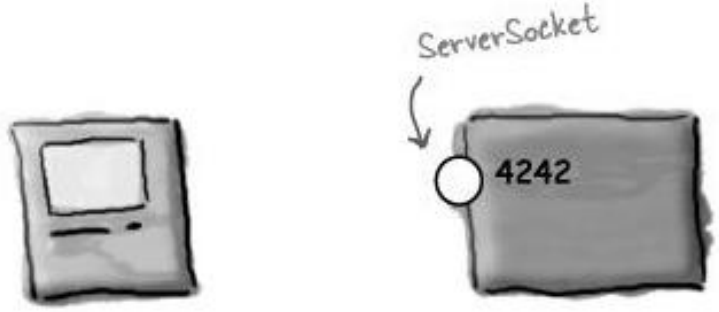
chain a BufferedReader to an InputStreamReader to the input stream from the Socket.

this closes ALL the streams

readLine()은 FILE에 체인으로 연결된 BufferedReader를 사용하는 것과 정확하게 같다. 다르게 말하면, BufferedReader 메소드를 호출할 때 reader는 문자들이 어디에서 오는지 모를뿐만 아니라 신경쓰지도 않는다.

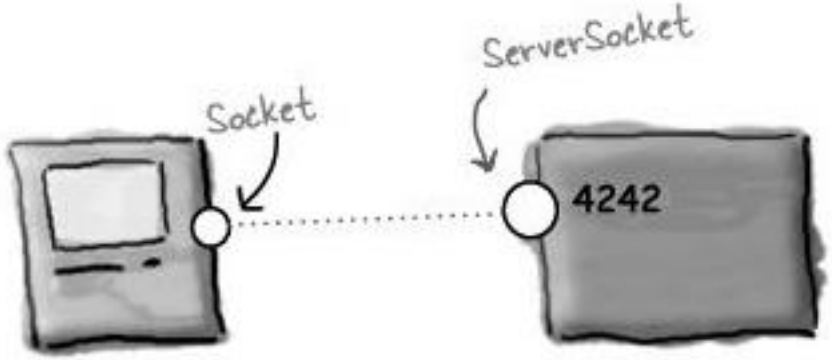
Writing a simple server

1. 서버 애플리케이션에서 특정 포트에 대한 **ServerSocket**을 만든다.
   `ServerSocket serverSock = new ServerSocket(4242);`



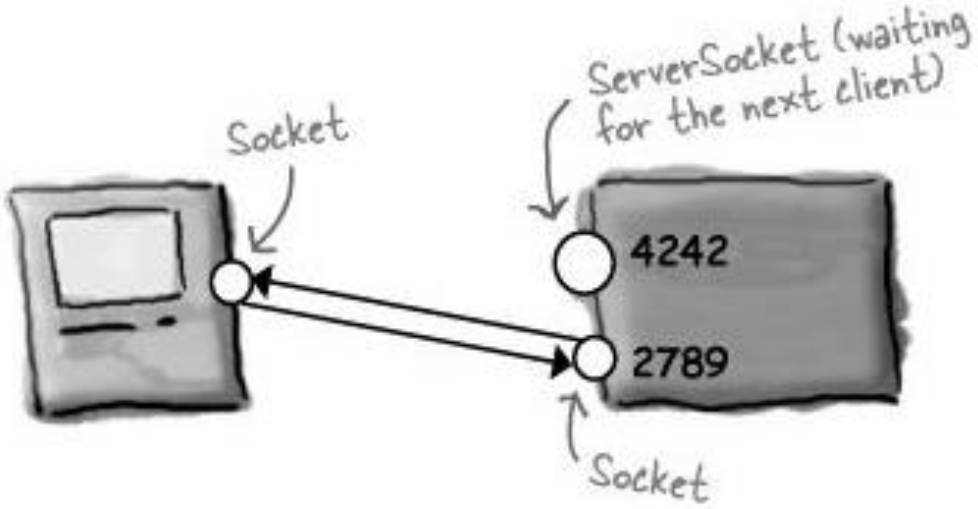
   서버 애플리케이션은 4242번 포트로 들어오는 클라이언트 요청을 리스닝하기 시작한다.

2. 클라이언트가 서버 애플리케이션에 소켓 연결을 만든다.
   `Socket sock = new Socket("190.165.1.103", 4242);`



3. 서버에서는 새로운 소켓을 만들어 클라이언트와 통신한다.

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



accept() 메소드는 요청이 들어올 때까지 블록되어 있다가 요청이 들어오면 클라이언트와 통신하기 위한 임의의 **Socket**을 반환해준다.

DailyAdviceServer code

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

public class DailyAdviceServer {

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

   public void go() {

      try {
         ServerSocket serverSock = new ServerSocket(4242);

         while(true) {

            Socket sock = serverSock.accept();

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

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

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

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

remember the imports

daily advice comes from this array

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

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

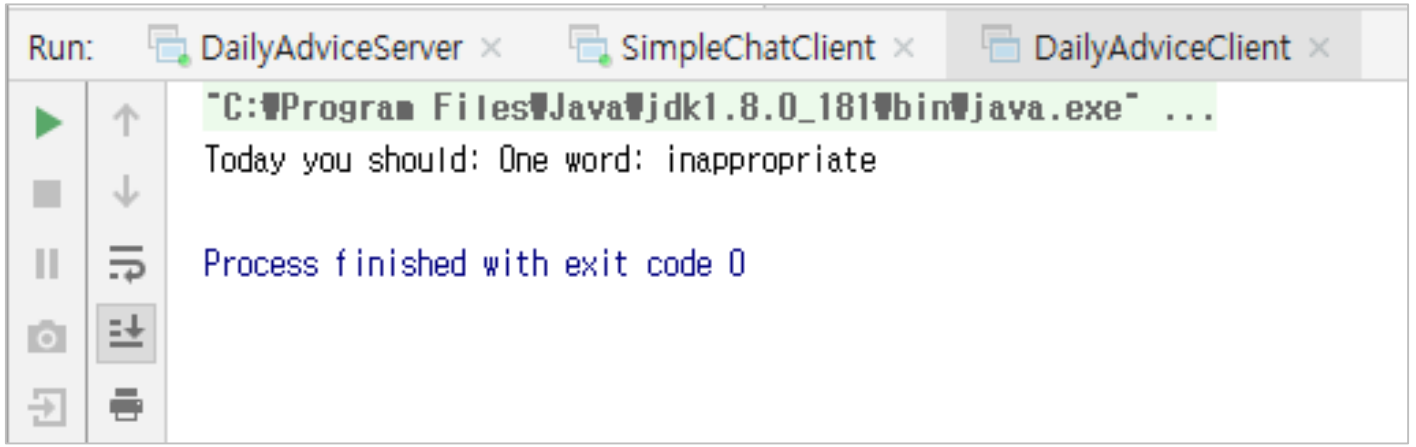
ServerSocket은 이 서버 어플리케이션을 하나의 포트 4242에서 클라이언트 요청을 'listen' 하도록 한다.

accept 메소드는 요청이 들어올 때까지 블록되어 있다가 요청이 들어오면 클라이언트와 통신하기 위한 임의의 Socket을 반환해준다.

이제 클라이언트에 대한 Socket 연결을 이용하여 PrintWriter 객체를 만들어 println()에 String 메시지를 보낸다.

실습과제 15-1

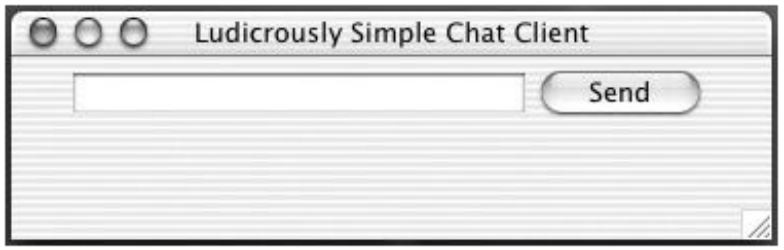
DailyAdviceClient.java / DailyAdviceServer.java 실행



Writing a Chat Client

Version One: send-only

Code outline




```
public class SimpleChatClientA {

   JTextField outgoing;
   PrintWriter writer;
   Socket sock;

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

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

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

} // close outer class
```

Version One: send-only

SimpleChatClientA.java

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

public class SimpleChatClientA {

    JTextField outgoing;
    PrintWriter writer;
    Socket sock;

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

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

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

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

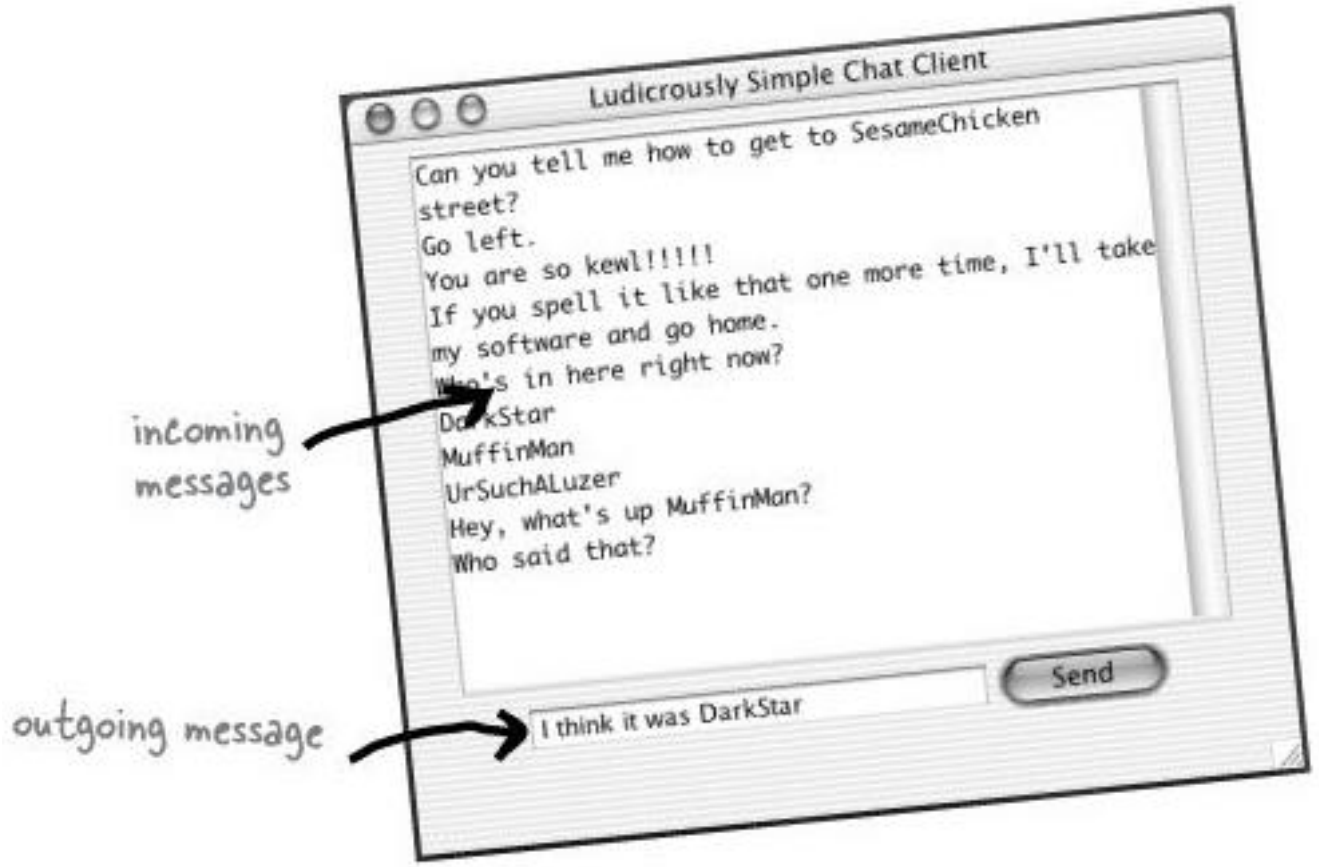
we're using localhost so you can test the client and server on one machine

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

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

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

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

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

Version Two: send and receive



NOTE

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

HOW do you get messages from the server?

BufferedReader를 통하여

WHEN do you get messages from the server?

옵션 1 : 20초마다 서버를 확인해본다.

옵션 2 : 사용자가 메시지를 보낼 때마다 서버로부터 메시지를 받아온다.

옵션 3 : **서버로부터 메시지가 보내지자마자 메시지를 읽는다.**

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

Multithreading in Java

우리는 **옵션 3**을 사용할거라는걸 이미 알고 있다!

사용자의 GUI 작업을 전혀 방해하지 않고서도 서버로부터 오는 메시지를 체크할 수 있다.

즉 장막 뒤에서 서버로부터 오는 메시지를 읽어 들이는 작업을 계속할 수 있다.

Java는 언어 자체에 **multithreading** 기능이 내장되어 있다.

새로운 실행 쓰레드(thread of execution)를 만들어 보자:

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

새로운 Thread 객체를 생성하게 되면 독립된 **Call Stack**이 생성되어 그곳에 실행 thread가 들어가게 된다.

한 가지 문제점이 있다!!

- ✓ 방금 생성한 thread는 아무것도 하지 않기 때문에 만들어지자마자 죽어버린다.
- ✓ thread가 죽으면 새로 만들어진 스택도 사라지고, 결국 모든 일이 허사가 되어버린다!
- ✓ 중요한 요소가 하나 빠져있기 때문이다. 즉 **쓰레드가 해야 할 작업** (thread's job)이 없다.
- ✓ 별도로 만들어진 thread는 실행시킬 코드가 필요하다.

이것은 **thread와 그 thread에 의해 수행시킬 작업 모두가 필요하다**는 것을 의미한다.

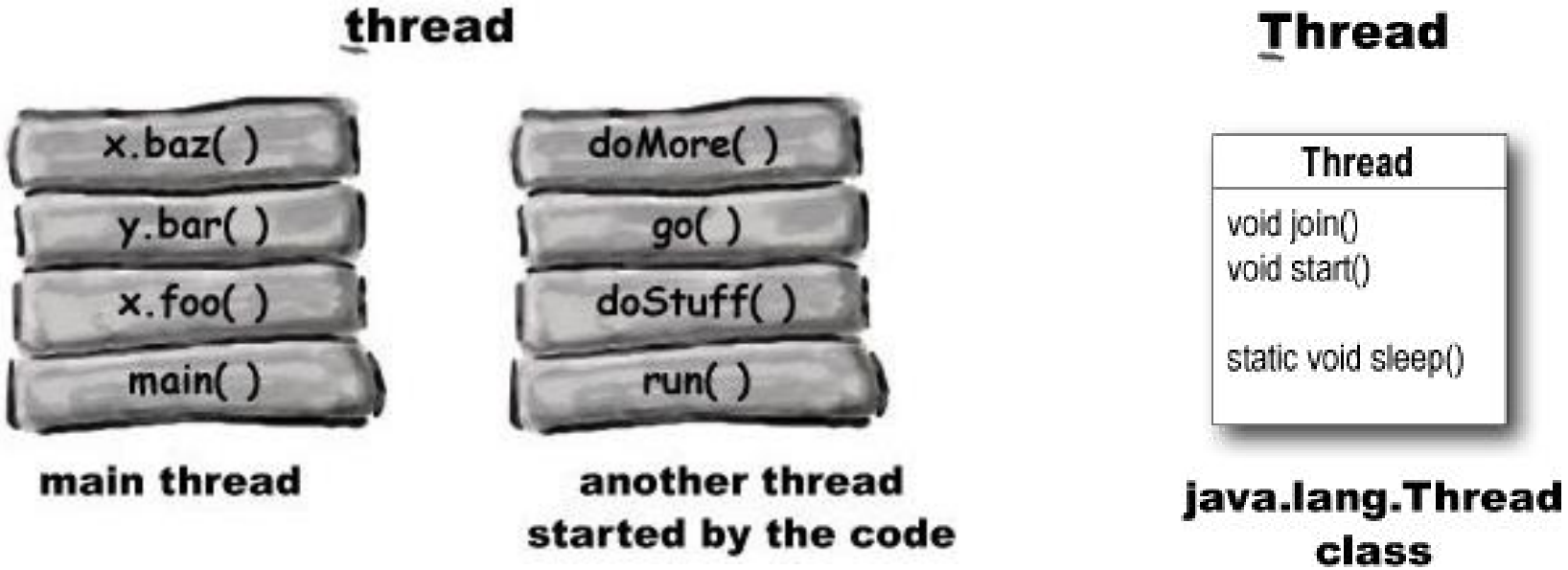
Java has multiple threads but ONLY ONE Thread class

소문자 't'를 가지는 thread와 대문자 'T'를 가지는 Thread는 엄연히 그 의미가 다르다.

thread => 독립된 실행 쓰레드

Thread => 클래스 (자바에서 클래스는 대문자로 시작)

- ✓ Thread는 java.lang 패키지의 클래스이다.
- ✓ Thread 객체는 실행 쓰레드를 의미한다.
- ✓ 새로운 실행 쓰레드를 시작시키고 싶을 때마다 Thread 클래스의 인스턴스를 생성하면 된다.



콜 스택이 2개 이상 있다는 의미는?

콜 스택이 2개 이상 있다면

- ✓ multiprocessor system => 실제 한번에 1개 이상의 일을 할 수 있다
- ✓ Java thread => 실제 프로세서는 하나지만 몇 가지 일을 동시에 하는 것처럼 보이게 할 수 있다!

Java thread

- ✓ 스택 사이를 매우 빠르게 왔다갔다하면서 실행 => 마치 모든 스택이 동시에 실행되는 것처럼 느끼게 된다.
- ✓ JVM은 그것이 무엇이던 간에 현재 실행중인 스택의 탑에 있는 것을 실행
  즉 100밀리 초마다 서로 다른 스택의 서로 다른 메소드 사이를 오가며 코드를 실행하게 된다!

thread가 해야만 하는 일 중의 하나는 현재 thread의 스택에서 어느 메소드가 실행되고 있는지를 추적하는 것이다.

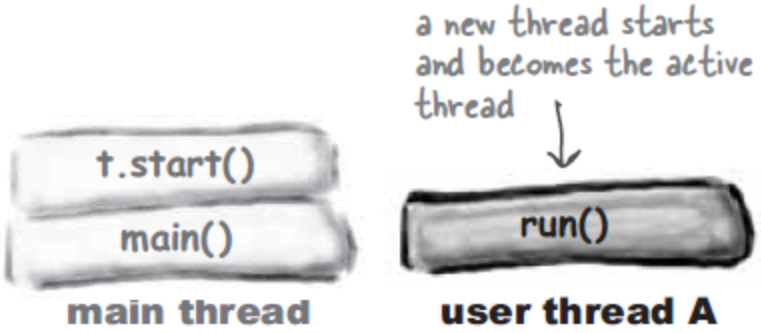
1 The JVM calls the main() method.

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

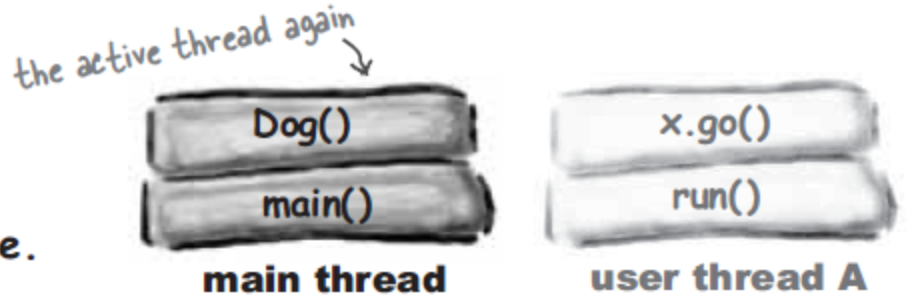
2 main() starts a new thread. The main thread is temporarily frozen while the new thread starts running.

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

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



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



How to launch a new thread:

1. Runnable 객체(thread의 작업)를 생성한다.

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

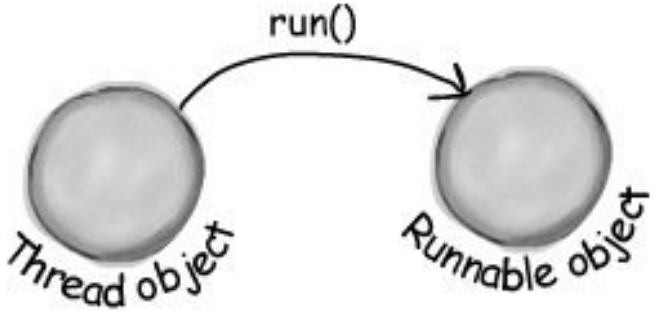
2. Thread 객체(일꾼)를 만들어서 Runnable 객체(작업)를 전달한다.

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

새로운 Runnable 객체를 Thread 생성자에게 전달한다.

3. Thread 객체를 시작시킨다.

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



Thread의 start() 메소드가 호출될 때까지는 아무 일도 일어나지 않는다.

모든 쓰레드는 작업이 필요하다: 새로운 쓰레드 스택에 넣을 메소드!

Runnable 객체와 **Thread** 객체와의 관계는 작업(job)과 일꾼(worker)과의 관계와 같다.

Runnable 객체에 쓰레드가 실행시켜야 할 작업이 들어있다.

Runnable 객체에는 새로운 쓰레드의 스택 맨 밑에 들어가게 되는 **run()** 메소드가 들어있다.

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

thread가 할 일을 만들려면 Runnable 인터페이스를 구현하라

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

```
public class MyRunnable implements Runnable {

    public void run() {
        go();
    }

    public void go() {
        doMore();
    }

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

2

Runnable은 구현해야 할 메소드가 단 하나 밖에 없다: public void run(). 쓰레드가 실행해야 할 JOB을 두는 곳이다. 새로운 스택의 맨 바닥으로 들어가게 된다.

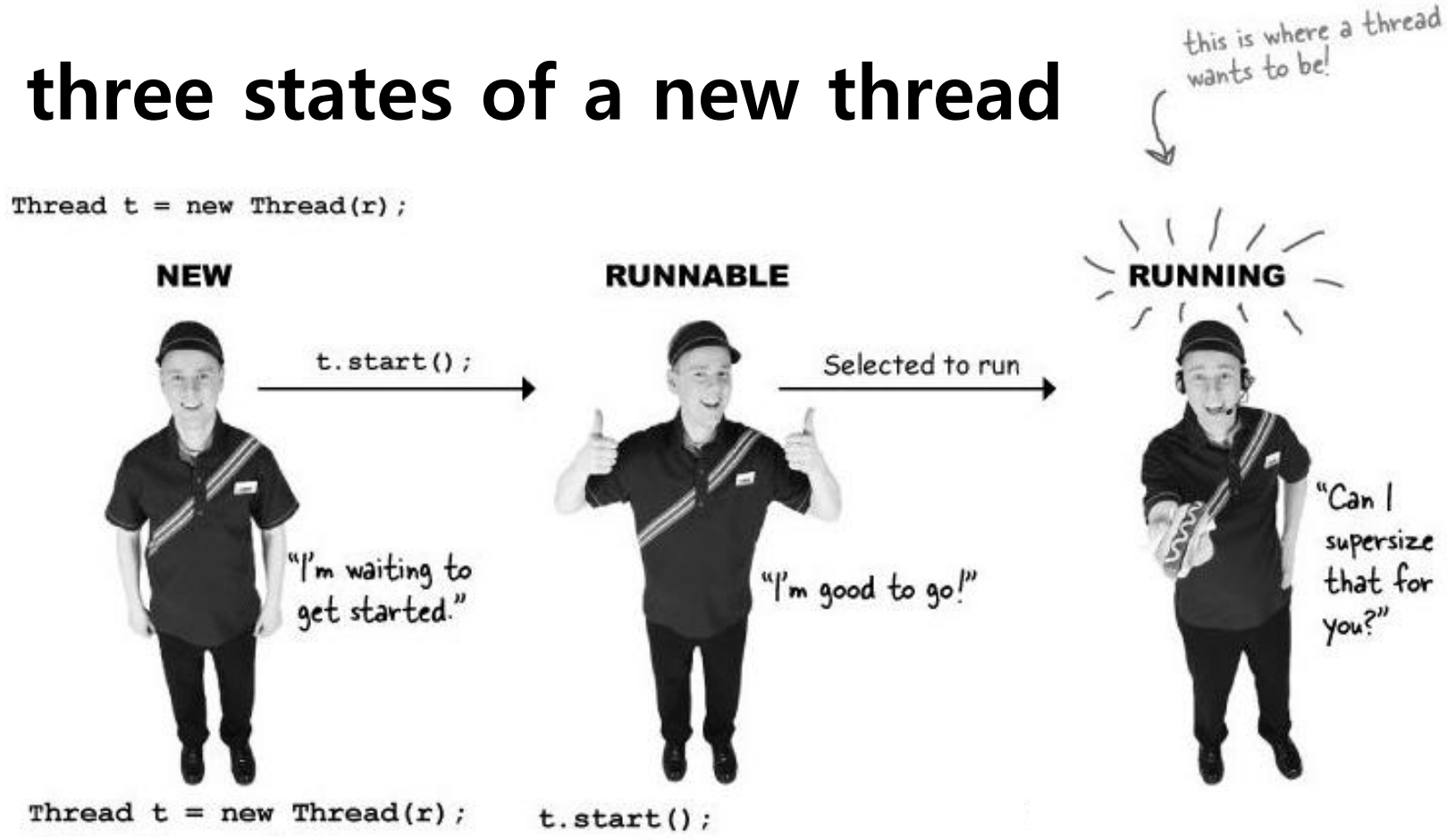

```
class ThreadTester {

    public static void main (String[] args) {

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

        myThread .start();

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

새로운 Runnable 인스턴스를 새로운 Thread 생성자에게 전달해준다. 이것은 새로운 스택의 바닥에 어느 메소드를 두어야 하는 지를 말해준다. 즉, 새로운 쓰레드가 실행할 첫 번째 메소드.

1

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'.

The three states of a new thread



Thread 인스턴스가 생성되었지만 시작되지는 않았다. 즉 Thread 객체는 있지만 실행 쓰레드가 없다.

쓰레드를 실행시키면 runnable 상태로 바뀐다. 이것은 쓰레드가 실행될 준비가 되어 있음을 의미한다. 이 시점에서 이 쓰레드를 위한 새로운 콜 스택이 있다.

실행할 쓰레드 선택은 오로지 JVM 쓰레드 스케줄러가 결정한다!

하지만 이게 전부가 아니다!!

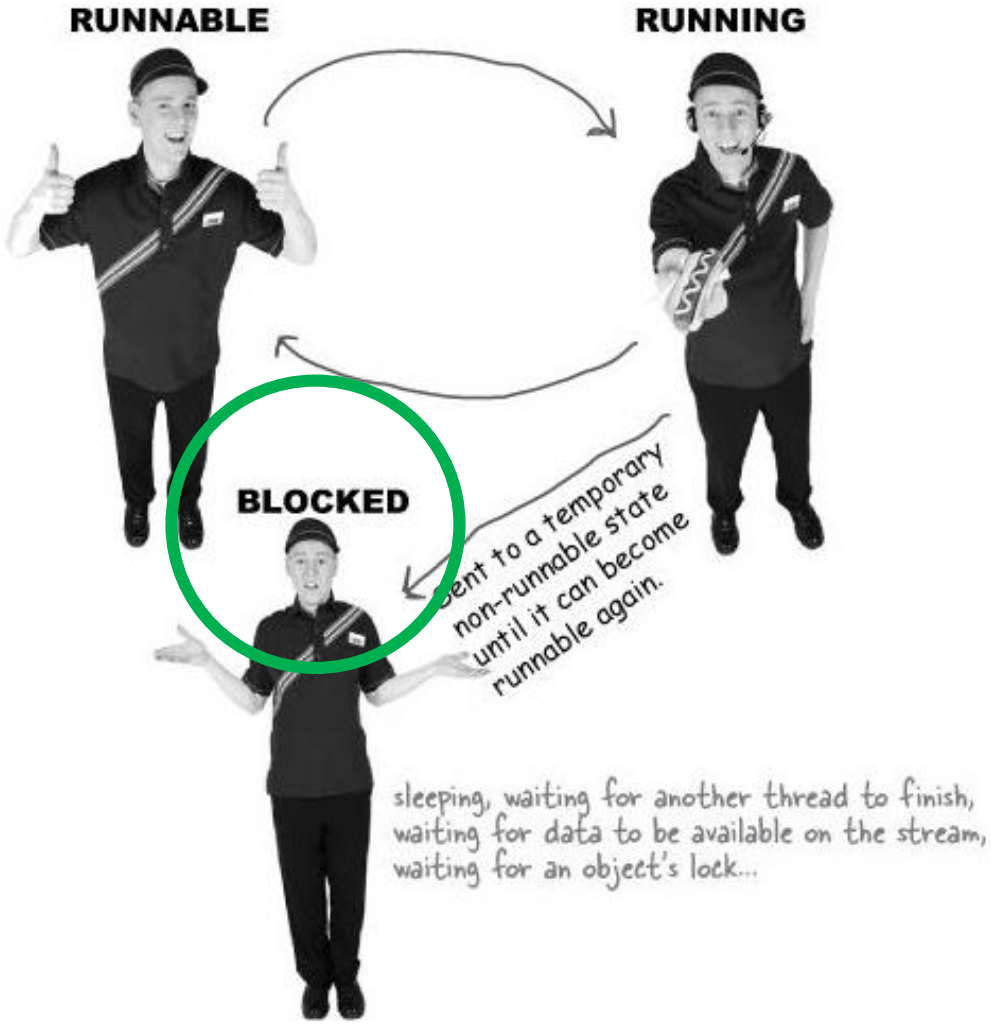
Typical runnable/running loop

thread는 전형적으로 runnable과 running 상태를 왔다갔다한다.

즉 JVM 쓰레드 스케줄러가 실행시킬 쓰레드를 하나 선택하고 다음 번에는 다른 쓰레드에게 기회를 주는 과정을 반복한다.

A thread can be made temporarily not-runnable

쓰레드 스케줄러가 여러 가지 이유로 인해 실행중인 쓰레드를 블록 상태로 전환시킬 수도 있다.



Thread Scheduler

쓰레드 스케줄러가 어떤 특별한 방식으로 작동하리라고 예상하고 프로그램을 만들면 안된다!

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

스케줄러의 불확실성을 보여줄 수 있는 예제

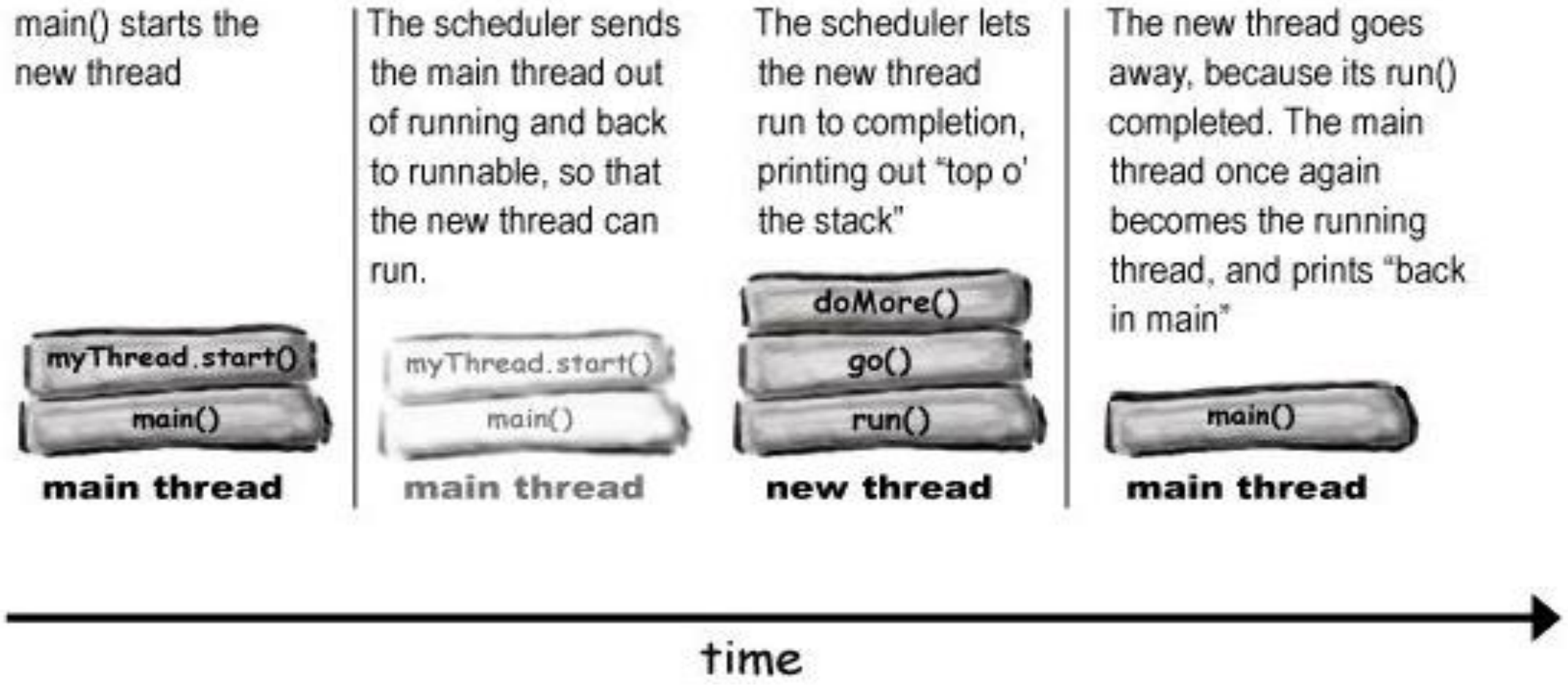
```
public class MyRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {
      doMore();
   }

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


class ThreadTestDrive {

   public static void main (String[] args) {

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

      myThread.start();

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

Produced this output:

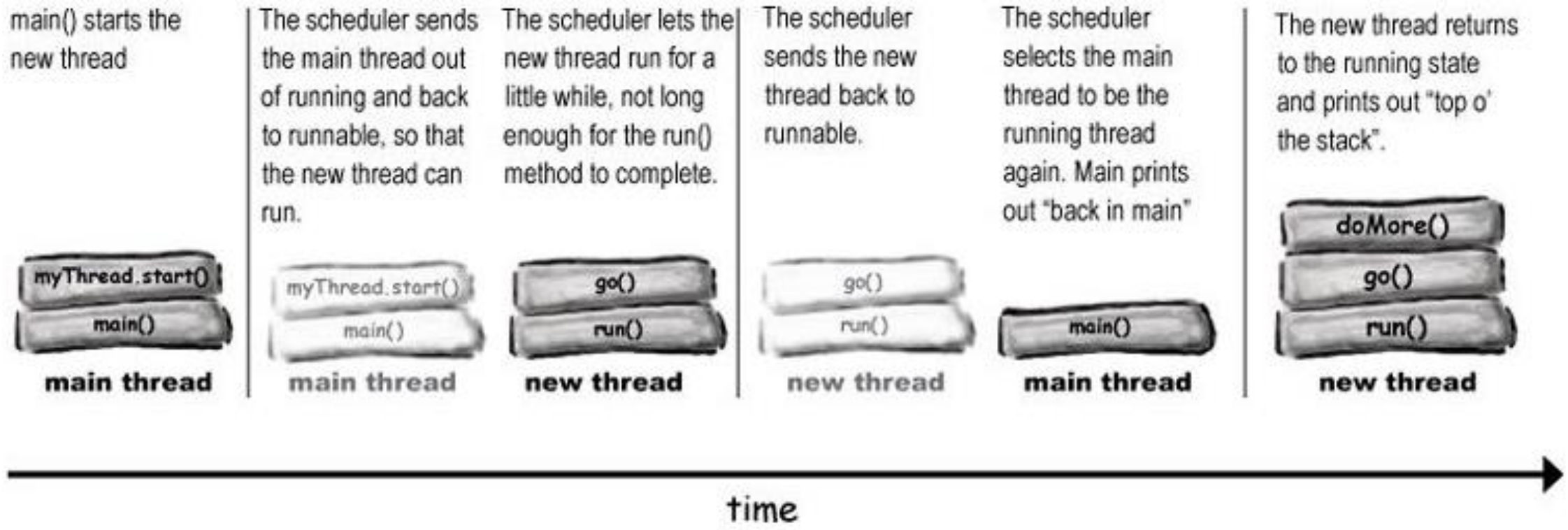
```
File Edit Window Help PickMe
% java ThreadTestDrive
back in main
top o' the stack
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java ThreadTestDrive
top o' the stack
back in main
% java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

순서가 아무렇게나 바뀌어 있다!

Sometimes it runs like this:



And sometimes it runs like this:



Putting a thread to sleep

쓰레드를 잠재울 수도 있다.

예를 들어, 2초 동안 쓰레드를 잠재우고 싶다면:

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

그런데 **sleep()** 메소드에서 **InterruptedException** (확인 예외)를 발생시킬 수 있기 때문에 아래처럼 **try/catch** 블록으로 감싸주어야 한다.

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

Sleep을 사용하여 프로그램을 더 예측 가능하게!

```
public class MyRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {

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

      doMore();
   }

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

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

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

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

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

실습과제 15-2

2개의 쓰레드를 만들고 시작시키는 예제:

```
public class RunThreads implements Runnable {

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

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

Make one Runnable instance.

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

Name the threads.

Start the threads.

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

2개의 쓰레드를 만들고 시작시키는 예제:

Part of the output when the loop iterates 25 times.