Slide 1

Administrivia

- Reminder: Continue to work on homeworks.
- Office hours Friday 12:30pm to 2:30pm. Info about office hours next week coming soon.

Slide 2

I/O in Java — Recap

- For text — `Scanner` and `PrintWriter` are a good start.
- For non-text data — `Data*Stream` for primitives and strings, or `Object*Stream` for objects. (The latter is what makes the game-framework screen editor works.)

Slide 3

Networking Basics

- Inter-computer communication based on layered approach and "protocols":
  - Application level — HTTP, FTP, telnet, SMTP, POP, IMAP, NTP, etc., etc.
  - Transport level — TCP (Transmission Control Protocol), UDP (User Datagram Protocol).
  - Network level — IP (Internet Protocol — addressing, routing of packets).
  - Link level — device drivers, etc.
- Messages are routed to
  - A machine ("host"), identified by IPA or name.
  - A process, identified by "port number" (16 bits). 0 — 1023 are "well-known ports", others available for applications.

Slide 4

Networking Basics — TCP and UDP

- UDP — independent messages, no guarantees about reliability or message order — analogous to (snailmail) letter.
- TCP — point-to-point channel, guarantees reliability and message order — analogous to phone call. Endpoints called "sockets".

Slide 5

Networking in Java

- Classes for communicating at application level — e.g., `URL` ("show URL" example).
- Classes for communicating at network level:
  - TCP — `Socket`, `ServerSocket`.
  - UDP — `Datagram*`.
- RMI (Remote Method Invocation).

Slide 6

Networking in Java — Sockets

- Client/server model:
  - Server sets up "server socket" specifying port number, then waits to accept connections. Connection generates socket.
  - Client connects to server by giving name/IPA and port number — generates a socket.
  - On each side, get input/output streams for socket. Program must define protocol for the two sides to communicate.
- Simple example in binary-I/O program from last week. More complex example — chat program.

Slide 7

Client/Server Programming with Sockets — Chat Example

- In client/server programming, program must define "protocol" for clients and server to communicate. For chat program, fairly simple:
- Interaction starts with client sending identifying information and server responding with list of participants.
- Interaction continues with client sending messages to server, which broadcasts them (to other clients), and accepting broadcast messages from server.
- Interaction ends when client sends "done" message to server, which broadcasts this information to other clients.

Slide 8

Client/Server Programming with Sockets — Chat Example, Continued

- Code is fairly simple — classes for client and server, plus inner class for server to keep track of clients. Only tricky bits are related to concurrency . . .
- Server needs to be able to communicate with multiple clients asynchronously (i.e., no way to know which one will send a message next). One way to deal with this — start a new thread for each client. Must then be sure these threads don't concurrently modify shared data (here, list of clients).
- Client needs to be able to present GUI and also listen for messages broadcast by server. Less coding here since GUI runs in its own thread automagically, so we can use the main thread to listen for message from server. Only complication is that anything in this thread that needs to change the GUI must use `SwingUtilities.invokeLater` to be sure changes happen in event dispatch thread.

Slide 9

- (Code on sample programs page.)

Slide 10

Networking in Java — RMI

- Motivation — for client/server applications, can be annoying to have to design your own protocol.
- Instead, idea is to define "remote objects" that can be treated (at program level) like any other objects — invoke methods.
- Typical use in client/server program:
  - Server creates some remote objects and "registers" them.
  - Clients look up server's remote objects and invoke their methods.
  - Both sides can pass around references to other remote objects.
- Dynamic code loading possible too.

Slide 11

Networking in Java — RMI, Quick How-To

- Define a class for remote objects:
  - Define interface that extends `Remote`
  - Define class that implements that interface, extends a Java "remote object" class. Can also include other methods, only available locally.
  - Write code using classes — if using as remote object, reference interface; otherwise can reference class.
- Compile and execute:
  - Compile as usual. (Prior to Java 1.5, an extra step was required to generate "stubs" to be used in communicating with remote objects as remote objects.
  - Make classes network-accessible.
  - Start `rmiregistry`.
  - Run server and clients as usual.

Slide 12

Networking in Java — RMI

- Example — revised chat program. Design is somewhat more elaborate than absolutely necessary, in an attempt to be modular and flexible:
  - Common interface `ChatParty` for remote objects for both client and server, with subinterfaces `ChatClient` and `ChatServer`, and classes implementing all of these.
  - Interface `ChatClientUI` for non-remote local UI for clients, with two implementations.
- Need for multithreading in server goes away — all handled by RMI under the hood (though we still need to be careful about possible concurrent access to variables — experiment suggests RMI may use multiple threads). In client UI, however, we still need separate threads to get input from the user and listen for messages from the server.

Slide 13

Minute Essay

- None — sign in.