Networking

Review

What is the point of RTTI?

a) It lets you discover the properties of classes you wrote

b) It lets you discover the properties of classes you downloaded

c) It lets you discover the properties of system classes

d) It lets you discover the properties of parent classes

Review

Which of these can be a problem with Generics?

a) They are extremely inefficient

b) They only work when also using interfaces

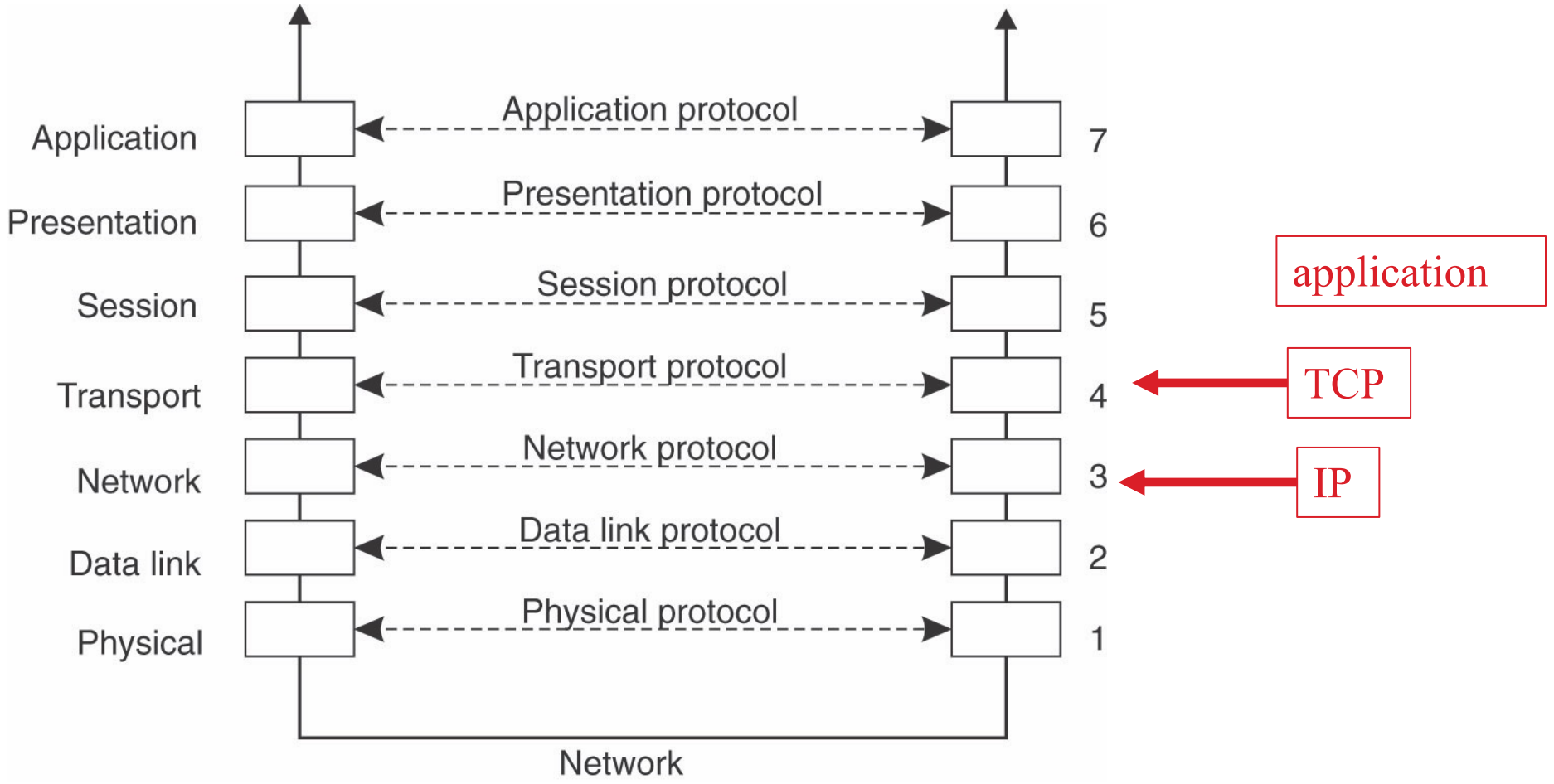
c) Template method calls may not work

d) They allow generic member data, but not generic methods

Objectives

- After this lecture, you will be able to:
  - define basic networking concepts
  - write Java programs that use sockets
  - understand the basics of the Java RMI protocol

OSI Model, 7 Layers



Tanenbaum & Van Steen

Low Level Communication

- OSI's upper levels are not always used
- Three “real” levels: application, connection, physical
- Connection level: two levels in OSI: transport and network
  - Network layer: ***IP (Internet Protocol)***
    - host-to-host, connectionless
  - Transport layer: either
    - ***UDP (User Datagram Protocol):*** process-to-process, connectionless, unreliable: fire and forget (and hope it gets there)
    - ***TCP (Transmission Control Protocol):*** process-to-process, structured, connection-oriented, reliable via ack/nack+retransmit; re-orders subpackets on arrival: if things don't arrive correctly, re-do

TCP/IP

- These are usually grouped as ***TCP/IP***: Transport Control Protocol and Internet Protocol (even though UDP is in there, too)
  - They rely on even lower-level protocols, down to the actual hardware
- Other, higher-level protocols or applications use their services
  - sendmail, ssh, http, ftp, …
  - socket programming
  - Remote Method Invocation (RMI)

Networking Concepts, cont.

- Basic ideas:
  - ***DNS*** (Domain Name Service) address: four-part numeric address of a computer; it usually has a multipart string alias. Examples: 192.168.10.100; cmu.heinz.edu
  - ***URL*** (Uniform Resource Locator): network name. Example: http://cmu.heinz.edu/media/article1.html
  - A URL must be translated (looked up, resolved) to a numeric DNS address

Networking Concepts, cont.

- ***Socket***: software record (class/object) for accessing the network
- ***Port***: integer connection number from 0-65535. These are logical, not physical, id's. 0-1023 are reserved (e.g. 22 is for ssh); 1024-49151 are for apps that may or may not be registered; the rest are for your program
  - If a port is in use, you can try another one
  - For security purposes, ports may be closed, or made not accessible through a firewall

https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

Networking Concepts, cont.

- Think of the socket like a ***street address***: your mail gets sent here.
- Then the port is like your ***apartment number***: not all mail sent to this street address belongs to you; you should only receive the stuff with your apartment number on it

Networking Concepts, cont.

- Typically, one application is listening to one port on a server; some of those are system applications (like a mail server or a web server); many ports are not used and are available for an app that you write – with the restrictions mentioned above.
  - But some apps need multiple listening ports, that's okay, just more complicated

Networking Concepts, cont.

- ***Client:*** app that connects to a server to use its services by first making a connection request, then sending a message and (optionally) getting a reply
- ***Server***: app that sets up a socket and waits for client connection requests, get service requests, and (optionally) replies
  - Server is started first - because it waits for a connection request, it just pauses.
  - Client (eventually) asks to connect, then messages can be exchanged

Networking Concepts, cont.

- How client and server interact depends on the app
  - ***Web server***: client sends URL, server sends HTML-encoded web page (or something more complicated)
  - ***Database server***: client sends SQL queries, server sends tuples and/or acknowledgement (e.g. INSERT doesn't return tuples)
  - ***Chat server***: exchange of String messages
  - ***Peer-to-peer*** networks: every machine is both a client and a server; data is either replicated across multiple machines or partitioned across multiple machines; finding things is more complicated
  - and many, many more

Sockets in Java

- The `Socket` and `ServerSocket` classes use the computer's network card
  - "Socket" should have been named "ClientSocket"
- The server knows (can look up) its own address
  - But it chooses a port to listen to
  - ***Port stepper***: code that tries to connect using a port; if it fails, increments the port number, tries again
- The client doesn't care about its own address
  - It choose a server by name AND port number
  - Its (return) address is given to the server via TCP

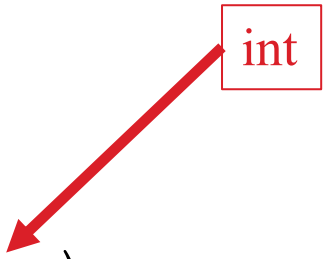
Server Sockets

- `ServerSocket`: uses the port number the app will listen to
- Then call `accept( )` to wait for a client connection request
  - Blocking call: waits for the connection
  - Returns a `Socket` object
- Then read requests and write replies to the `Socket` object
  - Why doesn't it directly use the ServerSocket for that? So that it can listen for other clients
  - This should use a new Thread for each connection

Client Sockets

- `Socket`: use the server's address and port number that the app will connect to, then write to the socket and read answers back
  - in a simple app, the client hard-codes these or gets them from the user
  - in real apps, they might come from a broker (database of available servers) or name server
- The client *must* know the server's address and the port it's listening to
  - If you mess up the address or the port, your "mail" goes to the wrong place and is discarded
  - Unless you're using a lookup service

Outline of Server Code




```
// Server – missing some important details
ServerSocket s = new ServerSocket(portNumber);
Socket c = s.accept();
while (some condition) {
   message = get stuff from c's inputStream
   write a reply to c's outputStream
}
```

Server Code

```
ServerSocket serverSocket = null;
Socket clientConnection = null;
try {
   serverSocket = new ServerSocket(8001);
   clientConnection = serverSocket.accept();
   // Get the input connection to read from the client
   // Get the output connection to send replies to the client
   Scanner in = new Scanner(clientConnection.getInputStream());

   PrintWriter out = new PrintWriter(
      new BufferedWriter(
         new OutputStreamWriter(
           clientConnection.getOutputStream())));
```

for example: we're hoping this port is available

input connection

output connection

Server Code, cont.

```
    // Get a message from the input stream
    String message = in.nextLine();
    // Send a message on the output stream
    out.println("OKAY");
} catch (IOException e) {
    e.printStackTrace();
}
```

Outline of Client Code


```
// Client – missing some important details
Socket s = new Socket(hostname, portNumber);
while (some condition) {
    write a message to s's outputStream
    message = get answer from s's inputStream
}
```

Client Code

```
Socket clientSocket = null;
try {
   clientSocket = new Socket("localhost", 8001);

   BufferedReader in = new BufferedReader(
      new InputStreamReader(clientSocket.getInputStream()));

   PrintWriter out = new PrintWriter(
         new BufferedWriter(
            new OutputStreamWriter(clientSocket.getOutputStream())));
```

for example

must match the port the server is listening to

input connection

output connection

Client Code, cont.

```
    // Send a message to the server
    out.println("Hello from earth");
    // Read the return message
    String inMessage = in.readLine();
    System.out.println("Client received back: " + inMessage);
} catch (IOException e) {
    e.printStackTrace();
}
```

Sockets, cont.

- For testing purposes:
  - "localhost" is used at the server's address in the Socket( ) constructor to connect to your own machine
  - Normally, use the real server address, and the programs are run on different machines
  - Run the server first, then run the client – two separate programs running on the same machine
  - Recall that the server will wait at the accept( ) call

Sockets, cont.

- Real servers run "forever" – that is, infinite loops - and are stopped manually
- Real servers start a Thread to handle the client Socket
  - Then they loop back to accept( ), so multiple clients can be handled in parallel

Sockets, cont.

- You can send objects across the connection instead: use the ObjectInputStream and ObjectOutputStream classes instead
- Both the client and the server must have exactly the same class definition that you're reading/writing
- That class must implement Serializable

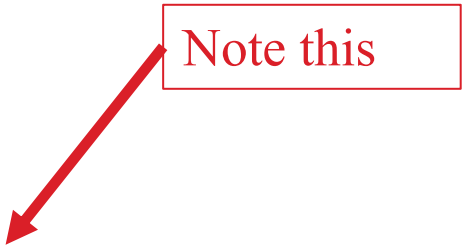
High Level Communication

- Programmers use function or procedure calls to structure code, so why not use that for communication?
  - ***Remote Procedure Call*** (RPC): mimic conventional function call – and – return, with parameters and return value
    - hides (most) low-level details of communication => transparency
  - ***Remote Method Invocation*** (RMI): object-oriented version of RPC, with some nicer features
- Java uses (can use) RMI

Java Remote Method Invocation

- Define an interface based on Remote:

```
public interface myInterface extends Remote
{
  returntype fn( params ) throws RemoteException;
}
```



- Server implements the interface, and inherits from UnicastRemoteObject

```
public class myClass extends UnicastRemoteObject implements
myInterface {
    … code for fn( ) …
}
```

Java RMI, cont.

- Server generates stubs using *rmic*
- Server registers with registry service from main( ) using `Naming` class:

  `bind( ), rebind( ), lookup( )`
- Server started in background
  - "Real" servers run forever loops, so the have to be stopped manually
  - Alternatively, some special message can signal "stop"

Java RMI, cont.

- Client uses registry service to find server
- Client calls remote method

```
public class client {
public static void main(String[] args) {
    String url = "rmi://someserver.com";
    myclass m = (myclass)Naming.lookup(url + "binding name");
    m.fn( );  // Calls remote method fn( )
}
```

Java RMI, cont.

- Java also supports:
  - ***callbacks***: the client registers with the server to respond to server-side events
  - ***reflection***: request message can contain information about what method should be called, via `invoke( )`