Serialization

Jörg Kienzle & Alexandre Denault

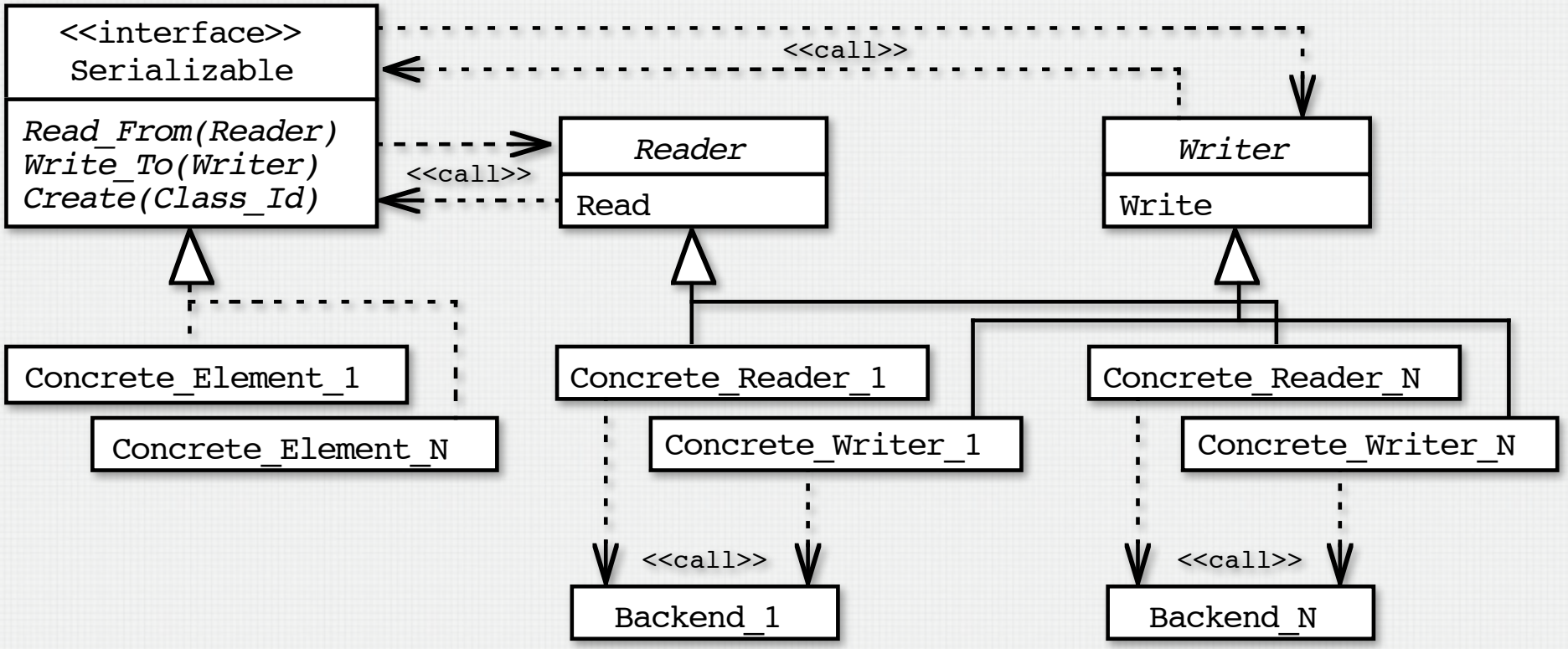
Serialization Overview

- Serializer Pattern
- Java Streams
  - Example: Writing to a file using streams
- Issues with Deep Serialization
- Customizing Serialization
- Serialization and Evolution
- Serialization and Networking
  - TCP/IP, IPs and Ports
  - Sockets
  - Setting Up A Connection
  - Example: Echo Client and Server
- Serialization and Turn-Based Games

Serialization

- Serialization is the process of taking the memory data structure of an object and encoding it into a serial (hence the term) sequence of bytes
- In our context, serialization is useful for:
  - Sending / receiving of objects / data over the network
  - Saving / loading of objects / data to a file

Serializer Design Pattern



Java Streams (1)

- Writing to a file

```
FileOutputStream out = new
    FileOutputStream("theTime");
ObjectOutputStream s = new
    ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

- ObjectOutputStream is constructed on some other stream
  - writeObject serializes the specified object, traverses its references to other objects recursively, and serializes them as well
  - writeObject throws a NotSerializableException if it is given an object that is not serializable

Java performs
Deep Serialization

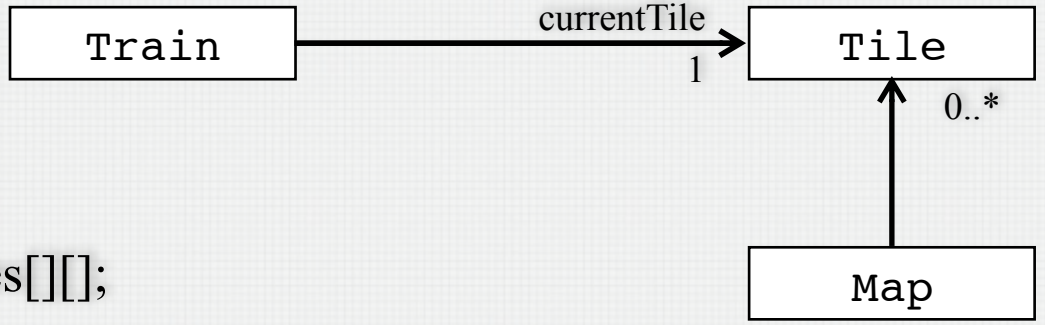
Java Streams (2)

- Reading from a file

```
FileInputStream in = new
   FileInputStream("theTime");
ObjectInputStream s = new
   ObjectInputStream(in);
String today = (String) s.readObject();
Date date = (Date) s.readObject();
```

- The objects must be read from the stream in the same order in which they were written
  - readObject deserializes the next object in the stream and traverses its references to other objects recursively to deserialize all objects that are reachable from it
  - Reading creates new objects!

Deep Serialization Example (1)

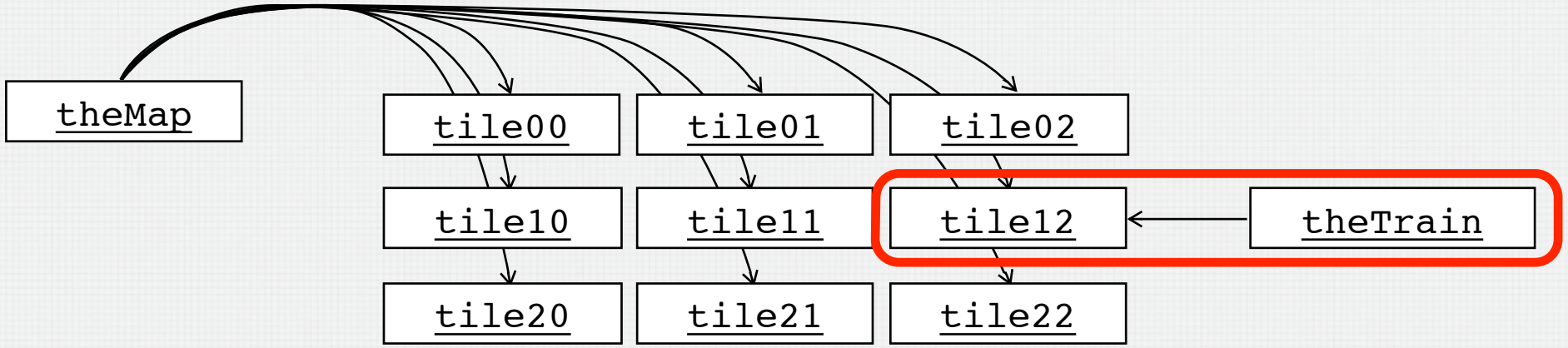



```
class Map {
   private Tile myTiles[][];
   public Map() {
      myTiles = new myTiles[3][3];
      for (int y=0; y<3; y++) {
         for (int x=0; x<3; x++) {
            myTiles[x][y] = new Tile(x,y);
         }
         y++;
      }
   }
   public Tile getTile(int x,y) {
      return myTiles[x][y];
   }
}
```

```
class Train {
   private Tile currentTile;
   public void setPosition(Tile t) {
      currentTile = t;
   }
}
```

Deep Serialization Example (2)

```
public main() {
    Map theMap = new Map();
    Train theTrain = new Train();
    theTrain.setPosition(theMap.getTile(2,1));
```

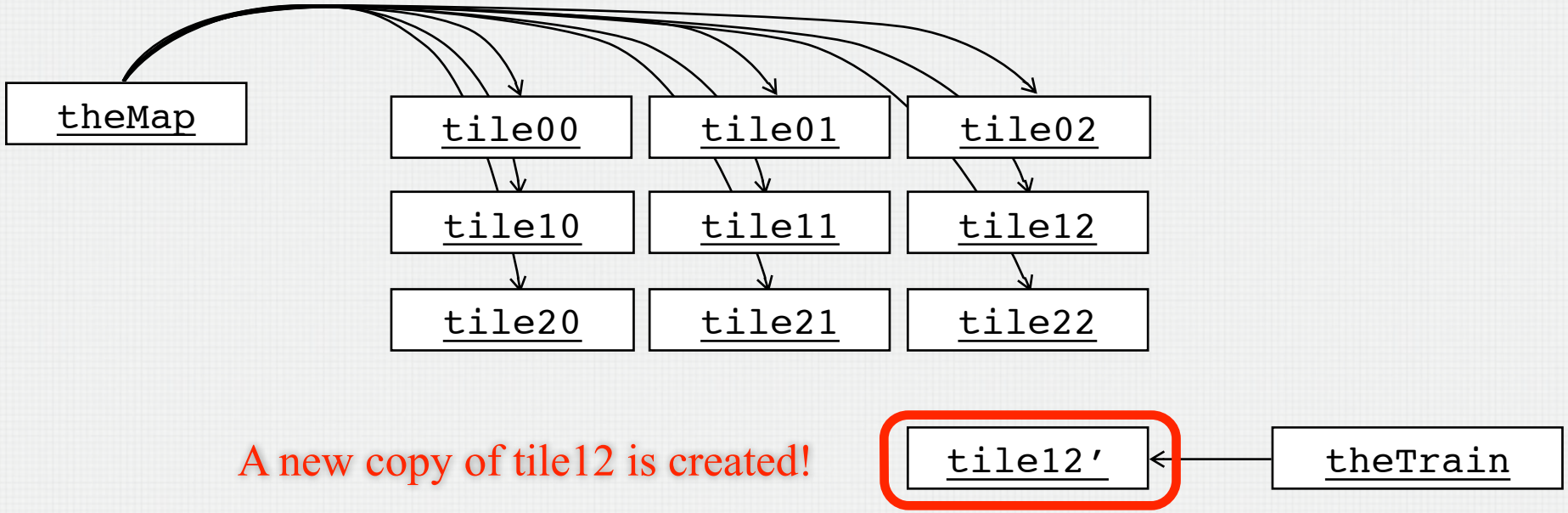


Both objects are serialized!

```
    ...
    myOutputStream.writeObject(theTrain);
}
```

Deep Serialization Example (3)

Train theTrain = myInputStream.readObject();



Customizing Serialization

- Make a class serializable by implementing Serializable

```
public class MySerializableClass implements Serializable
  {...}
```

- Default implementation deep-serializes everything except
  - Transient and static fields are not serialized

- Custom serialization by overriding writeObject and readObject

```
private void writeObject(ObjectOutputStream s)
  throws IOException {
  s.defaultWriteObject();
  // customized serialization code
}
```

Serialization and Evolution

- Imagine the following scenario:
  - A program writes an object a of class A to a file f
  - Class A is modified, for instance by adding a new field
  - The program attempts to read a from the file f
- The Java run-time verifies that the classes are compatible with respect to serialization, and if not, throws a InvalidClassException
- How is this done?

Serial Version ID

- Java stores a 64-bit value with your object
  - It's a hash computed based on the class signature
- You can specify this value for class by defining

```
static final long serialVersionUID = 1234L;
```

- Default behavior
  - Deleted fields are ignored
  - New fields remain uninitialized
- It's probably best to customize serialization

TCP/IP

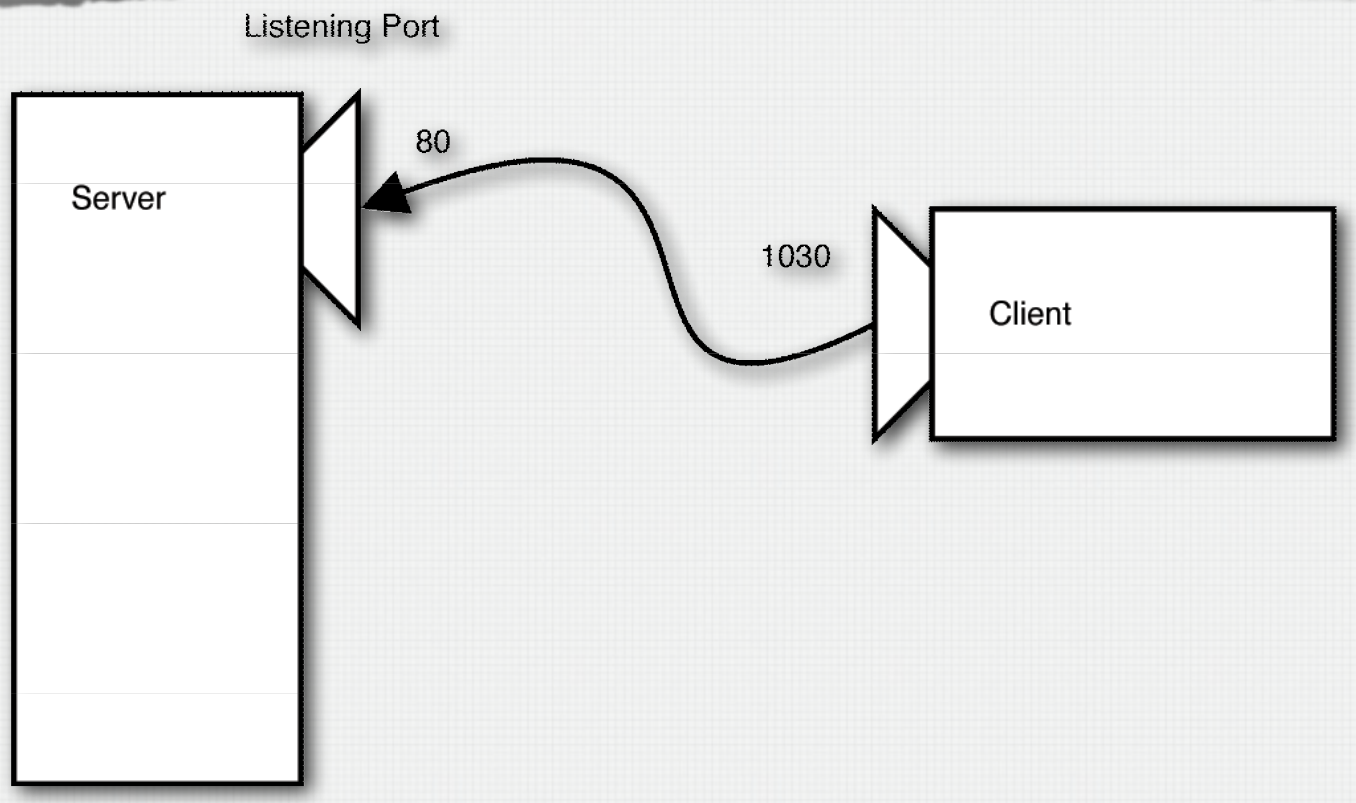
- Every machine has a unique IP address
  - 132.206.51.234 (CS mail server)
  - mail.cs.mcgill.ca (domain name)
- Every machine has a 65536 ports
  - 0 - 1023: Well-known Ports
    - 20/21 : File transfer protocol (FTP)
    - 22 : Secure Shell (SSH)
    - 23 : Telnet
    - 25 : Simple Mail Transfer Protocol (SMTP)
    - 80 : World Wide Web (HTTP)
    - 137/138/139 : NetBIOS (Microsoft File Sharing)
    - 143 : Internet Mail Protocol (IMAP)
    - 443 : HTTP protocol over TLS/SSL
  - 1024 - 49151: Registered Ports
  - 49152 - 65535: Dynamic / Private Ports

Sockets

- A socket is one endpoint of a two-way communication link between two applications
  - Sockets are bound to a port number
- Java.net provides Socket and ServerSocket classes that hide the operating system details

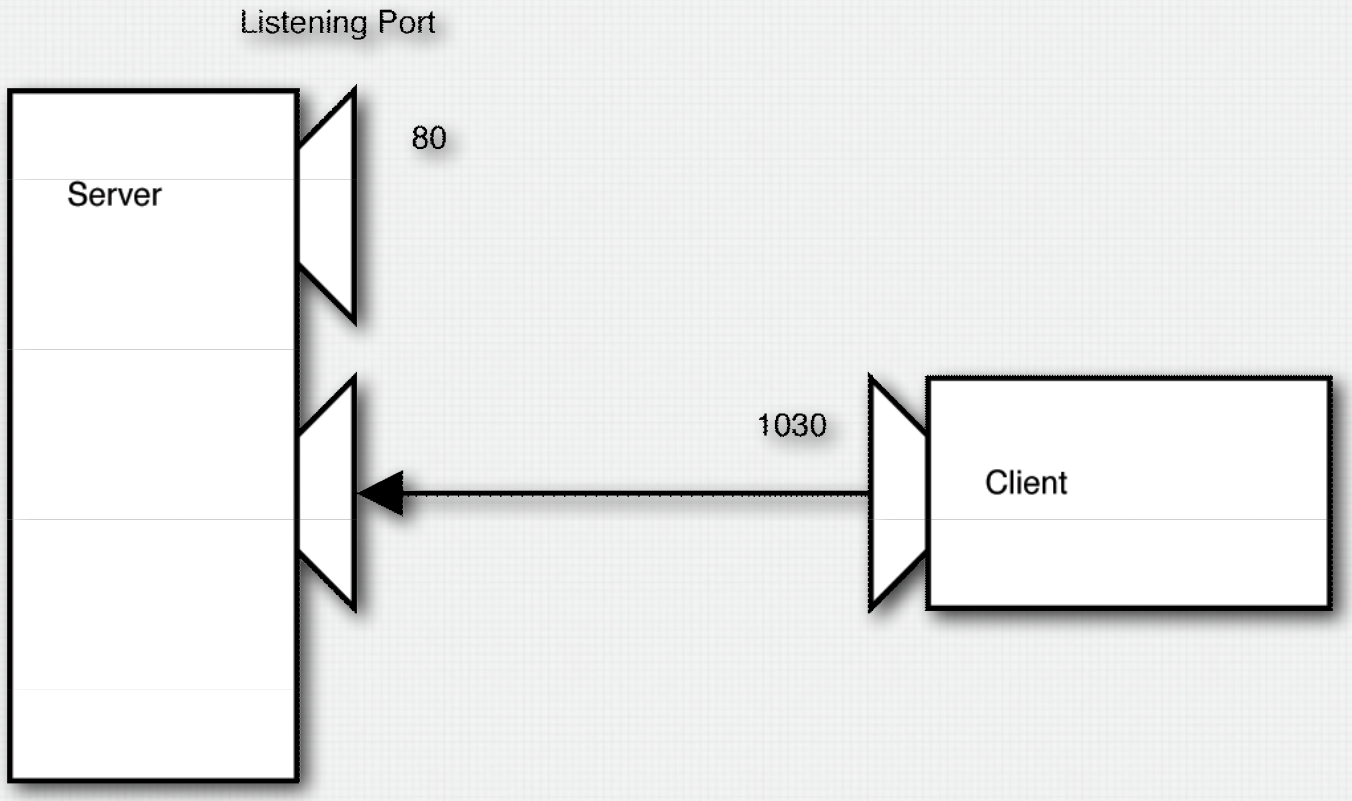
Setting Up a Connection (1)

- Server creates a socket, listens on a port
- Client creates a socket and connects to the server socket



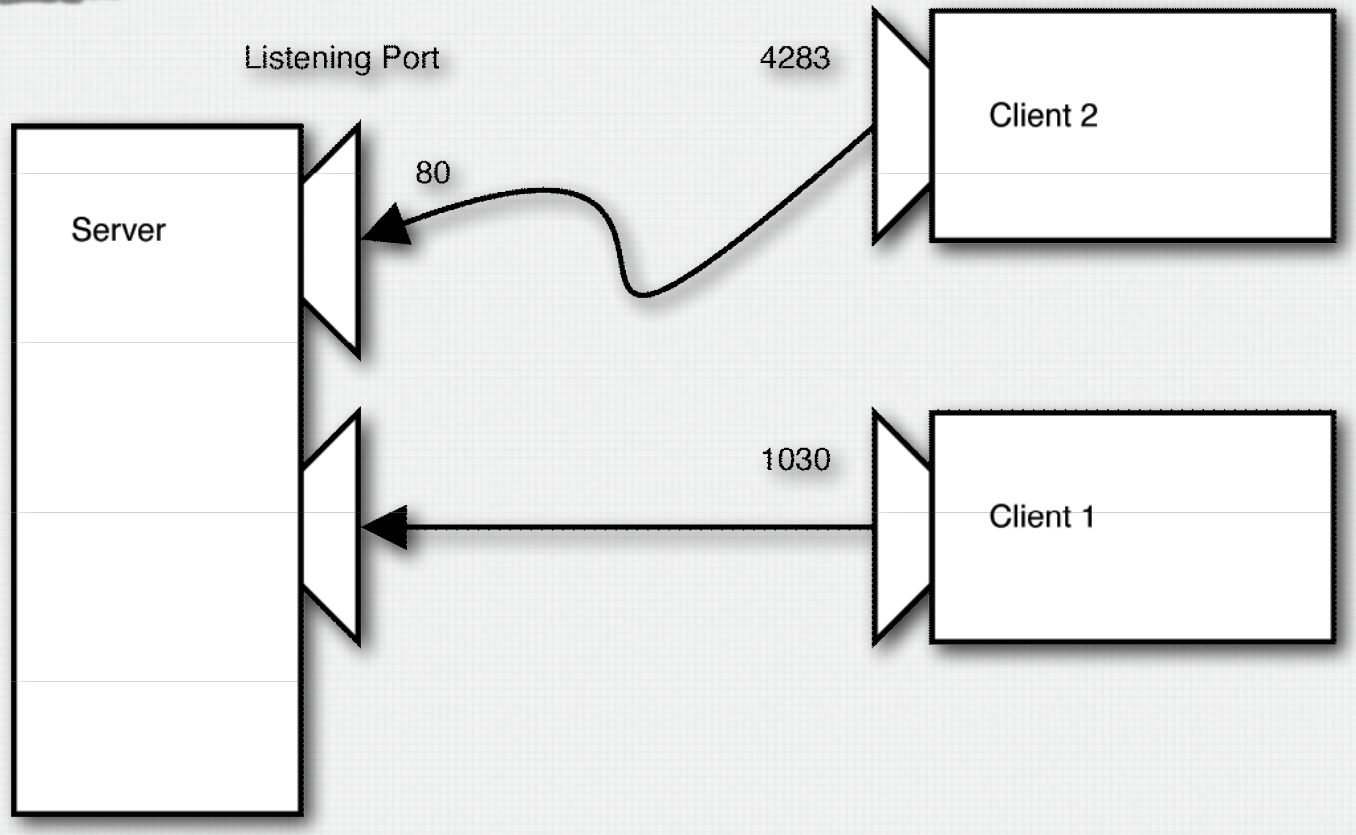
Setting Up a Connection (2)

- Upon connection, a new server socket is created



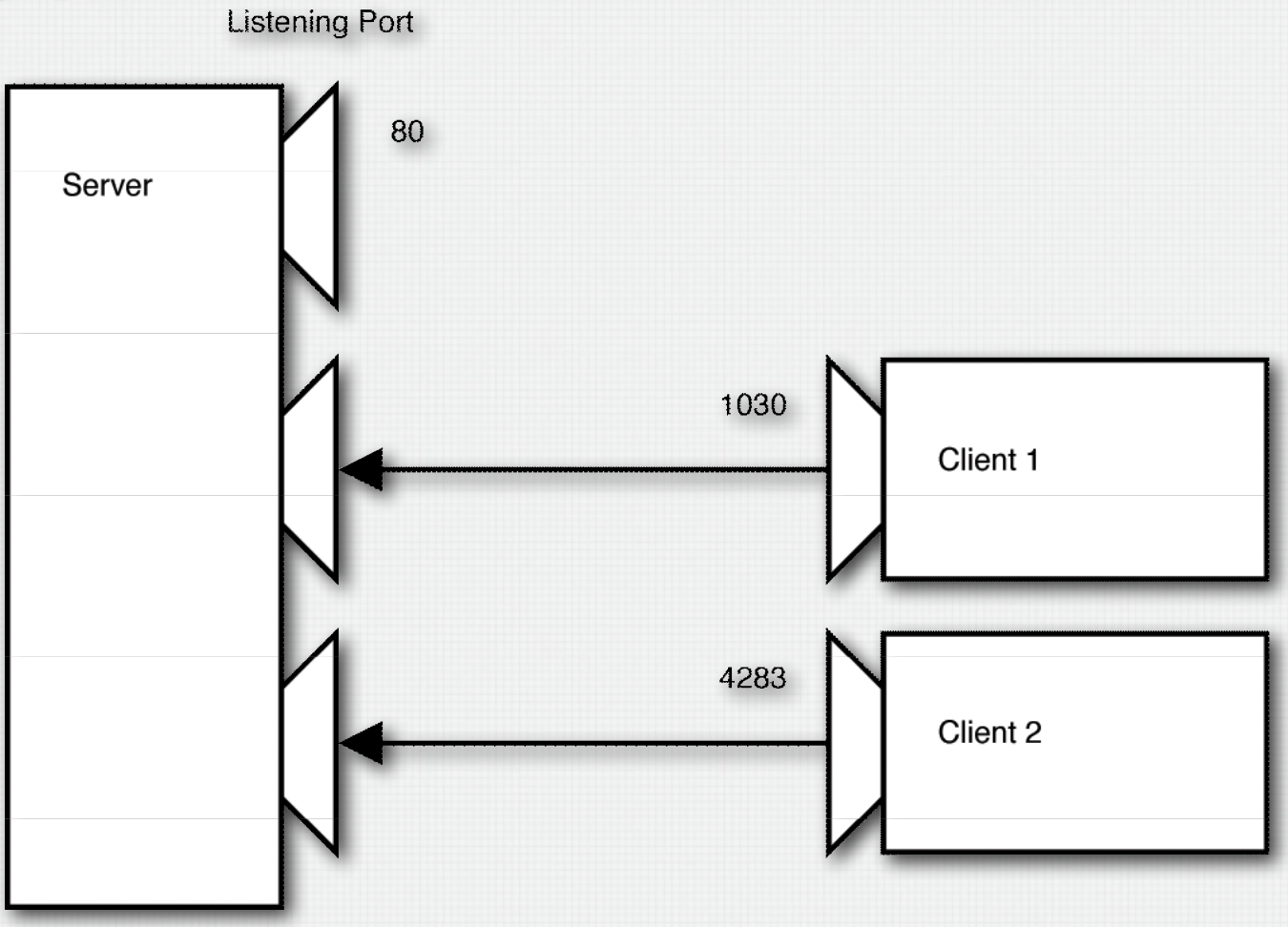
Setting Up a Connection (3)

- A new socket is needed to allow the server to continue to listen on the original socket



Setting Up a Connection (4)

- A new socket is needed to allow the server to continue to listen on the original socket



Simple Example: Echo Client & Server

- Echo client
  - Creates a socket and connects to the echo server
  - Reads input from keyboard and forwards it to the server
  - Displays all data sent by the server
- Echo server
  - Receives data and sends it back to the client

Echo Client (1)

```
import java.io.*;
import java.net.*;
public class EchoClient {
    public static void main(String[] args)
      throws IOException {
    Socket echoSocket = null;
    PrintWriter out = null;
    BufferedReader in = null;
```

Echo Client (2)

connect to port 4444

```
try {
   echoSocket = new Socket("taranis.cs.mcgill.ca", 4444);
   out = new PrintWriter (echoSocket.getOutputStream(), true);
   in = new BufferedReader(new InputStreamReader
        (echoSocket.getInputStream()));
} catch (UnknownHostException e) {
   System.err.println("Don't know about host: taranis.");
   System.exit(1);
} catch (IOException e) {
   System.err.println("Couldn't get I/O for the connection to: taranis.");
   System.exit(1);
}
```

Echo Client (3)

```
        String userInput;
        BufferedReader stdIn = new BufferedReader
                               InputStreamReader(System.in));

        while ((userInput = stdIn.readLine()) != null) {
          out.println(userInput);
          System.out.println("echo: " + in.readLine());
        }
        out.close();
        in.close();
        stdIn.close();
        echoSocket.close();
    }
}
```

send to server

read from keyboard

display on screen

read from server

Echo Server (1) - Listening

- The server uses two types of sockets
  - A ServerSocket to listen for new connection
  - Regular Socket to communicate with the client
- Example: setting up a server socket and waiting for incoming connections on port 4444

```
try {
  serverSocket = new ServerSocket(4444);
catch (IOException e) {
  System.out.println("Could not listen on port: 4444");
  System.exit(-1)
}
```

Echo Server (2) - Accepting

- A call to accept() blocks until a connection is established
- Accept() hands back a new Socket reference, that can then be used for communication with the client

```
Socket clientSocket = null;
try {
   clientSocket = serverSocket.accept();
} catch (IOException e) {
   System.out.println("Accept failed: 4444");
   System.exit(-1);
}
```

Echo Server (3) - Closing

- Once the server socket is closed, the server will not accept any new incoming communication attempts

  serverSocket.close();

- This call does not affect sockets that are already established.
- To disconnect clients from the server, each socket must be individually closed

Supporting Many Clients

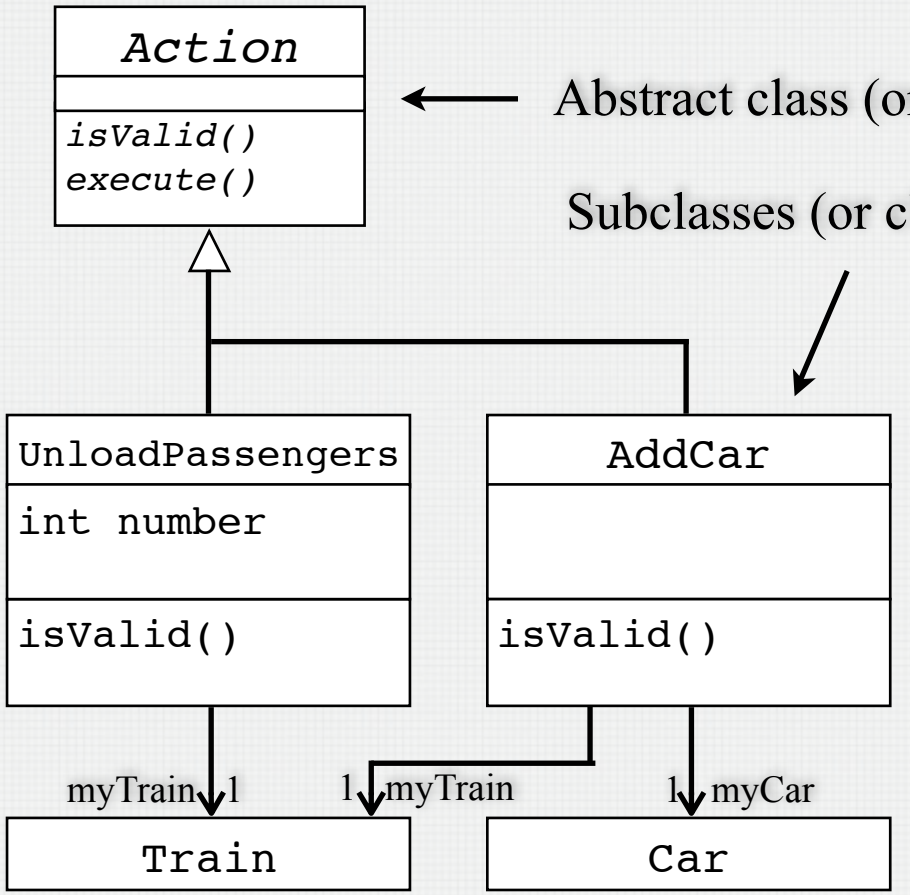
- The echo server we described can listen for and handle a single connection request
- New client connection requests are queued at the port, so the server must accept the connections sequentially
- The server can service them simultaneously through the use of threads - one thread for each client connection

```
while (true) {
  accept a connection ;
  create a thread to deal with the client ;
}
```

Networking and Turn-Based Games

- Movements of players have to be sent over the network
  - From client to server, or
  - From peer to peer
- Object-oriented solution (Command pattern)
  - Define a class hierarchy of actions
  - Each action knows how to validate and execute itself, which results in updating the game state
  - Uses serialization, works with both Sockets or RMI

Action Hierarchy




```
public AddCar extends Action {
   public AddCar(Train t, Car c) {
      myTrain = t;
      myCar = c;
   }
   public void execute() {
      myTrain.addCar(myCar);
   }
}
```

Action Execution in Peer-2-Peer Setting

- On current player's computer
  - GUI handles player input until it determined what action the player wants to execute
  - GUI instantiates the corresponding action
  - GUI verifies if action is valid by calling isValid()
  - isValid() calls the appropriate verification methods on the model (i.e. package / classes containing the game state)
  - GUI gives action to the action executor
  - Executor executes action on the game state by calling execute()
  - Action is sent to the other players' computers
- On other computers
  - Action instance is read from the network and given to executor
  - Executor executes action on the game state by calling execute()

Action Execution in Client-Server Setting

- On current player's client computer
  - GUI handles player input until it determined what action the player wants to execute
    - Optional verification (only possible if the client knows about relevant game state)
  - GUI instantiates the corresponding action
  - Action is sent to server
- On server
  - Action instance is read from the network and given to executor
  - Executor validates action by calling isValid() (if not already done on the client)
  - Executor executes action on the game state by calling execute()
    - If no previous verification, and if action invalid, exception is sent back to client
    - Otherwise, "action effect" is sent to all players
- On all player's client computers
  - Action effects are displayed

Questions?