Frameworks

Reusable Software

Frameworks are ...

❑ a reusable application or environment that can be modified by addition of application-specific code, *without modifying the framework* code.

❑ frameworks provide a reusable *architecture*, not just reusable code.

Examples

Java Collections Framework

- you can use it to create custom collections that reuse the base collections logic and interfaces
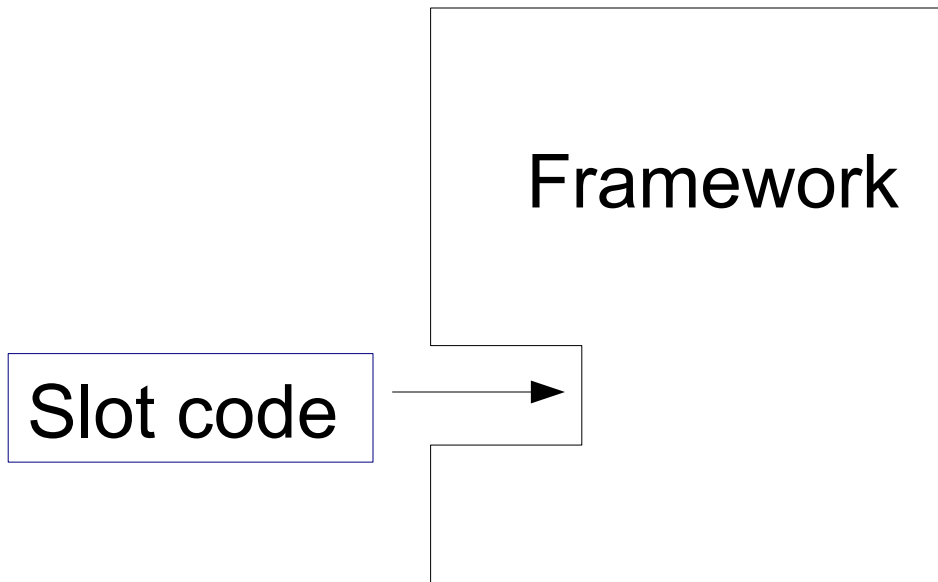
Web Frameworks

- provide logic and architecture for request mapping, session management, database access, and more.
- Spring Framework, Struts2, Play (Java)
- Django (Python)
- Rails (Ruby)
- Symphony, CakePHP, Lavarel (PHP)

JUnit testing framework

"Slots": Required Customization

Many frameworks require you to add some code before they can be used.

Some people call these slots.

Framework

Slot code →

Slot can be a class or a method.

Object Client-Server Framework

OCSF is a TCP-based client-server framework.

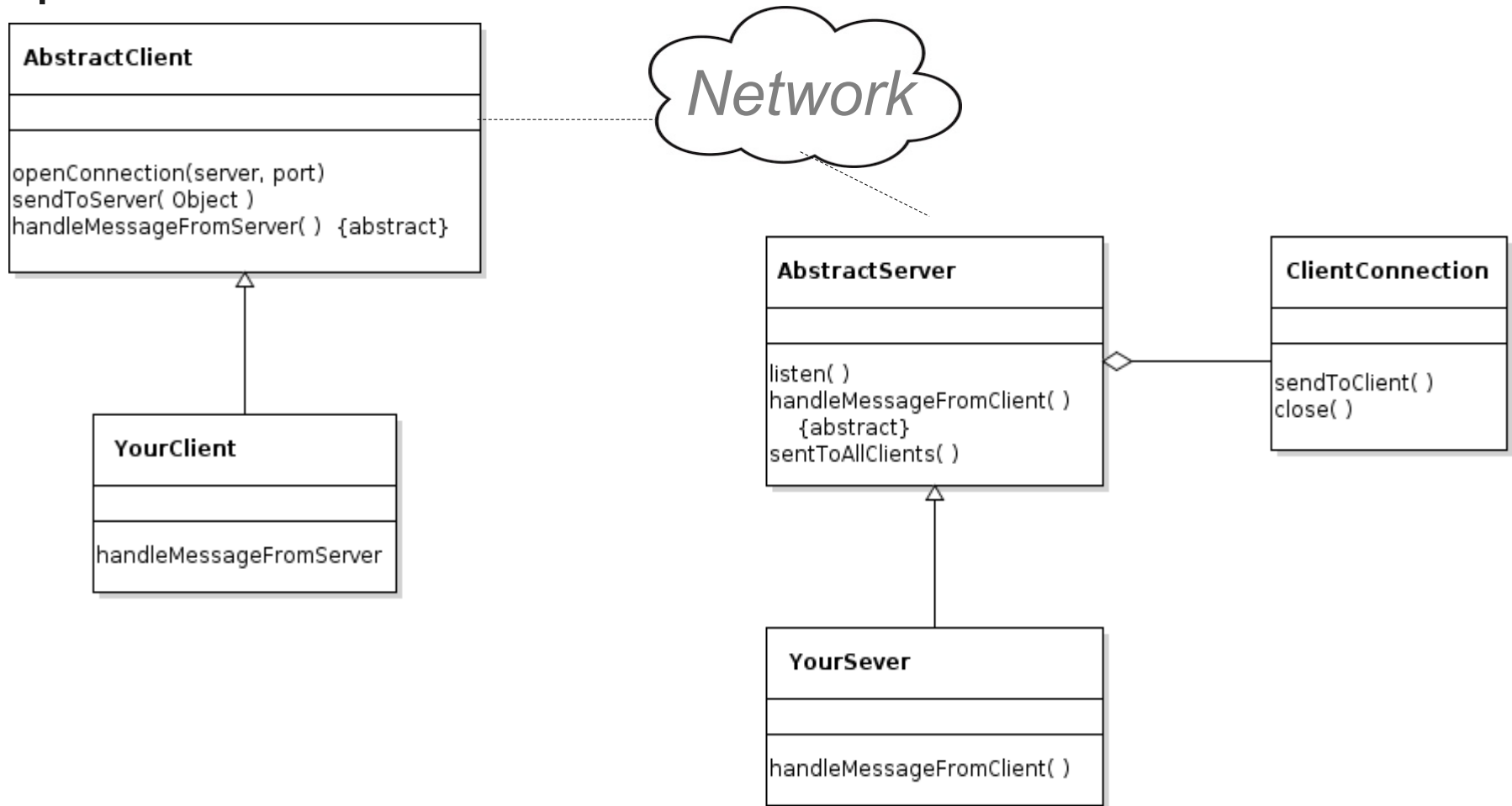*Client Side:*

connects to server

sends messages to server

receives message from server

handles connect & disconnect events

Server Side:

· *manage connections from clients*

· *receive messages from clients*

· *send messages to clients*

OCSF

AbstractClient

openConnection(server, port)
sendToServer(Object)
handleMessageFromServer( ) {abstract}

YourClient

handleMessageFromServer

Network

AbstractServer

listen( )
handleMessageFromClient( )
    {abstract}
sentToAllClients( )

ClientConnection

sendToClient( )
close( )

YourSever

handleMessageFromClient( )

"slots" are usually *callbacks*

A callback is a method in your code that the framework invokes. You *start* the framework, then it *calls you back* when an event occurs.

This is also called inversion of control.

  You start the framework, then it takes control.

In OCSF, the required *callbacks* (slots) are:

Client: `handleMessageFromServer`

Server: `handleMessageFromClient`

That's it! You can write a network client-server application just by writing 2 methods.

Code Reuse, Architecture Reuse

OCSF provides the architecture and code for a TCP client-server application.

You can use the framework without knowing *how* it works

(or how TCP works).

- but you should study OCSF to learn how to use networking in Java.

You can *modify* and *extend* the framework by overriding callbacks (slots and hooks).

OCSF AbstractClient

```
<<controls>> (commands to the framework)
  openConnection( )
  sendToServer( Object )
  closeConnection()
<<hooks>>     (optional callbacks)
  connectionEstablished( )
  connectionClosed( )
  connectionException( )
<<slot>>      (required callbacks)
  handleMessageFromServer( Object )
<<accessors & mutators>>
  isConnected()
  getPort( ), setPort(port)
  getHost( ), setHost(server)
```

OCSF AbstractServer

```
<<controls>> (commands to the framework)
  listen( )
  stopListening( )
  sendToAllClients(Object msg)
<<hooks>>      (optional callbacks)
  clientConnected( )
  clientDisconnected( )
  several others
<<slot>>       (required callbacks)
  handleMessageFromClient( Object )
<<accessors & mutators>>
  isListening()
  getClientConnection(int id)
  getPort( ), setPort(port)
```

Example

A messaging client that sends strings (message).

All clients receive the message.

Use port 5555 (port > 1024 is suggested for Linux and MacOS).

Client side

❑ Extend AbstractClient & implement the callback method

```java
import com.lloseng.ocsf.client.AbstractClient;
public class ChatClient extends AbstractClient {
   public ChatClient(String host, int port) {
     super(host, port);
    }


   @Override
   protected void handleMessageFromServer(Object msg)
   {
      System.out.println("> " + msg);
   }
}
```

Run the client

1) Create a client with server (host) name and server port.

2) Connect to the server.

3) In a loop...

      1) wait for user to type a message

      2) send message to server

TODO: provide a way to quit

Server Side: an Echo Server

- ❑ Create a server that just echoes messages to all client.

- ❑ Extend AbstractServer. Override the "slot" method.

```java
public class EchoServer extends AbstractServer {

  /** create a new echo server */
  public EchoServer(int port) {
    super(port);
  }


  @Override
  protected void handleMessageFromClient(
        Object msg, ConnectionToClient client) {

    super.sendToAllClients(msg);
  }
```

Running the Server

```java
private static final int PORT = 5555;

public static void main(String[] args) {

    EchoServer server = new EchoServer(PORT);
    try {
        server.listen();
        System.out.printf("Listening on port %d\n",
                          PORT);
    } catch (IOException e) {
        System.out.println("Couldn't start server:");
        System.out.println(e);
    }
}
```

Using Hooks

Server:

print a message when a client connects or disconn.

Client:

print a message if server closes the connection.

What hooks (callbacks) can we should use to do this?

How OCSF Works

You don't know *how* a framework works in order to use it.

This is the advantage of a framework; it provides an abstraction for what you want to do.

Think "value added" … don't waste time re-inventing logic and architecture that has been done already.

TCP is Connection Oriented
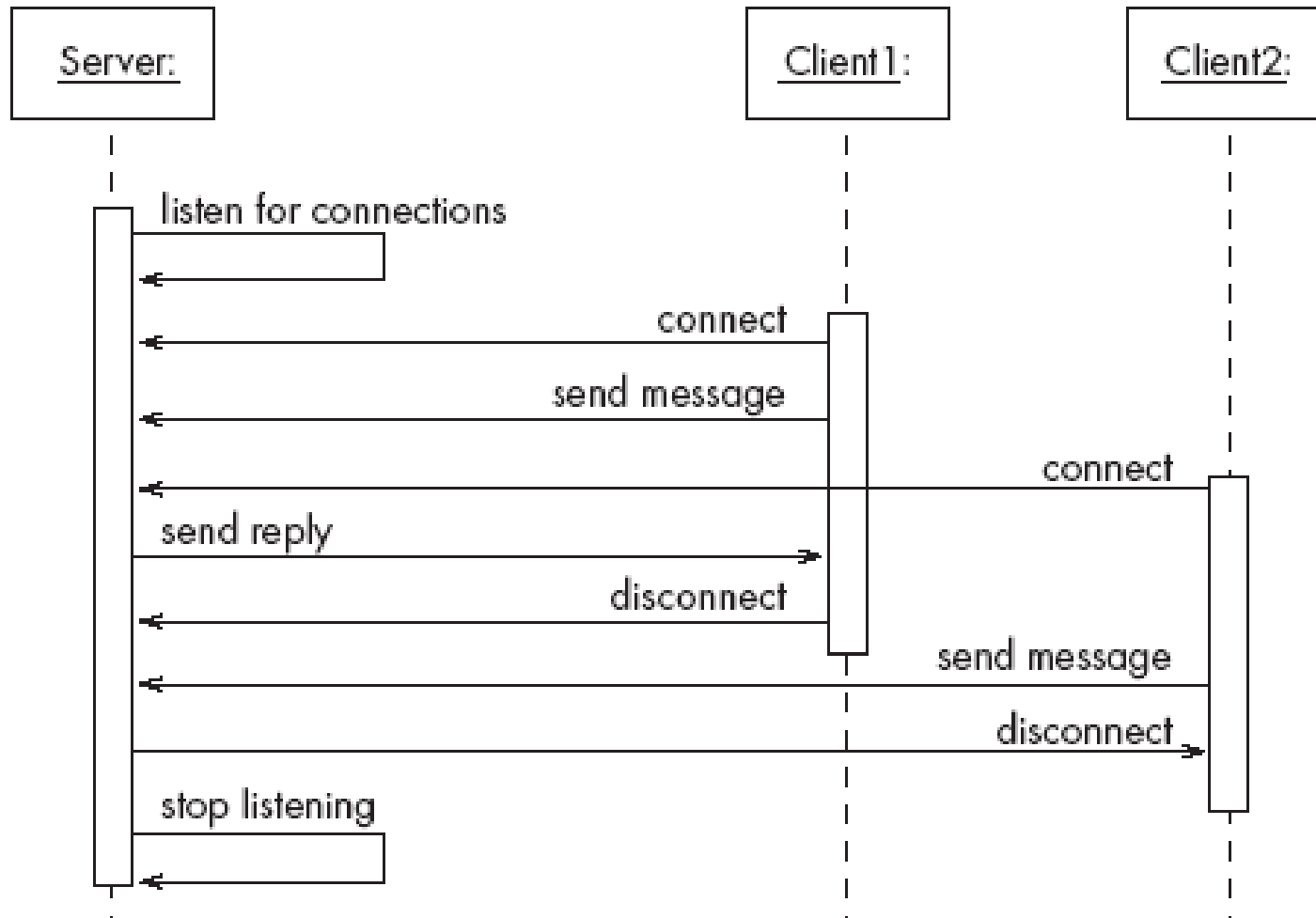
In TCP, a server listens for connections on a *port number*.

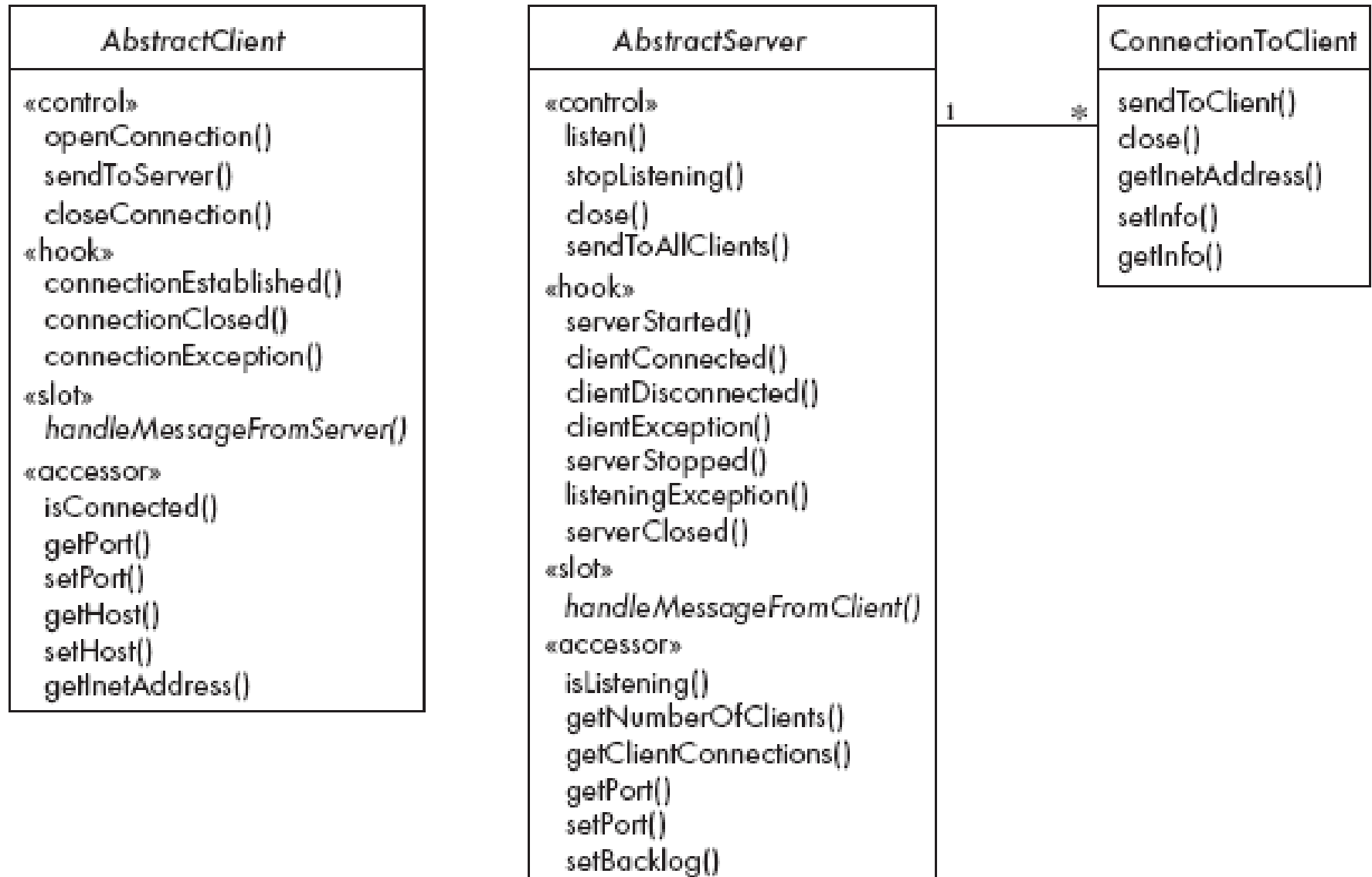A client connects using server's IP address and port number.

Either side can send messages.

A server can accept *many connections* on the same port.

When a client connects, the server creates a *new thread* to handle communication with one client.

TCP Example

OCSF's Main Classes

AbstractClient

«control»
 openConnection()
 sendToServer()
 closeConnection()
«hook»
 connectionEstablished()
 connectionClosed()
 connectionException()
«slot»
 handleMessageFromServer()
«accessor»
 isConnected()
 getPort()
 setPort()
 getHost()
 setHost()
 getInetAddress()

AbstractServer

«control»
 listen()
 stopListening()
 close()
 sendToAllClients()
«hook»
 serverStarted()
 clientConnected()
 clientDisconnected()
 clientException()
 serverStopped()
 listeningException()
 serverClosed()
«slot»
 handleMessageFromClient()
«accessor»
 isListening()
 getNumberOfClients()
 getClientConnections()
 getPort()
 setPort()
 setBacklog()

1 —— *

ConnectionToClient

sendToClient()
close()
getInetAddress()
setInfo()
getInfo()

The Client Side

AbstractClient m*ust* be subclassed

- Any subclass must provide an implementation for handleMessageFromServer
  - Takes appropriate action when a message is received from a server

Implements the Runnable interface

- Has a run method which
  - Contains a loop that executes for the lifetime of the thread

trantLock();

The public interface of AbstractClient

Control methods (you can call these, but don't override)
- openConnection
- closeConnection
- sendToServer

Status and Accessor/Mutator
- isConnected
- getHost
- setHost
- getPort
- setPort
- getInetAddress

Callback methods of AbstractClient

Callbacks that *may* be overridden:

- connectionEstablished
- connectionClosed

Callback that *must* be implemented:

- handleMessageFromServer

References

Object Client-Server Framework

> http://www.site.uottawa.ca/school/research/lloseng/supportMaterial/ocsf/ocsf.html (OCSF is in chapter 3 and chapter 6)

> Youtube lecture by one of the authors of OCSF:

> https://www.youtube.com/watch?v=hGM1eT8EVuI

XMPP - another messaging framework with many applications.

> https://xmpp.org/

> Smack - Java XMPP client library