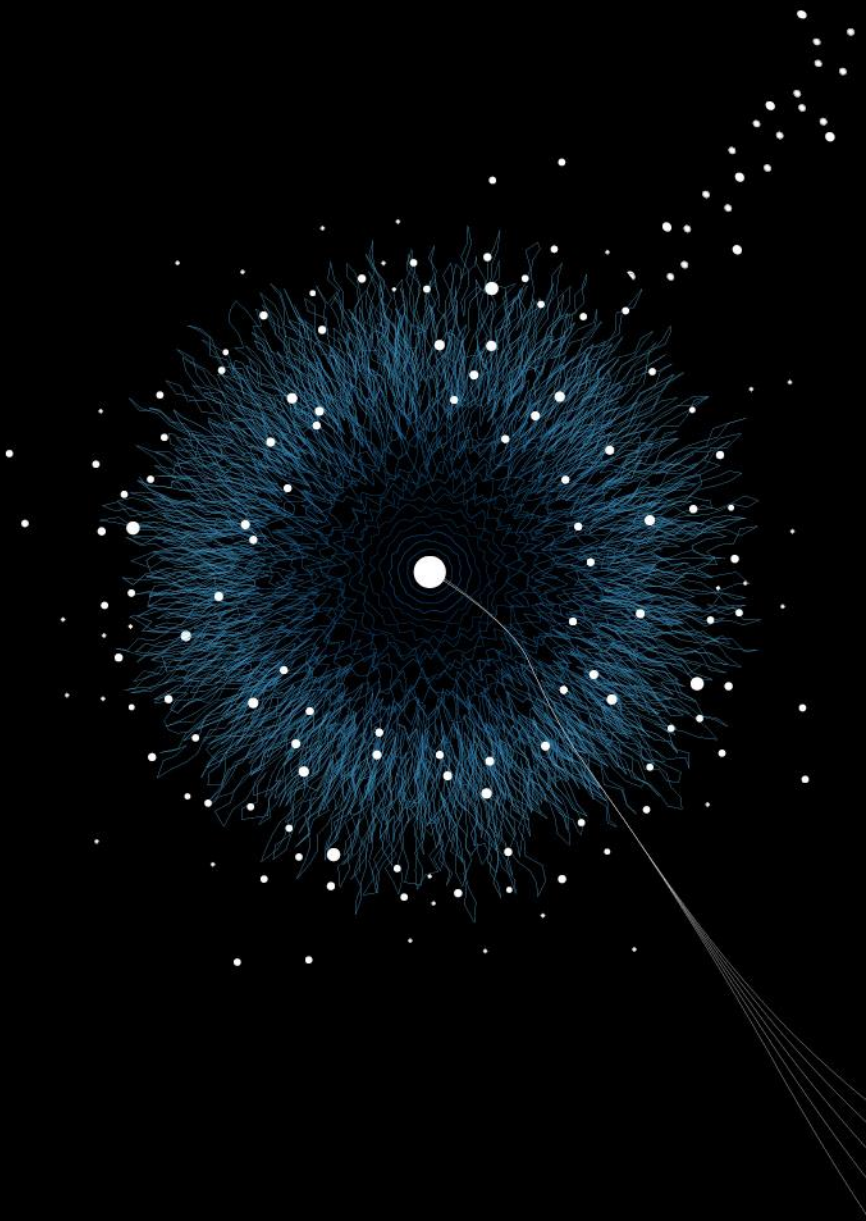
NETWORKING, DEFENSIVE PROGRAMMING, AND PARSING



M2 PROGRAMMING LECTURE 7.2

BY ALEXANDER STEKELENBURG

UNIVERSITY
OF TWENTE.

TOPICS OF THIS LECTURE

- Summary and questions about Week 6

- Networking and the client-server architecture in Java
- Parsing
- Defensive programming

PROGRAMMING OVERVIEW

Week 1 (IP) Procedural programming Arrays Debugging	**Week 2 (IP)** Classes and Objects Documenting and Testing Exceptions (AP)	**Week 3 (IP)** Interfaces, Inheritance, Polymorphism Subtyping Streams and Files (AP)
Week 4 (AP) Collections: Set, List, Map Specification with JML	**Week 5 (AP)** Design Patterns User Interfaces	**Week 6 (AP)** Basic Concurrency Project kick-off
Week 7 (AP) Basic Networking Defensive programming Security	**Week 8/9 (Project)** Project	**Week 10 (Project)** Project Exam

PROGRAMMING OVERVIEW

Java Notes book:

- Week 1: Chapter 1 – 4
- Week 2: Chapter 5.1 – 5.4
- Week 3: Chapter 5.5 – 5.8
- Week 4: Chapter 7.1 – 7.3, 8.1 – 8.3, 9.1 – 9.2, 10.1 – 10.4, 11.1 – 11.2
- Week 6: Chapter 12.1 – 12.3
- Week 7: Chapter 11.4

SO FAR

Good programs are

- Functional
- Reliable, testable
- Understandable, documented
- Formally specified / verified
- Designed for **change**

SO FAR

- (W1) Procedural programming (variables, types, procedures, control statements)
- (W2) Basics of object-oriented programming (classes, packages, static, final, public, private, constructors, standard methods)
- (W2) Program design principles (encapsulation, single responsibility principle, cohesion, coupling, command-query separation)
- (W2) Testing and code readability
- (W2) Exceptions
- (W3) Inheritance, polymorphism, typing, abstract classes, inheritance
- (W3) Streams and files
- (W4) Java Collections (List, Set, Map, generics)
- (W4) Specifications with JML
- (W5) Design patterns
- (W6) Concurrency

Q&A

Any questions about…

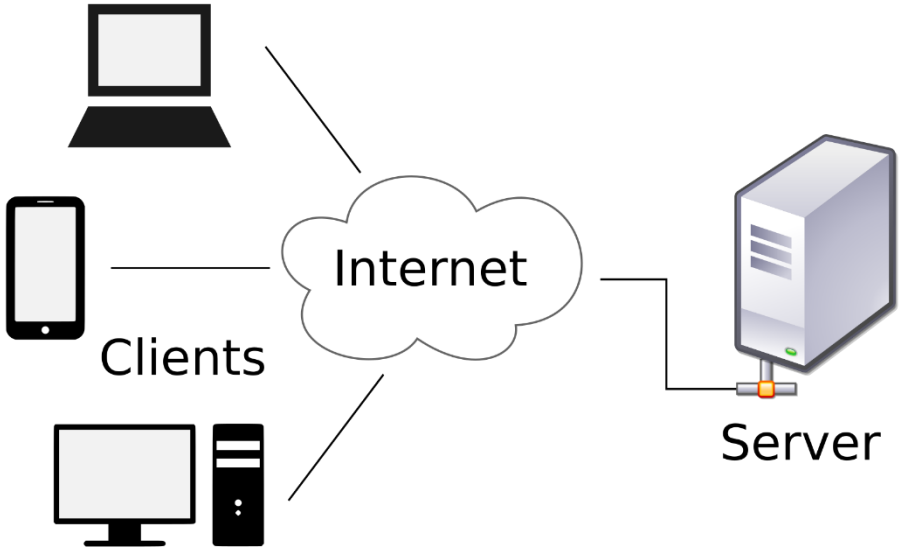
- Concurrency: race conditions, data races, atomicity
- Threads: Thread#start, Thread#join
- Synchronization: synchronized, ReentrantLock
- Monitors: wait, notify, notifyAll, await, signal, signalAll

CLIENT-SERVER ARCHITECTURE

- Central **server**, many **clients**
- Communication between parts via network
- Communication **protocols**, for example REST, HTTP
- **Thin client** or **fat client**

- \+ Centralized management of resources on the server
- \+ Centralized security
- \- Single point of failure
- \- Relies on network communications

- Examples
  - Almost every website
  - The programming project



CLIENT-SERVER ARCHITECTURE IN JAVA

- **Server**: ServerSocket for incoming connections, Socket for each client
- **Client**: Socket for connection with server

- Every Socket has an InputStream and an OutputStream

- You learned how to use these in **week 3**

- Use the InputStream to **read** bytes
- Use the OutputStream to **write** bytes
- Use Readers and Writers to read/write **characters**

- For example, in AP-5.11 you used the BufferedReader and the PrintWriter

SOCKETS

- A Socket is an object used for **communicating across a network**
- Represents a **TCP/IP** connection
- When a Socket is created, it will try to **connect immediately**
- Important methods:
  - Socket#getInputStream, Socket#getOutputStream
  - Socket#close

```
Socket mySocket = new Socket("130.89.253.64", 4444); // Will connect or throw an IOException
InputStream input = mySocket.getInputStream();
OutputStream output = mySocket.getOutputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
PrintWriter writer = new PrintWriter(new OutputStreamWriter(output))
writer.println("Hello World"); // Write a message to the output stream
writer.flush();  // Flushing it causes it to be sent to the server
System.out.println(reader.readLine()); // We can also get messages from the server
```

SERVER SOCKETS

- A ServerSocket **listens** for new connections on a given port
- Important methods:
  - ServerSocket#accept returns a Socket for a new connection, call this in a **loop**
  - ServerSocket#close closes the ServerSocket, all **existing connections remain open**

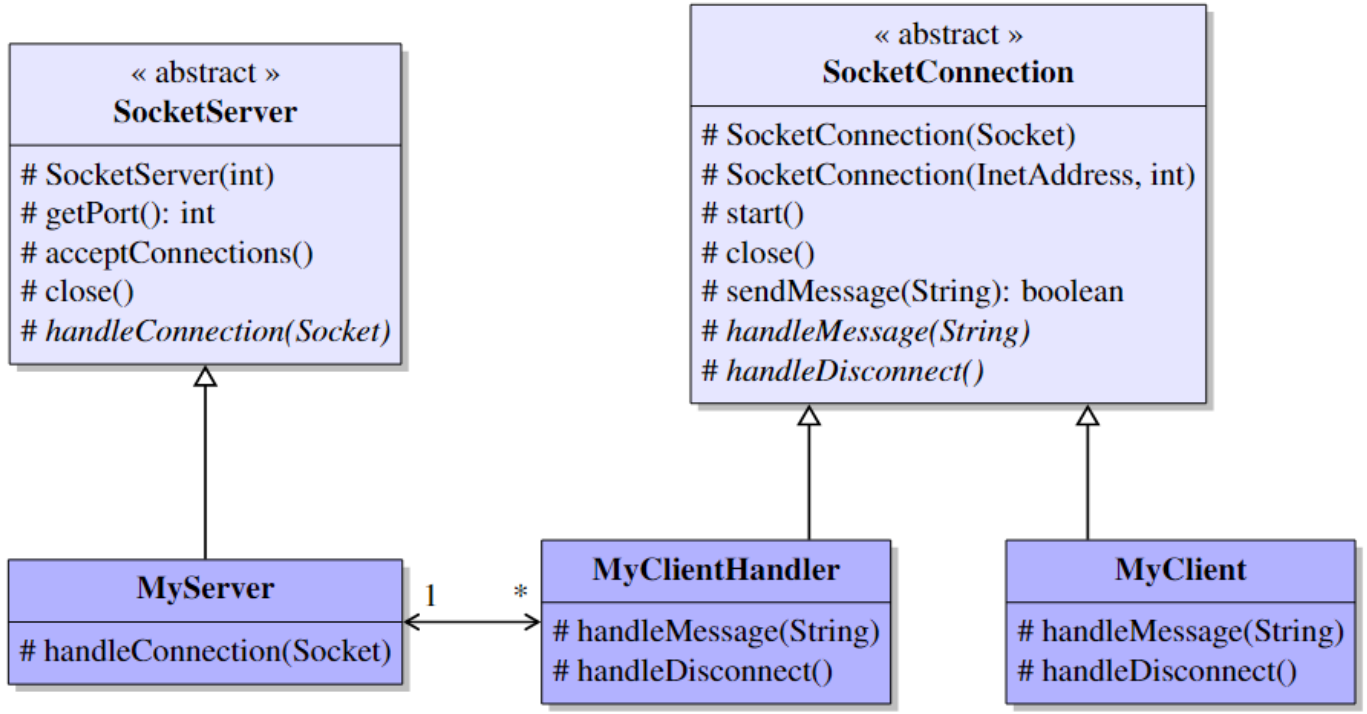
```
ServerSocket myServerSocket = new ServerSocket(4444);
Socket mySocket = myServerSocket.accept();
InputStream input = mySocket.getInputStream();
OutputStream output = mySocket.getOutputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
PrintWriter writer = new PrintWriter(new OutputStreamWriter(output))
writer.println("Hello World"); // Write a message to the output stream
writer.flush();  // Flushing it causes it to be sent to the client
System.out.println(reader.readLine()); // We can also get messages from the client
```

DETECTING CONNECTION LOSS

- Detecting a lost connection is difficult, there is **no guarantee** that you get notified
- Three different scenarios can occur
  - The connection is lost, then you attempt to write to the OutputStream → IOException
  - The connection is lost, then you attempt to read from the InputStream → IOException
  - You are reading from the InputStream when the connection is lost → Returns null
- You'll learn more about TCP/IP in module 3

WHAT WE PROVIDE

- We provide:
  - The abstract class SocketConnection
  - The abstract class SocketServer
- You write:
  - A class extending SocketServer with an implementation of **handleConnection**
  - Two classes extending SocketConnection with implementations of **handleMessage** and **handleDisconnect**



EXTENDING AND USING SOCKETSERVER

```
public class MyServer extends SocketServer {
  public MyServer(int port) throws IOException { super(port); }
  @Override
  protected void handleConnection(Socket socket) {
    try {
      ClientHandler handler = new ClientHandler(socket);
      handler.start(); // Starts a new thread to handle messages from the client
    } catch (IOException e) {
      …
    }
  }
  public static void main(String[] args) {
    MyServer server = new MyServer(4444);
    try {
      server.acceptConnections(); // This calls ServerSocket#accept in a loop
    } catch (IOException e) {
      …
    }
  }
}
```

IMPLEMENTING A CLIENT HANDLER

```
public class ClientHandler extends SocketConnection {
  public ClientHandler(Socket socket) throws IOException { super(socket); }
  @Override
  protected void handleMessage(String message) {
    System.out.println(message);
    sendMessage(message);
  }
  @Override
  protected void handleDisconnect() {
    // Called when we know the connection is lost
    System.out.println("Lost connection with client!");
  }
  @Override
  public void handleStart() {
    // Runs after the call to ClientHandler#start, but before we wait for messages
    sendMessage("Connected to the server");
  }
}
```

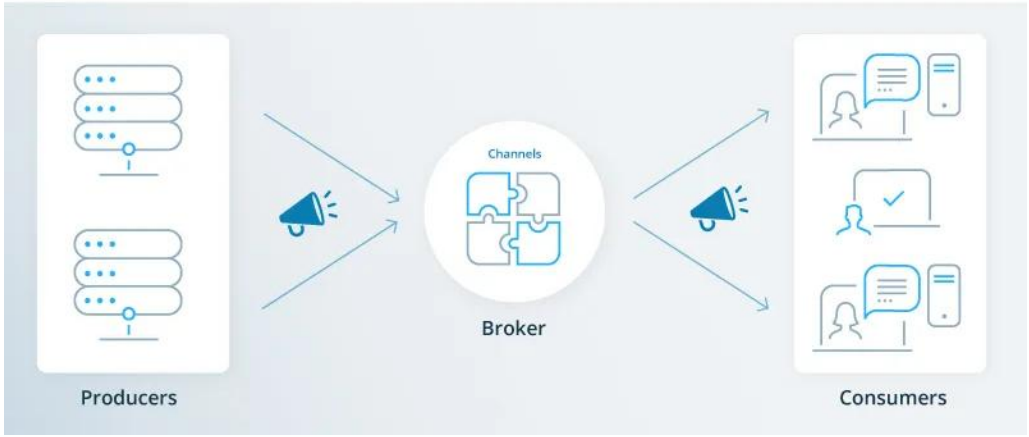
IMPLEMENTING A CLIENT

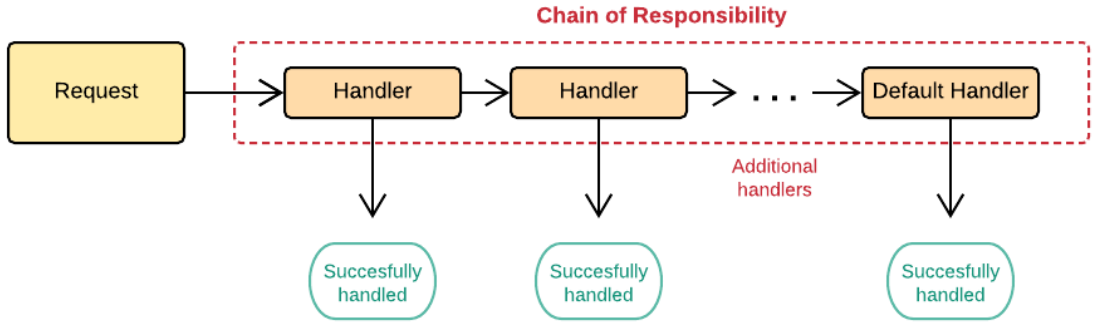
```
public class MyClient extends SocketConnection {
  private boolean connectionLost = false;
  public MyClient(String host, int port) throws IOException { super(new Socket(host, port)); }
  @Override
  protected void handleMessage(String message) {
    System.out.println(message);
  }
  @Override
  protected void handleDisconnect() { connectionLost = true; }
  public static void main(String[] args) {
    try {
      MyClient client = new MyClient("localhost", 4444); // Establish connection
      client.start(); // Start a thread waiting for messages from the server
      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
      while (!client.connectionLost) {
        client.sendMessage(reader.readLine()); // Send a message to the server
      }
    } catch (IOException e) { … }
  }
}
```

HOW TO DESIGN YOUR SERVER

- Main class + Client handlers
- How many threads do you need?
- What design patterns are applicable?
  - Event-driven architecture
  - Chain of responsibility

- Alternatives: Spring Boot, Java NIO
- Advanced features: logging, user input

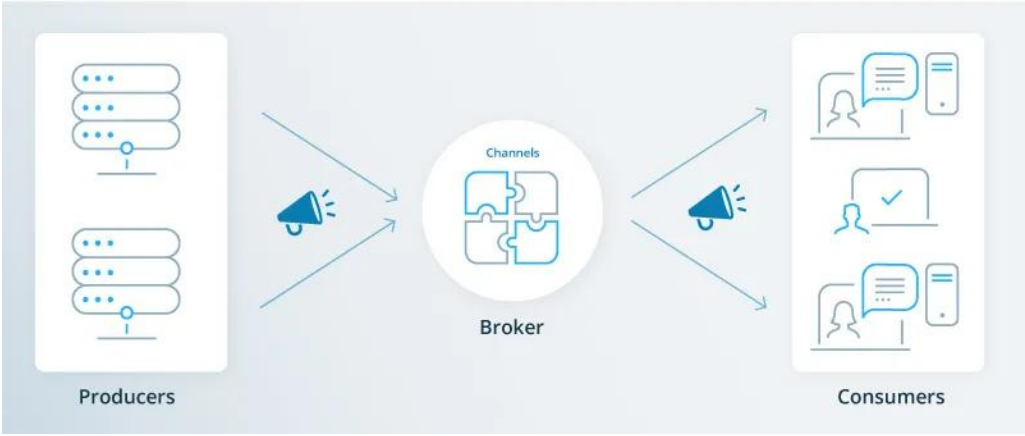




HOW TO DESIGN YOUR CLIENT

- After making a connection, how many threads do you need?
- What design patterns are applicable?
  - Event-driven architecture
  - Listeners

- Where to put the protocol implementation
  - Part of the connection handler, or
  - A separate class



CONCURRENCY

- The client and the server are **multi-threaded** applications
- What happens on the client when the user types a message and a command is received from the server at the same time?
- What happens when two client messages are received at the same time?

- You need **locks**, **queues**, or other synchronisation methods to avoid race conditions

PARSING

Input.txt:

```
Apples   10      €1
Bananas  2       €0.5
Pears    5       $2
```

- Parsing is the act of splitting up some input into chunks of data that your program can use or store
- Java has several utilities for parsing text
  - BufferedReader#**readLine**
  - String#**split**
  - Type.**parseType** (Integer.parseInt, Long.parseLong, etc.)
  - String#**substring**
  - Regular expressions (**Patterns** and **Matchers**)
  - The **Scanner** class

```
Scanner scanner = new Scanner(new File("Input.txt"));
List<Product> products = new ArrayList<>();
while (scanner.hasNext()) {
    String name = scanner.next("[A-za-z]+");
    int amount = scanner.nextInt(),
    scanner.skip("[$€]");
    float price = scanner.nextFloat();
    products.add(new Product(name, amount, price));
    scanner.nextLine(); // This normally returns the rest of the line
}
}
}
```

DEFENSIVE PROGRAMMING

- A program that receives input may also receive **invalid input**
- Defensive programming is **anticipating** what may go wrong and adding code to deal with those scenarios
- In the programming project, what can go wrong?
  - The user enters an invalid move
  - An invalid move is received over the network
  - The user enters text where we expect a port number
  - A message that does not conform to the protocol is received over the network
    - For example: the message is empty, the message contains too few parts, the message arrives at an unexpected time
  - The connection is lost
  - Many more…
- How can we deal with these cases
  - Add **checks** for invalid input (e.g. is the index is in bounds, is the message is long enough, is the tile already occupied)
  - Add **try-catch** to catch exceptions (including unchecked exceptions like NumberFormatException)
  - What about a **catch-all** try-catch for the entire program? **Usually a bad idea, but there are use cases**

SUMMARY

- Implementing a **client** and **server**
  - Socket and ServerSocket (**for the project** use the SocketConnection and SocketServer classes we provide)
  - **Client**: one thread waits for user input, one thread waits for server messages, both threads can send
  - **Server**: one thread waits for new connections, one thread waiting for client messages for each client
  - Both have an **event-driven architecture**

- Parsing
  - Simple: String#split, String#substring, BufferedReader#readLine, Type.parseType
  - More advanced: Regular expressions, Scanner

- Defensive programming
  - **Anticipate** what can go wrong
  - Check input and catch exceptions

Any questions?