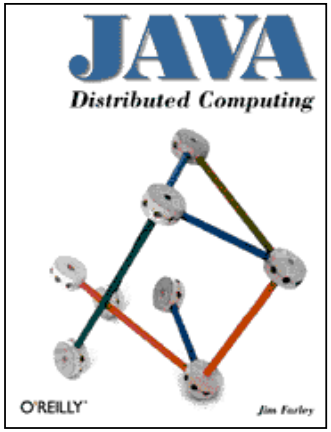
# Client-Server mit Sockets in Java

- Beispiel aus dem Buch *Java Distributed Computing* von Jim Farley (O'Reilly, 1998, ISBN 1-56592-206-9)




- Hier der Client:

```
import java.lang.*;
import java.net.*;
import java.io.*;

public class SimpleClient
{
  // Our socket connection to the server
  protected Socket serverConn;

    public SimpleClient(String host, int port)
      throws IllegalArgumentException {
    try {
      System.out.println("Trying to connect to "
      + host + " " + port);
      serverConn = new Socket(host, port);
    }
    catch (UnknownHostException e) {
      throw new IllegalArgumentException
        ("Bad host name given.");
    }
    catch (IOException e) {
      System.out.println("SimpleClient: " + e);
      System.exit(1);
    }

    System.out.println("Made server connection.");
  }
```

Konstruktor



```
public static void main(String argv[]) {
  if (argv.length < 2) {
    System.out.println ("Usage: java \
       SimpleClient <host> <port>");
    System.exit(1);
  }

  int port = 3000;
  String host = argv[0];
  try { port = Integer.parseInt(argv[1]); }
  catch (NumberFormatException e) {}

  SimpleClient client  =  new SimpleClient(host, port);
  client.sendCommands();
}

public void sendCommands() {
  try {
    DataOutputStream dout =
      new DataOutputStream(serverConn.getOutputStream());
    DataInputStream din =
      new DataInputStream(serverConn.getInputStream());

    // Send a GET command...
    dout.writeChars("GET goodies ");
    // ...and receive the results
    String result = din.readLine();
    System.out.println("Server says: \"" + result + "\"");
  }
  catch (IOException e) {
    System.out.println("Communication SimpleClient: " + e);
    System.exit(1);
  }
}

public synchronized void finalize() {
  System.out.println("Closing down SimpleClient...");
  try { serverConn.close(); }
  catch (IOException e) {
    System.out.println("Close SimpleClient: " + e);
    System.exit(1);
  }
}
```

Host- und Port-nummer von der Kommandozeile

Wird vom Garbage-Collector aufgerufen, wenn keine Referenzen auf den Client mehr existieren ('close' ggf. am Ende von 'sendCommands')

# Der Server

```
import java.net.*;
import java.io.*;
import java.lang.*;

public class SimpleServer {
  protected int portNo = 3000;
  protected ServerSocket clientConnect;

  public SimpleServer(int port) throws
      IllegalArgumentException {
    if (port <= 0)
      throw new IllegalArgumentException(
       "Bad port number given to SimpleServer constructor.");

    // Try making a ServerSocket to the given port
    System.out.println("Connecting server socket to port");
    try { clientConnect = new ServerSocket(port); }
    catch (IOException e) {
     System.out.println("Failed to connect to port " + port);
      System.exit(1);
    }

    // Made the connection, so set the local port number
    this.portNo = port;
  }

  public static void main(String argv[]) {
    int port = 3000;
    if (argv.length > 0) {
      int tmp = port;
      try {
        tmp = Integer.parseInt(argv[0]);
      }
      catch (NumberFormatException e) {}
      port = tmp;
    }

    SimpleServer server = new SimpleServer(port);
    System.out.println("SimpleServer running on port " +
       port + "...");
    server.listen();
  }
```

Default-Port, an dem der Server auf eine Client-Verbindung wartet

Socket, der Verbindungs-wünsche entgegennimmt

Konstruktor

Portnummer von Kommandozeile

Aufruf der Methode "listen" (siehe unten)



```
public void listen() {
    try {
      System.out.println("Waiting for clients...");
      while (true) {
        Socket clientReq = clientConnect.accept();
        System.out.println("Got a client...");
        serviceClient(clientReq);
      }
    }
    catch (IOException e) {
      System.out.println("IO exception while listening.");
      System.exit(1);
    }
  }

  public void serviceClient(Socket clientConn) {
    SimpleCmdInputStream inStream = null;
    DataOutputStream outStream = null;
    try {
      inStream = new SimpleCmdInputStream
        (clientConn.getInputStream());
      outStream = new DataOutputStream
        (clientConn.getOutputStream());
    }
    catch (IOException e) {
      System.out.println("SimpleServer: I/O error.");
    }
    SimpleCmd cmd = null;
    System.out.println("Attempting to read commands...");
    while (cmd == null || !(cmd instanceOf DoneCmd)) {
      try { cmd = inStream.readCommand(); }
      catch (IOException e) {
        System.out.println("SimpleServer (read): " + e);
        System.exit(1);
      }

      if (cmd != null) {
        String result = cmd.Do();
        try { outStream.writeBytes(result); }
        catch (IOException e) {
          System.out.println("SimpleServer (write): " + e);
          System.exit(1);
        }
      }
    }
  }
}
```

Warten auf connect eines Client, dann Gründen eines Sockets

Von DataInput-Stream abge-leitete Klasse

Klasse SimpleCmd hier nicht gezeigt

Schleife zur Entgegennahme und Ausführung von Kommandos

finalize-Methode hier nicht gezeigt

# Java als “Internet-Programmiersprache”

- Java hat eine Reihe von Konzepten, die die Realisierung verteilter Anwendungen erleichtern, z.B.:
  - Socket-Bibliothek zusammen mit Input- / Output-Streams
  - Remote Method Invocation (RMI): Entfernter Methodenaufruf mit Transport (und dabei Serialisierung) auch komplexer Objekte
  - CORBA-APIs
  - eingebautes Thread-Konzept
  - java.security-Paket
  - plattformunabhängiger Bytecode mit Klassenlader (Java-Klassen können über das Netz transportiert und geladen werden; Bsp.: Applets)

Im Vergleich zu RPC: Nicht notw. Master-Slave, sondern peer-to-peer

Damit z.B. Realisierung eines “Meta-Protokolls”: Über einen Socket vom Server eine Klasse laden (und Objekt-Instanz gründen), was dann (auf Client-Seite) ein spezifisches Protokoll realisiert. (Vgl. “mobiler Code”, “mobile Agenten”, Jini...)

- Das UDP-Protokoll kann mit “Datagram-Sockets” verwendet werden, z.B. so:

```
try {
  DatagramSocket s = new DatagramSocket();
  byte[] data = {'H','e','l','l','o'};
  InetAddress addr = InetAddress.getByName("my.host.com");
  DatagramPacket p = new DatagramPacket(data,
    data.length, addr, 5000);
  s.send(p);
}
catch (Exception e) {
  System.out.println("Exception using datagrams:");
  e.printStackTrace();
}
```

Port-Nummer

  - entsprechend zu “send” gibt es ein “receive”
  - InetAddress-Klasse repräsentiert IP-Adressen
  - diese hat u.a. Methoden “getByName” (klassenbezogene Methode) und “getAddress” (instanzbezogene Methode)
  - UDP ist verbindungslos und unsicher (aber effizient)



# URL-Verbindungen in Java

- Java bietet einfache Möglichkeiten, auf “Ressourcen” (i.w. Dateien) im Internet mit dem HTTP-Protokoll lesend und schreibend zuzugreifen
  - falls auf diese mittels einer URL verwiesen wird
- Klasse “URL” in java.net.*
  - auf höherem Niveau als die Socket-Programmierung
  - Sockets (mit TCP) werden vom Anwender verborgen benutzt
- Beispiel: zeilenweises Lesen einer Textdatei
  - aber auch hier noch diverse Fehlerbedingungen abfangen!

```
// Objekt vom Typ URL anlegen:
URL myURL;
myURL = new URL("http", ..., "/Demo.txt");
...
DataInputStream instream;
instream = new DataInputStream(myURL.openStream());
String line = "";
while((line = instream.readLine()) != null)
  // line verarbeiten
...
```

hier Hostname angeben

Name der Datei

- Es gibt auch Möglichkeiten, Daten an eine URL zu senden (POST-Methode, z.B. an ein CGI-Skript)
- Ferner: Information über das Objekt ermitteln
  - z.B. Grösse, Kodierung, letztes Änderungsdatum, HTTP-Header etc.
- Es existiert eine ContentHandlerFactory, die den MIME-Typ prüft und einen ContentHandler dafür liefert
  - ContentHandler liest ein Objekt über eine URL-Verbindung und konstruiert lokal eine geeignete Objekt-Instanz

# Übungsbeispiel: Ein Bookmark-Checker

```
import java.io.*;
import java.net.*;
import java.util.Date;
import java.text.DateFormat;

public class CheckBookmark {

  public static void main (String args[]) throws
       java.io.IOException, java.text.ParseException {

    if (args.length != 2) System.exit(1);

    // Create a bookmark for checking...
    CheckBookmark bm = new CheckBookmark(args[0], args[1]);

    bm.checkit(); // ...and check

    switch (bm.state) {
      case CheckBookmark.OK:
        System.out.println("Local copy of " +
           bm.url_string + " is up to date"); break;
      case CheckBookmark.AGED:
        System.out.println("Local copy of " +
           bm.url_string + " is aged"); break;
      case CheckBookmark.NOT_SUPPORTED:
        System.out.println("Webserver does not support \
           modification dates"); break;
      default: break;
    }
  }

  String url_string, chk_date;
  int state;

  public final static int OK = 0;
  public final static int AGED = 1;
  public final static int NOT_SUPPORTED = 2;

  CheckBookmark(String bm, String dtm) // Constructor
  { url_string = new String(bm);
    chk_date = new String(dtm);
    state = CheckBookmark.OK;
  }
```



```
  public void checkit() throws java.io.IOException,
       java.text.ParseException {

    URL checkURL = null;
    URLConnection checkURLC = null;

    try { checkURL = new URL(this.url_string); }
    catch (MalformedURLException e) {
      System.err.println(e.getMessage() + ": Cannot \
          create URL from " + this.url_string);
      return;
    }

    try {
      checkURLC = checkURL.openConnection();
      checkURLC.setIfModifiedSince(60);
      checkURLC.connect();
    }

    catch (java.io.IOException e) {
      System.err.println(e.getMessage() + ": Cannot \
          open connection to " + checkURL.toString());
      return;
    }

    // Check whether modification date is supported
    if (checkURLC.getLastModified() == 0) {
      this.state = CheckBookmark.NOT_SUPPORTED;
      return;
    }

    // Cast last modification date to a "Date"
    Date rem = new Date(checkURLC.getLastModified());

    // Cast stored date of bookmark to Date
    DateFormat df = DateFormat.getDateInstance();
    Date cur = df.parse(this.chk_date);

    // Compare and set flag for outdated bookmark
    if (cur.before(rem)) this.state = CheckBookmark.AGED;
  }
}
```

# Adressierung

- *Sender* muss in geeigneter Weise spezifizieren, wohin die Nachricht gesendet werden soll
  - ggf. mehrere Adressaten zur freien Auswahl (Lastverteilung, Fehlertoleranz)
  - ggf. mehrere Adressaten gleichzeitig (Broadcast, Multicast)

- *Empfänger* ist ggf. nicht bereit, jede beliebige Nachricht von jedem Sender zu akzeptieren
  - selektiver Empfang (Spezialisierung)
  - Sicherheitsaspekte, Überlastabwehr

- Probleme
  - *Ortstransparenz*: Sender weiss *wer*, aber nicht *wo* (sollte er i.a. auch nicht!)
  - *Anonymität*: Sender und Empfänger kennen einander nicht (sollen sie oft auch nicht)



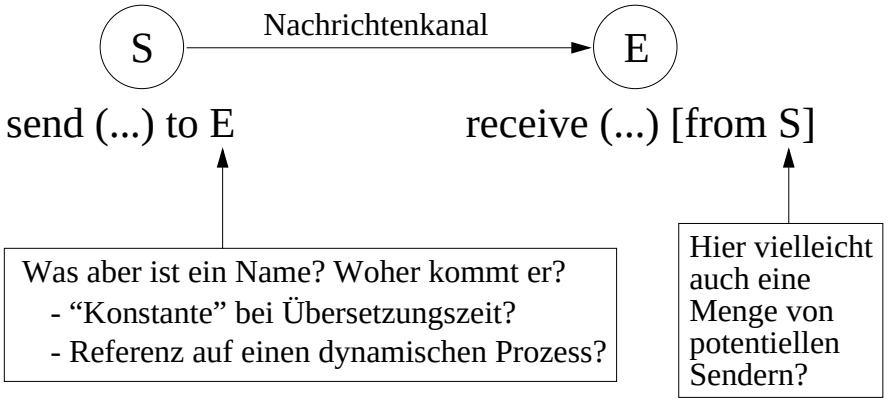
# Kenntnis von Adressen?

- Adressen sind u.a. Rechneradressen (z.B. IP-Adresse oder Netzadresse auf Ethernet-Basis), Portnamen, Socketnummern, Referenzen auf Mailboxes...

- Woher kennt ein Sender die Adresse des Empfängers?
  1) Fest in den Programmcode integriert --> unflexibel
  2) Über Parameter erhalten oder von anderen Prozessen mitgeteilt
  3) Adressanfrage per Broadcast “in das Netz”
     - häufig bei LANs: Suche nach lokalem Nameserver, Router etc.
  4) Auskunft fragen (Namensdienst wie z.B. DNS; Lookup-Service)
     - wie realisiert man dies effizient und fehlertolerant?

# Direkte Adressierung

- *Direct Naming* (1:1-Kommunikation):



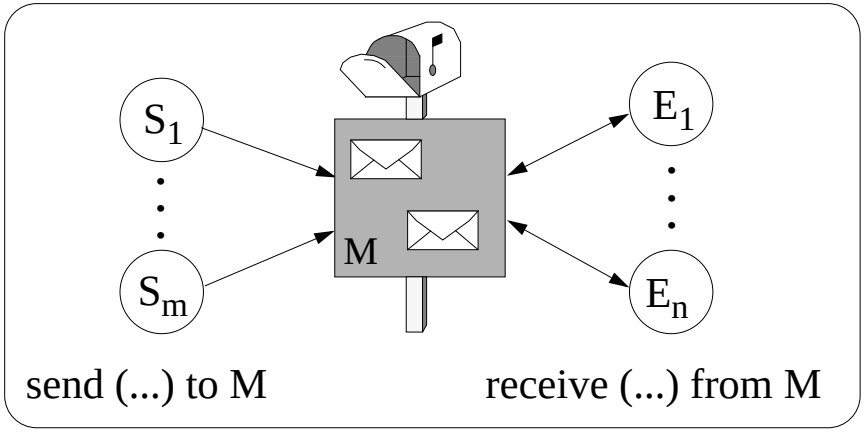

- Direct naming ist insgesamt relativ unflexibel

- Empfänger (= Server) sollten nicht gezwungen sein, potentielle Sender (= Client) explizit zu nennen
  - Symmetrie ist also i.a. gar nicht erwünscht



# Indirekte Adressierung - Mailbox

- m:n-Kommunikation möglich




- Eine Nachricht besitzt i.a. mehrere potentielle Empfänger
  - Mailbox spezifiziert damit eine *Gruppe* von Empfängern

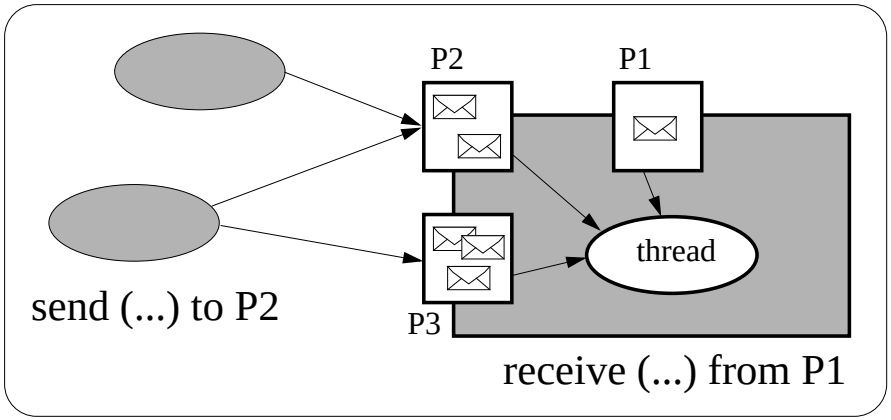
- Kann jeder Empfänger die Nachricht bearbeiten?
  - Mailbox i.a. typisiert: nimmt nur bestimmte Nachrichten auf
  - Empfänger kann sich u.U. Nachrichten der Mailbox ansehen / aussuchen...
  - aber wer garantiert, dass jede Nachricht irgendwann ausgewählt wird?

- Wo wird die Mailbox angesiedelt? (--> Implementierung)
  - als ein einziges Objekt auf irgendeinem (geeigneten) Rechner?
  - repliziert bei den Empfängern? Abstimmung unter den Empfängern notwendig (--> verteiltes Cache-Kohärenz-Problem)
  - Nachricht verbleibt in einem Ausgangspuffer des Senders: Empfänger müssen sich bei allen (welche sind das?) potentiellen Sendern erkundigen

- Mailbox muss gegründet werden: Wer? Wann? Wo?

# Indirekte Adressierung - Ports

- m:1-Kommunikation
- Ports sind Mailboxes mit genau einem Empfänger
  - Port gehört diesem Empfänger
  - Kommunikationsendpunkt, der die interne Struktur abkapselt
- Ein Objekt kann i.a. mehrere Ports besitzen



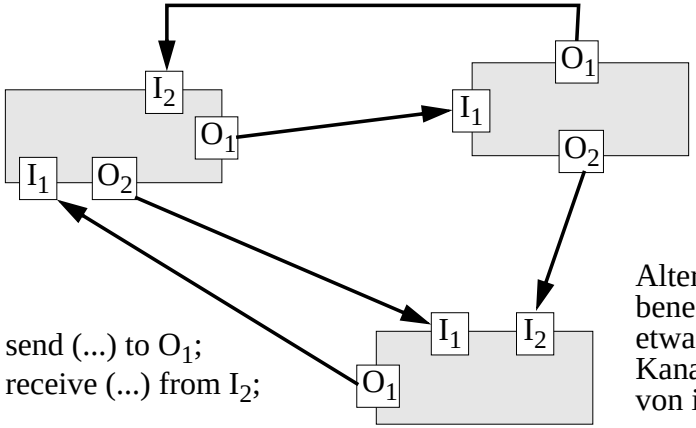

*Pragmatische Aspekte (Sprachdesign etc.):*

- Sind Ports statische oder dynamische Objekte?
- Wie erfährt ein Objekt den Portnamen eines anderen (dynamischen) Objektes?
  - können Namen von Ports verschickt werden?
- Sind Ports typisiert?
  - unterstützt den selektiven Nachrichtenempfang
- Grösse des Nachrichtenpuffers?
- Können Ports geöffnet und geschlossen werden?
  - genaue Semantik?



# Kanäle und Verbindungen

- Neben *Eingangsports* (“in-port”) lassen sich auch *Ausgangsports* (“out-port”) betrachten




Alternativ: Kanäle benennen und etwas auf den Kanal senden bzw. von ihm lesen

- Ports können als Ausgangspunkte für das Einrichten von *Verbindungen* (“Kanäle”) gewählt werden
- Dazu werden je zwei in- / out-Ports miteinander verbunden. Dies kann z.B. mit einer connect-Anweisung geschehen: **connect** p1 **to** p2
  - denkbar sind auch broadcastfähige Kanäle
- Die Programmierung und Instanziierung eines Objektes findet so in einer anderen Phase statt als die Festlegung der Verbindungen

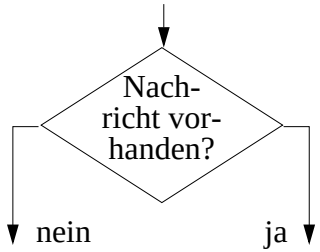
Konfigura-tionsphase

- Grössere Flexibilität durch die dynamische Änderung der Verbindungsstruktur
  - dadurch auch Anonymisierung von Sender und Adressat
- Kommunikationsbeziehung: wahlweise 1:1, n:1, 1:n, n:m

# Varianten beim Empfangen von Nachrichten - Nichtblockierung

- Typischerweise ist ein “receive” blockierend

- Aber auch nichtblockierender Empfang denkbar:




- “Non-blocking receive”

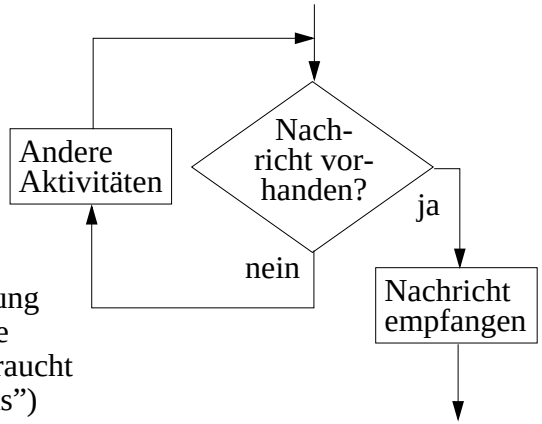
- Sprachliche Realisierung z.B. durch “Returncode” eines als Funktions-aufruf benutzten “receive”

---

- Aktives Warten:
  (“busy waiting”)

  - Nachbildung des blockierenden Wartens wenn “andere Aktivi-täten” leer

  - Nur für kurze Wartezeiten sinnvoll, da Monopolisierung der cpu, die ggf. für andere Prozesse oder threads gebraucht werden könnte (“spin locks”)




---

- Weitere Möglichkeit: unterbrechungsgesteuertes (“asynchrones”) Empfangen der Nachricht (--> nicht unproblematisch!)



# Nichtblockierendes, alternatives Empfangen

- Sprachliche Realisierung z.B. so:

**select**

guard$_1$ --> **receive**
Statements

⋮

guard$_n$ --> **receive**
Statements

**[ else** . . . ]

**endselect**

else-Fall wird gewählt, wenn kein guard ‘true’ ist bzw. dort momentan keine Nachricht anliegt

- Aktives Warten durch umschliessende while-Schleife

  - im else-Fall könnte dann die while-Bedingung auf false gesetzt werden, falls das aktive Warten abgebrochen werden soll, oder es könnte mittels timer (“wait”) eine kurze Zeit gewartet werden...

  - else-Fall kann auch einfach das leere Statement enthalten

- Typischerweise blockierend, wenn else-Alternative ganz fehlt

# Zeitüberwachter Nachrichtenempfang

- Empfangsanweisung soll maximal (?) eine gewisse Zeit lang blockieren (“timeout”)
  - z.B. über return-Wert abfragen, ob Kommunikation geklappt hat

- Sinnvoll bei:
  - Echtzeitprogrammierung
  - Vermeidung von Blockaden im Fehlerfall (etwa: abgestürzter Kommunikationspartner)

    --> dann sinnvolle Recovery-Massnahmen treffen (“exception”)

    --> timeout-Wert “sinnvoll” setzen!

    Quelle vielfältiger Probleme...

- Timeout-Wert = 0 kann ggf. genutzt werden, um zu testen, ob eine Nachricht “jetzt” da ist

---

- Analog ggf. auch für synchrones (!) *Senden* sinnvoll

  --> Verkompliziert zugrundeliegendes Protokoll: Implizite Acknowledgements kommen nun “asynchron” an...



# Zeitüberwachter Empfang

- Möglicher Realisierung:
  - Durch einen Timer einen *asynchronen Interrupt* aufsetzen und Sprungziel benennen
  - Sprungziel könnte z.B. eine Unterbrechungsroutine sein, die in einem eigenen Kontext ausgeführt wird, oder das Statement nach dem receive

  --> “systemnahe”, unstrukturierte, fehleranfällige Lösung; schlechter Programmierstil!

---

- Sprachliche Einbindung besser z.B. so:

Vorsicht!

**receive** ... **delay** t ← Blockiert maximal t Zeiteinheiten

**select**
  guard$_1$ --> **receive** ...
  ⋮
  **delay** t --> Anweisungen ...
**endselect**

Wird nach *mind*. t Zeiteinheiten ausgeführt, wenn bis dahin noch keine Nachricht empfangen

- Genaue Semantik beachten: Es wird *mindestens* so lange auf Kommunikation gewartet. Danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Programms verstreichen!

- Frage: ist “delay 0” äquivalent zu “else”?