## JAVA E LA RETE

**L'architettura Java è *network-ready***

- **Package `java.net`**
  **Concetti (classi) fondamentali:**
  - **`Socket`, `ServerSocket`**
  - **`URL`, `AudioClip`, `Image`**
- **Package `java.rmi`**
  **Uso di *oggetti remoti***
- **...**

## URL e CONNESSIONI

- **La classe `URL` cattura il concetto di *indirizzo Internet (URL)* nella forma standard:**
  - `http://localhost/index.html`
  - `file:///autoexec.bat`
  - `...`
- **Un *oggetto URL* si crea a partire dall'indirizzo che rappresenta:**

```
URL url = new URL("....");
```

**e si usa per *aprire una connessione* verso tale indirizzo.**

## URL e CONNESSIONI

- **Per aprire una connessione, si invoca sull'oggetto `URL` il metodo `openConnection()`:**

```
URLConnection c =
            url.openConnection();
```

- **Il risultato è un oggetto `URLConnection`, che rappresenta una "connessione aperta"**
  - **in pratica, così facendo si è stabilito un canale di comunicazione verso l'indirizzo richiesto**
- **Per connettersi tramite tale connessione:**

```
c.connect();
```

## COMUNICAZIONE

**Per comunicare *si recuperano dalla connessione i due stream* (di ingresso e di uscita) a essa associati, tramite i metodi:**

- `public InputStream getInputStream()`
  - restituisce lo stream di input da cui leggere i dati (byte) che giungono dall'altra parte
- `public OutputStream getOutputStream()`
  - restituisce lo stream di output su cui scrivere i dati (byte) da inviare all'altra parte

**Poi, su questi stream si legge / scrive *come su qualunque altro stream di byte.***

## ESEMPIO CON URL

**Connettersi all'URL dato e, *nell'ipotesi che esso invii testo*, visualizzarne il contenuto**

```
import java.io.*; import java.net.*;
class EsempioURL {
 public static void main(String args[]){
  URL u = null;
  try {
     u = new URL(args[0]);
  } catch (MalformedURLException e) {
    System.err.println("URL errato: " + u);
  }
  ...
```

Indirizzo passato dalla riga di comando

## ESEMPIO CON URL

```
...
URLConnection c = null;
try {
 System.out.print("Connecting...");
 c = u.openConnection(); c.connect();
 System.out.println("..OK");
 InputStreamReader is = new
   InputStreamReader(c.getInputStream());
 BufferedReader r = new BufferedReader(is);
 System.out.println("Reading data...");
 ...
```

## ESEMPIO CON URL

```
  ...
    String line = null;
    while((line=r.readLine())!=null)
      System.out.println(line);
  } catch (IOException e) {
    System.err.println(e);
  }
 }
}
```

Esempio d'uso:

```
D:\esercizi>java EsempioURL file:///Prova.txt
Connecting.....OK
Reading data...
3.1415 true 1.4142 X 12
```

## ESEMPIO CON URL

Un altro esempio d'uso *(lettura di un file HTML)*

```
D:\esercizi>java EsempioURL file:///Prova.html
Connecting.....OK
Reading data...
<TITLE> Esempio di form </TITLE>
<H1> Esempio di form </H1>
<FORM METHOD="POST"
ACTION="http://localhost/cgi/prova.exe">
Inserisci il testo: <INPUT NAME="testo">
e poi premi invio: <INPUT TYPE="submit"
VALUE="invio">
</FORM>
```

## ESEMPIO CON URL

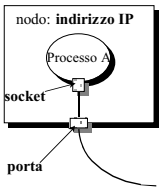
Un altro esempio d'uso *(lettura di un file HTML)*

```
D:\esercizi>java EsempioURL file:///Prova.html
Connecting.....OK
Reading data...
<TITLE> Esempio di form </TITLE>
<H1> Esempio di form </H1>
<FORM METHOD="POST"
ACTION="http://localhost/cgi/prova.exe">
Inserisci il testo: <INPUT NAME="testo">
e poi premi invio: <INPUT TYPE="submit"
VALUE="invio">
</FORM>
```

**Se avessimo uno schermo grafico, su cui poter mostrare font, colori, stili... *questo sarebbe un browser Internet***

## IL CONCETTO DI SOCKET

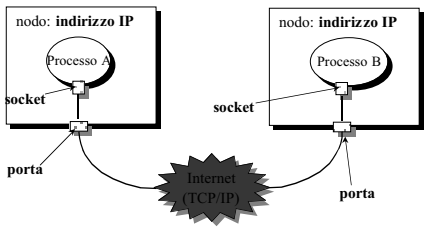
- **Una *socket* è concettualmente una *porta*, una "*presa*" verso la rete**
- **Collega un certo *processo* al mondo esterno**
- **È identificata da un *numero (port number)* unico *su una data macchina* (*nodo* o *host*)**
- **Ogni nodo è identificato dal suo *indirizzo IP***




## COMUNICAZIONE VIA SOCKET

- **La socket è un *canale di comunicazione***
- **Permette a due processi, *residenti sulla stessa macchina o anche molto distanti,* di comunicare fra loro *nello stesso modo***
- **Modello cliente / servitore:**
  - **il servitore deve *stare in attesa* di possibili comunicazioni in arrivo *(ente passivo)***
  - **i clienti (anche più di uno), quando vogliono, *parlano* con il servitore *(enti attivi)***

## COMUNICAZIONE VIA SOCKET



## COMUNICAZIONE VIA SOCKET

- **Esistono fondamentalmente *due tipi di socket:* socket stream e socket datagram**
- **Le socket stream**
  - **sono *affidabili,* stabiliscono una connessione *stabile* e *bidirezionale* con l'altra parte, che dura finché non si decide di chiuderla**
- **Le socket datagram**
  - ***non* sono affidabili, *non* stabiliscono una connessione stabile: la comunicazione è *unidirezionale* come un telegramma**
  - **ma sono *meno costose***

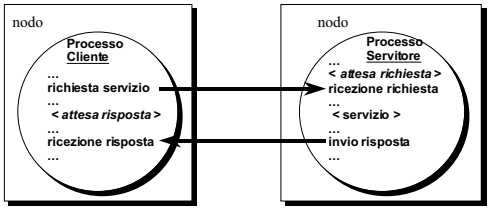
## COMUNIC

- **Esistono fon *socket:* sock**
- **Le socket stream**
  - **sono *affidabili,* stabiliscono una connessione *stabile* e *bidirezionale* con l'altra parte, che dura finché non si decide di chiuderla**
- **Le socket datagram**
  - ***non* sono *non* stabiliscono una**
  - **r**

- Usare quando l'ordine dei messaggi è importante e l'affidabilità è cruciale
- Spesso c'è un limite massimo alle connessioni che si possono aprire

- Usare quando le prestazioni sono fondamentali e/o ci vorrebbero troppe connessioni aperte insieme
- Non devono esserci problemi se i messaggi arrivano in ordine qualunque

## COMUNICAZIONE: LO SCHEMA



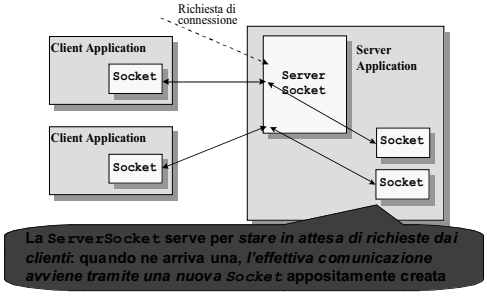

## SCHEMA CON SOCKET STREAM

1) **Il servitore crea la sua `ServerSocket` con un *numero noto,* e si mette in attesa**
2) **Un cliente, quando vuole comunicare col servitore, crea la sua `Socket` *specificando con chi vuole parlare***
   - **nome dell'host** dove risiede il servitore
   - **numero di porta** su cui è in ascolto il servitore
3) **Il servitore accetta la richiesta del cliente: con ciò si crea una `Socket` *già collegata al cliente,* tramite cui i due comunicano.**

## SCHEMA CON SOCKET STREAM

- **Alla `Socket` sono associati *due stream*:**
  - **uno *dal cliente verso il servitore***
  - **uno *dal servitore verso il cliente***
- **La comunicazione cliente/servitore è *bidirezionale***
  - **i ruoli "cliente" e "servitore" *sono tali solo nella fase iniziale*, quando si instaura la connessione**
  - **una volta connessi, i due processi possono parlarsi reciprocamente "alla pari"**

## SCHEMA CON SOCKET STREAM



## LA CLASSE `Socket`

**Costruire una `Socket` significa *aprire la comunicazione* verso l'altra parte**

- **`public Socket(InetAddress remoteAddr, int remotePort)`**
  - Crea una socket stream e la collega alla porta specificata della macchina remota corrispondente all'indirizzo IP dato
- **`public Socket(String remoteHost, int remotePort)`**
  - Crea una socket stream e la collega alla porta specificata della macchina remota corrispondente al nome dato

**Esempio**

```
Socket s = new Socket("mypc.unibo.it",13);
```

## LA CLASSE `Socket`

**Alcuni metodi utili**

- **`public InetAddress getInetAddress()`**
  - restituisce l'indirizzo della macchina remota a cui la socket è connessa
- **`public InetAddress getLocalAddress()`**
  - restituisce l'indirizzo della macchina locale
- **`public int getPort()`**
  - restituisce il numero di porta sulla macchina remota a cui la socket è connessa
- **`public int getLocalPort()`**
  - restituisce il numero di porta sulla macchina locale a cui la socket è legata

## LA CLASSE `Socket`

**Per comunicare, *si recuperano dalla socket i due stream* (di ingresso e di uscita) a essa associati, tramite i metodi:**

- **`public InputStream getInputStream()`**
  - restituisce lo stream di input da cui leggere i dati (byte) che giungono dall'altra parte
- **`public OutputStream getOutputStream()`**
  - restituisce lo stream di output su cui scrivere i dati (byte) da inviare all'altra parte

**Poi, su questi stream si legge / scrive *come su qualunque altro stream di byte.***

## LA CLASSE `Socket`

**Al termine, *per chiudere la comunicazione* si chiude la `Socket` col metodo:**

- **`public synchronized void close()`**
  - chiude la connessione e libera la risorsa

La parola chiave **`synchronized`** *non è rilevante* in questa sede.

## SERVIZI STANDARD

**Moltissimi servizi di uso comune sono *standardizzati* e disponibili su macchine sia Unix sia (non sempre) Windows**

- **echo (porta 7):** rimanda indietro tutto quello che gli si invia, come un'eco
- **daytime (porta 13):** restituisce data e ora
- **telnet (porta 23):** consente il collegamento remoto, da altro terminale (solo Unix)
- **smtp (porta 25):** consente la spedizione della posta (se abilitata)

## ESEMPIO 1

**Un cliente che si connette a una macchina remota sulla porta di daytime (porta 13), recupera data e ora, e li visualizza.**

```
import java.net.*;
import java.io.*;
public class EsempioNet1 {
 public static void main(String args[]){
   // creazione socket verso porta 13
   // recupero stream di input
   // lettura del messaggio dallo stream
   // stampa a video del messaggio
 }
}
```

## ESEMPIO 1

```
import java.net.*;
import java.io.*;
public class EsempioNet1 {
 public static void main(String args[]){
  Socket s = null;
  try { s = new Socket(…,13);
     InputStream is = s.getInputStream();
     InputStreamReader ir =
       new InputStreamReader(is);
     BufferedReader r =
       new BufferedReader(ir);
     ...
```

## ESEMPIO 1

```
     ...
     String line = r.readLine();
     System.out.println(line);
     s.close();
     } catch (UnknownHostException e){
        System.err.println("Host unknown");
     } catch (Exception e){
        System.err.println(e);
     }
  }
}
```

Risultato:
`Fri Feb 18 16:44:46 2000`

## ESEMPIO 2

**Un cliente che si connette a una macchina remota sulla porta di echo (porta 7), le invia un messaggio, e stampa ciò che riceve.**

```
import java.net.*;
import java.io.*;
public class EsempioNet2 {
 public static void main(String args[]){
  // creazione socket verso porta 7
  // recupero stream di input e di output
  // invio messaggio su stream di output
  // lettura e stampa da stream di input
 }
}
```

## ESEMPIO 2

```
import java.net.*;
import java.io.*;
public class EsempioNet2 {
 public static void main(String args[]){
  Socket s = null;
  try { s = new Socket(…,7);
     InputStream is = s.getInputStream();
     InputStreamReader ir =
       new InputStreamReader(is);
     BufferedReader r =
       new BufferedReader(ir);
     ...
```

## ESEMPIO 2

```
   ...
 OutputStream os = s.getOutputStream();
 OutputStreamWriter or =
        new OutputStreamWriter(os);
 BufferedWriter outr =
        new BufferedWriter(or);
 String msg = "Hello, I'm Donald Duck!";
 outr.write(message, 0, message.length());
 outr.newLine();
 outr.flush();
...
```

**Essenziale**:
- inviare il fine linea
- svuotare il buffer di uscita

## ESEMPIO 2

```
     ...
     String line = inr.readLine();
     System.out.println("Ricevuto: ");
     System.out.println(line);
     s.close();
     } catch (UnknownHostException e){
        System.err.println("Host unknown");
     } catch (Exception e){
        System.err.println(e);
     }
  }
}
```

Risultato:
`Hello, I'm Donald Duck!`

## UN SERVITORE

**Un servitore deve creare la propria `ServerSocket` *su una porta predeterminata***

- **il numero di porta del server *deve essere noto ai clienti* perché ne avranno bisogno per connettersi al servitore**
- **per i servizi standard, i numeri di porta sono standardizzati**
- **per i nostri servizi, possiamo scegliere un numero qualunque, purché libero**

## LA CLASSE `ServerSocket`

**Costruire una `ServerSocket` significa *predisporsi a ricevere richieste di connessione* da clienti remoti.**

- **`public ServerSocket(int localPort)`**
  - Crea una ServerSocket e la collega alla porta locale specificata
- **`public ServerSocket(int localPort, int count)`**
  - Crea una ServerSocket che accetta al massimo count richieste pendenti, e la collega alla porta locale specificata

**Esempio**

```
ServerSocket s = new ServerSocket(13000);
```

## LA CLASSE `ServerSocket`

**Il metodo principale**

- **`public Socket accept()`**
  - mette il servitore in attesa di nuove richieste di connessione: se non ce ne sono, il servitore si blocca in attesa
  - **restituisce un oggetto `Socket`, tramite cui avviene l'effettiva comunicazione tra cliente e servitore**

**Altri metodi utili**

- **`public InetAddress getInetAddress()`**
  - restituisce l'indirizzo della macchina locale
- **`public int getLocalPort()`**
  - restituisce il numero di porta sulla macchina locale a cui la socket è legata

## LA CLASSE `ServerSocket`

**Il metodo principale**

- **`public Socket accept()`**
  - mette il servitore in attesa di nuove richieste di connessione: se non ce ne sono, il servitore si blocca in attesa
  - **restituisce un oggetto `Socket`, tramite cui avviene l'effettiva comunicazione tra cliente e servitore**

**Altri metodi**

- **`pub`**
  -
- **`pub`**
  - restituisce il numero di porta sulla macchina locale a cui la socket è legata

**D'ora in poi, anche il servitore comunica col cliente tramite una normale `Socket`**
**Valgono quindi gli stessi mezzi visti poc'anzi.**

## TIPICO SCHEMA DI SERVER

```
...
ServerSocket ss = new ServerSocket(porta);
try {
 while (true) {
  Socket clientSock = ss.accept();
  // clientSock è la socket tramite cui
  // si parla col cliente
     ...comunicazione col cliente...
  clientSock.close();
 }
} catch (Exception e) { ... }
...
```

## TIPICO SCHEMA DI SERVER

```
...
Serv
try {
 while (true) {
  Socket clientSock = ss.accept();
  // clientSock è la socket tramite cui
  // si parla col cliente
     ...comunicazione col cliente...
  clientSock.close();
 }
} catch (Exception e) { ... }
...
```

**Una volta attivato, il server non termina mai: svolge *per costruzione* un ciclo infinito!**
**Per fermarlo: CTRL+C**

## ESEMPIO DI SERVER

**Un nostro servitore daytime sulla porta 7777 (rimanda indietro la data corrente)**

```
import …;
public class EsempioServer {
 public static void main(String args[]){
   // creazione ServerSocket su porta 7777
   // ciclo senza fine:
   //  attesa connessione,
   //  recupero socket cliente,
   //  recupero stream e comunicazione
   //  chiusura socket cliente
 }
}
```

## ESEMPIO DI SERVER

```
import java.net.*;
import java.io.*;
public class EsempioServer {
 public static void main(String args[]){
  ServerSocket ss = null;
  try {
   ss = new ServerSocket(7777);
   while (true) {
     Socket clientSock = ss.accept();
     OutputStream os =
           clientSock.getOutputStream();
    ...
```

## ESEMPIO DI SERVER

```
    ...
    PrintStream outp = new PrintStream(os);
    outp.println(new java.util.Date());
    clientSock.close();
   }
  } catch (UnknownHostException e){
     System.err.println("Host unknown");
  } catch (Exception e){
     System.err.println(e);
  }
 }
}
```

Una volta attivato, il server non termina mai: svolge *per costruzione* un ciclo infinito! Per fermarlo: CTRL+C

## CLIENT & SERVER: USO

**Se ora usiamo il cliente di prima con il nostro server:**

```
public class EsempioNet2 {
 public static void main(String args[]){
  Socket s = null;
  try { s = new Socket("localhost",7777);
  ...
```

Otteniamo:
`Fri Feb 18 16:44:46 GMT+01:00 2000`

## UN ESEMPIO "CLIENT + SERVER"

- **Il server risponde con un numero progressivo, a partire da 1, a ogni cliente che si connette.**
- **Il cliente si limita a visualizzare tutto quello che gli arriva dal server, fino a quando non riceve il messaggio `Stop`: a quel punto, il cliente termina.**

## UN ESEMPIO "CLIENT + SERVER"

**Il cliente**

```
import java.net.*;  import java.io.*;
public class Cliente1 {
 public static void main(String args[]){
  Socket s = null;
  try {
   s = new Socket("localhost",11111);
   BufferedReader r = new BufferedReader(
     new InputStreamReader(s.getInputStream()));
   String line;
   ...
```

## UN ESEMPIO "CLIENT + SERVER"

**Il cliente (segue)**

```
    ...
    while((line=r.readLine())!=null ){
     System.out.println(line);
     if (line.equals("Stop")) break;
    }
    r.close(); s.close();
   } catch (UnknownHostException e){
      System.err.println("Host unknown");
   } catch (Exception e){
      System.err.println(e);
   }
  }
}
```

## UN ESEMPIO "CLIENT + SERVER"

**Il server**

```
import java.net.*;  import java.io.*;
public class Server1 {
 public static void main(String args[]){
  Socket cs = null; ServerSocket ss = null;
  int numero = 1;
  try {
   ss = new ServerSocket(11111);
   while (true) { // ciclo infinito del server
     cs = ss.accept();
    ...
```

## UN ESEMPIO "CLIENT + SERVER"

**Il server (segue)**

Autoflush attivo

```
    PrintWriter pw =
      new PrintWriter(cs.getOutputStream(),true);
    pw.println("Nuovo numero: " + numero);
    numero++; pw.println("Stop");
    pw.close(); cs.close();
    } // end while
   } catch (UnknownHostException e){
      System.err.println("Host unknown"); }
     catch (Exception e){ System.err.println(e); }
  }
}
```

## UN ESEMPIO "CLIENT + SERVER"

**Il risultato invocando più clienti:**

```
D:\esercizi>java Cliente1
Nuovo numero: 1
Stop

D:\esercizi>java Cliente1
Nuovo numero: 2
Stop

D:\esercizi>java Cliente1
Nuovo numero: 3
Stop

...
```

## UN ALTRO ESEMPIO client/server

**Una mini-calcolatrice "client / server"**

- **Il cliente invia al server una serie di valori `double`: lo 0 indica la fine della sequenza** (ipotesi: i valori sono sulla riga di comando). **Poi si mette in attesa del risultato dal server, e lo stampa a video.**
- **Il server riceve dal cliente una serie di valori `double`, e *li somma uno all'altro* fino a che riceve uno 0; a quel punto, invia il risultato al cliente.**

## UN ALTRO ESEMPIO client/server

**Una mini-calcolatrice "client / server"**

- **Il cliente invia al server una serie di valori `double`: lo 0 indica la fine della sequenza** (ipotesi: i valori sono sulla riga di comando).

**Attenzione: meglio non usare `available()`!**

**C'è il rischio di corse critiche: se la rete è lenta, i byte inviati dal client *possono non essere ancora arrivati* quando il server esegue available() → *protocollo insicuro***

**Alternativa: un ciclo di attesa basato su `!available()`** *(soluzione poco elegante)*

## UN ALTRO ESEMPIO client/server

**Il cliente**

```
import java.net.*;  import java.io.*;
public class Cliente2 {
 public static void main(String args[]){
  Socket s = null;
  try {
   s = new Socket("localhost",11111);
   DataInputStream is =
     new DataInputStream(s.getInputStream()));
   DataOutputStream os =
     new DataOutputStream(s.getOutputStream()));
   ...
```

## UN ALTRO ESEMPIO client/server

**Il cliente (segue)**

```
   ...
   for (int i=0; i<args.length; i++) {
    System.out.println("Sending " + args[i]);
    os.writeDouble(Double.parseDouble(args[i]));
   }
   os.writeDouble(0);
   // lettura risultato dal server
   double res = is.readDouble();
   System.out.println("Risultato = " + res);
   is.close(); os.close(); s.close();
  } catch (Exception e){System.err.println(e);}
 }
}
```

## UN ALTRO ESEMPIO client/server

**Il server**

```
import java.net.*;  import java.io.*;
public class Server2 {
 public static void main(String args[]){
  Socket cs = null; ServerSocket ss = null;
  try {
   ss = new ServerSocket(11111);
   while (true) { // ciclo infinito del server
    cs = ss.accept();
    DataOutputStream os =
     new DataOutputStream(cs.getOutputStream());
    ...
```

## UN ALTRO ESEMPIO client/server

```
   ...
   DataInputStream os =
     new DataInputStream(cs.getInputStream());
   double res = 0, val = 0;
   do { val = is.readDouble();
     res += val;
   } while(val>0)
   os.writeDouble(res);
   os.close(); is.close(); cs.close();
   } // end while
  } catch (UnknownHostException e){
   System.err.println("Host unknown");
  } catch (Exception e){ System.err.println(e);}
 }
}
```

soluzione pulita: ciclo **do / while**

se lo stream finisce anzi-tempo, EOFException

## UN ALTRO ESEMPIO client/server

**Il risultato in alcune situazioni:**

```
D:\esercizi>java Cliente2 3 4 5
Sending 3
Sending 4
Sending 5
Risultato = 12.0
D:\esercizi>java Cliente2 34 5
Sending 34
Sending 5
Risultato = 39.0
```

## ULTERIORI ESEMPI

- **Un server Web**