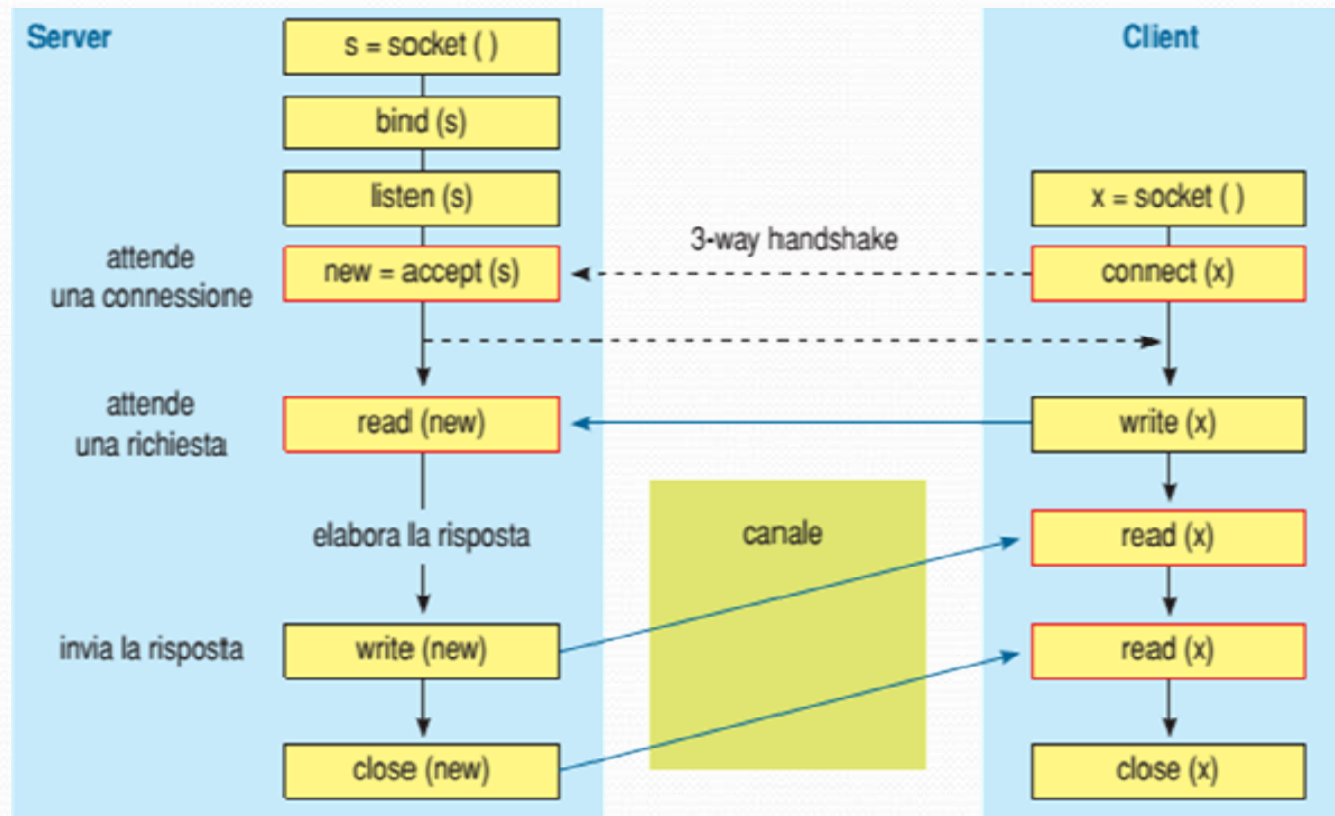




# Socket

Connessione

# Connessione con i Socket



# La sequenza di connessione

## Lato server


1. Creazione del socket
2. **Bind** ad una porta
3. **Listen**, predisposizione a ricevere sulla porta
4. **Accept**, blocca il server in attesa di una connessione
5. Lettura - scrittura dei dati
6. Chiusura

## Lato client

1. Creazione del socket
2. Richiesta di connessione
3. Lettura - scrittura dei dati
4. Chiusura

# Socket in Java - Connessione

- E' organizzata in maniera da soddisfare i principi dell'Object Orienting Programming
- Contiene classi per la manipolazione degli indirizzi e la creazione di socket sia lato **CLIENT** che **SERVER**
- Implementazione nel package `java.net`



Java Networking API (java.net) fornisce le classi e interfacce per le seguenti funzioni

1. Indirizzamento

➤ InetAddress

2. Creazione di connessioni TCP

➤ ServerSocket, Socket

3. Creazione di connessioni UDP

➤ DatagramPacket, DatagramSocket

4. Localizzare risorse di rete

➤ URL, URLConnection, HttpURLConnection

5. Sicurezza (autenticazione, permessi)

# Classe InetAddress

- Rappresenta un indirizzo IP e fornisce i metodi per manipolarlo.
- Non offre costruttori pubblici ma dei metodi statici per creare istanze di questa classe:
  - `InetAddress InetAddress.getByName(String hostname)`
  - `InetAddress[] InetAddress.getAllByName(String hostname)`
  - `InetAddress InetAddress.getLocalHost()`



# Classe InetAddress

- Il metodo `getByName()` può impiegare il DNS per recuperare l'indirizzo IP associato all'hostname fornito come argomento
- Si può specificare nel suo argomento sia un indirizzo IP nella forma decimale che il nome di un host
- In entrambi i casi, l'oggetto `InetAddress` restituito conterrà l'indirizzo IP a 32 bit
- Nel caso che ad un hostname siano associati più indirizzi IP, li si può ottenere chiamando la `getAllByName()`
- Il metodo `getLocalHost()` restituisce l'indirizzo IP dell'host su cui viene eseguita
- `getHostName` e `getHostAddress` restituiscono rispettivamente il nome e l'indirizzo IP che rappresentano

# Classi per i socket

- Java fornisce due diverse classi per la comunicazione con il protocollo TCP che rispecchiano la struttura client/ server:
  - creazione socket per il server : classe **ServerSocket**
  - creazione socket per il client : classe **Socket**
- La differenziazione del socket Client e Server è dovuto alle diverse operazioni che vengono svolte al momento di stabilire una connessione
  - Un server ottiene l'oggetto socket da una chiamata al metodo `accept()`,
  - Il client deve provvedere a creare un'istanza del socket.



# Fasi del SERVER

```
ServerSocket serverSocket = new ServerSocket(port);  
Socket socket = serverSocket.accept();
```

```
socket.getInputStream(); //Stream per la lettura  
socket.getOutputStream(); //Stream per la scrittura
```



# Costruttori del Socket (lato SERVER)

- `ServerSocket()`
  - `ServerSocket(int port)`
  - `ServerSocket(int port, int backlog)`
  - `ServerSocket(int port, int backlog, InetAddress bindAdd)`
- 
- Il parametro **port** specifica la porta su cui rimanere in attesa
  - Il parametro **backlog** il numero massimo di richieste di connessione (default 50 )
  - Il parametro **bindAdd** viene utilizzato dai server multihomed (con più interfacce di rete) per specificare un determinato indirizzo

# Classe ServerSocket

- Non è necessario specificare la famiglia dei protocolli; si opera sempre con IP
- impiego di questa classe implica l'uso del protocollo TCP
- Un eventuale errore genera una eccezione IOException sollevata dai costruttori
- Il costruttore realizza tutte le operazioni di socket(), bind() e listen()

## Classe Socket (lato SERVER)

- Il metodo più importante è `accept()`, che blocca il server in attesa di una connessione. E' questo il metodo che restituisce un oggetto di tipo **socket** completamente istanziato nei parametri ( dati locali e remoti) che viene poi utilizzato per gestire la comunicazione con il client.
- `Socket _miosocket = serverSocket.accept();`

# Lettura/scrittura

- Dopo che il metodo `accept()` ritorna un socket valido è possibile eseguire operazioni di I/O su quel socket

```
BufferedReader sockIN = new BufferedReader(  
    new InputStreamReader(  
        socket.getInputStream());  
PrintWriter sockOUT = new PrintWriter(socket.getOutputStream(), true);
```

```
String line = sockIN.readLine();  
  
sockOUT.println("...")
```

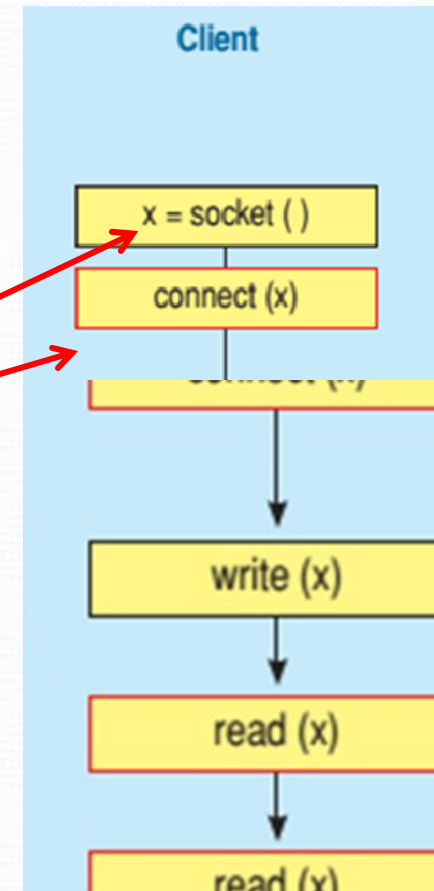


# Classe Socket (lato CLIENT)

- La classe Socket è la stessa utilizzata dal server;
- Nel client, però, per creare un oggetto di questa classe si usa il costruttore

```
Socket socket = new Socket(host, port);
```

- La lettura e la scrittura funziona come nel Server...



# Eccezioni

- I costruttori delle classi `ServerSocket`, `Socket`, il metodo `accept()` nonché i metodi per leggere e scrivere uno stream di dati possono generare delle ECCEZIONI (`IOException`) che vanno opportunamente gestite

# Chiusura dei Socket

- E' importante che i socket siano propriamente chiusi al termine da una applicazione mediante il metodo `close()` delle classi ( sia `Socket` che `ServerSocket`) poiché questi sono una risorsa di rete del sistema
- E' importante chiudere anche gli stream in lettura e scrittura

```
try {
    socket = new Socket(host, port);
    sockIN = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    sockOUT = new PrintWriter(socket.getOutputStream(), true);
```

```
    socket.close();
    sockIN.close();
    sockOUT.close();
```

```
    sockOUT.close();
```

```
    e) {
    }
```

L'operazione di chiusura può generare a sua volta un'eccezione che potrebbe impedire di chiudere tutte le connessioni

La clausola **Finally** risolve il problema... a patto di mettere ogni close() in un blocco try

# Try – with resource

- In Java il problema è stato risolto in maniera molto elegante con il costrutto try – with resource.

```
try (Socket socket = new Socket(host, port);  
    BufferedReader sockIN = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
    PrintWriter sockOUT = new PrintWriter(socket.getOutputStream(), true);) {  
    ...  
}  
  
catch (Exception e) {  
    System.out.println("errore");  
};  
}
```

- Le risorse saranno chiuse sempre e comunque dalla JVM indipendentemente dal flusso di esecuzione