Programowanie klient – serwer

Gniazda

Klasa InetAddress

Adres komputera w sieci opisany przez:

- domenę np. onet.pl
- numer IP, np. 212.51.207.68

Klasa nie ma publicznych konstruktorów.

Wykorzystujemy metodę **`getByName(String host)`**:

```
InetAddress addr =
InetAddress.getByName("wfis.uni.lodz.pl");
InetAddress addr =
InetAddress.getByName("212.60.60.68");
```

lub **`getByAddress(byte[] addr)`**:

```
byte ip[] = {(byte)212,(byte)60,(byte)60,(byte)68};
InetAddress poli = InetAddress.getByAddress(ip);
```

Klasa InetAddress

Niektóre hosty (te, które maja wiele kart sieciowych) pod jedna nazwą mogą mieć wiele adresów IP.

Tablicę obiektów reprezentujących adresy hosta możemy uzyskać za pomocą **`getAllByName(String host)`**.

Do stworzenia obiektu reprezentującego adres komputera lokalnego, na którym uruchomiony zostaje program służy metoda **`getLocalHost()`**.

TCP – gniazda

Klasa **`Socket`**.

Przesył danych strumieniowych.

Dwukierunkowa łączność.

Do nawiązywania połączenia ze zdalnym hostem służy konstruktor gniazda.

Każdy obiekt typu **`Socket`** jest związany z dokładnie jednym zdalnym hostem. Aby połączyć się z innym hostem, trzeba utworzyć nowy obiekt typu **`Socket`**.

TCP – gniazda

Obiekt typu **`Socket`** wymaga wyspecyfikowania zdalnego hosta oraz numeru portu, z którym chcemy utworzyć połączenie. Można to zrobić w postaci:

- nazwy hosta:

```
public Socket(String host, int port)
        throws UnknownHostException, IOException
```

- obiektu **`InetAddress`**:

```
public Socket(InetAddress address, int port)
        throws IOException
```

TCP – gniazda

W przypadku wielu lokalnych adresów możemy wskazać ten, z którego będziemy się łączyć:

```
public Socket(String host, int port, InetAddress
localAddr, int localPort) throws IOException
```

```
public Socket(InetAddress address, int port,
InetAddress localAddr, int localPort) throws
IOException
```

Można też wskazać numer portu, z którego będziemy się łączyć – zwykle numer ten przydzielany jest dynamicznie (poprzednie konstruktory).

W danej chwili na jednym porcie może być realizowane tylko jedno połączenie.

Dynamiczne wybieranie wolnego portu przez system - wartość portu 0.

TCP – gniazda – wyjątki

Konstruktory mogą zwracać wyjątki, trzeba zapewnić ich obsługę.

Ogólnie – klasa **`IOException.`**

Bardziej szczegółowe informacje zawiera klasa wyjątków związanych z gniazdem – **`SocketException`**:

`ConnectException` – połączenie odrzucone, np. żaden proces nie nasłuchuje na danym porcie zdalnego hosta.

`NoRouteToHostException` – host niedostępny, np. jest za firewallem.

`PortUnreachableException` – nieosiągalny port.

TCP – gniazda

Obiekt klasy **`Socket`** – informacja o gnieździe lokalnym i zdalnym.

Obiekt **`InetAddress`** do którego się podłączyliśmy i z którego się łączymy możemy pobrać metodami:

```
public InetAddress getInetAddress()
public InetAddress getLocalAddress()
```

Numery portów połączenia na zdalnym i lokalnym gnieździe:

```
public int getPort()
public int getLocalPort()
```

`getLocalPort()` – można sprawdzić, który port został dynamicznie wybrany przez klienta.

`toString()` – informacje o gnieździe w postaci łańcucha znakowego:

```
Socket[addr=onet.pl/212.10.10.2,port=21,
localport=1116]
```

Pobieranie i wysyłanie danych

Po nawiązaniu połączenia możemy przesyłać i odbierać dane.

Metody odbierające dane są metodami blokującymi (czekają aż pojawią się dane w strumieniu).

Rodzaje danych:

- tekstowe
- binarne
- serializowane (obiekty).

Obiekty **`InputStream`**, **`OutputStream`** reprezentują strumienie wejściowe i wyjściowe połączenia sieciowego.

Do wydobycia tych strumieni służą metody klasy **`Socket`**:

```
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

Czytanie danych tekstowych

Analogia do czytania z plików, z klawiatury, z URL.

Przejście ze strumienia do treści przesyłanej za pomocą klas konwertujących.

```
InputStream is = gniazdo.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
```

`InputStreamReader` – strumień przekształcający bajtowy strumień w strumień znaków.

`BufferedReader` - strumień buforujący dane i umożliwiający czytanie danych porcjami.

W przypadku klawiatury mamy: **`InputStream is = System.in;`**

Czytanie danych tekstowych

Klasa **`BufferedReader`** udostępnia metody wczytujące:

`public int read() throws IOException` – czyta pojedynczy znak ze strumienia. Zwraca go w postaci **`int`**. W przypadku napotkania końca strumienia zwraca -1.

`public int read(char[] cbuf, int off, int len) throws IOException` – wczytuje porcje znaków (**`len`**) do tablicy (**`cbuf`**) z określonego miejsca ze strumienia (off). Zwraca ilość wczytanych znaków lub -1 w przypadku końca strumienia.

`public String readLine() throws IOException` – wczytuje linię tekstu zakończoną '\n' lub '\r'.

Dane tekstowe – zapis

```
OutpuStream os = gniazdo.getOutputstream();
PrintWriter pw = new PrintWriter(os, true);
```

Klasa **`PrintWriter`** – metoda **`print(...)`** przyjmująca argumenty w postaci liczb lub łańcuchów znaków.

`println(...)` nie dodaje znaczników końca wiersza:

```
pw.print("tekst" + "\r\n");
```

Można pisać bezpośrednio do obiektu **`OutputStream`**. Metody:

`public void write(byte[] b) throws IOException` – zapisuje **`b.length`** bajtów z tablicy bajtowej do strumienia wyjściowego.

`public void write(byte[] b, int off, int len) throws IOException` – zapisuje **`len`** bajtów z tablicy bajtowej, zaczynając od pozycji **`off`** do strumienia wyjściowego.

Dane tekstowe – echo

```
import java.net.*;
import java.io.*;
public class Start{
String msg = "wiadomosc";
public static void main(String[] args) {
Start m = new Start();
m.echo("localhost");
}
public void echo(String host){
try{
Socket sock = new Socket(host, 7);
BufferedReader in = new BufferedReader(new
InputStreamReader(sock.getInputStream()));
PrintWriter out = new PrintWriter(sock.getOutputStream());
out.print(msg + "\r\n");
out.flush();
String reply = in.readLine();
System.out.println("wyslano \"" + msg + "\"");
System.out.println("odebrano \"" + reply + "\"");
sock.close();
}
```

Dane binarne

Jeśli chcemy przekazywać dane binarne musimy utworzyć obiekty klas `DataInputStream oraz DataOutputStream.`

Najprostszy sposób:

```
DataInputStream in = new DataInputStream
(gniazdo.getInputStream());
DataOutputStream out = new DataOutputStream
(gniazdo.getOutputStream());
```

Duże dane – strumienie buforowane:

```
DataInputStream in = new DataInputStream(
new BufferedInputStream(gniazdo.getInputStream()));
DataOutputStream out = new DataOutputStream(
new BufferedOutputStream(gniazdo.getOutputStream()));
```

Dane binarne

Klient odbierający liczbę `int` z hosta zdalnego:

```
try{
Socket sock = new Socket("192.168.0.2",10);
DataInputStream in = new DataInputStream(
new BufferedInputStream(sock.getInputStream()));
int liczba = in.readInt();
System.out.println("odebralem liczbe: " + liczba);
sock.close();
}
catch(IOException e){
System.out.println("Blad "+e);
}
```

Dane serializowane

Tworzymy obiekty klas **`ObjectInputStream`** oraz **`ObjectOutputStream`**. Analogicznie jak poprzednio:

```
Socket gniazdo = new Socket("195.117.215.1",1050);
ObjectInputStream in = new ObjectInputStream(
new BufferedInputStream(gniazdo.getInputStream()));
ObjectOutputStream out = new ObjectOutputStream(
new BufferedOutputStream(gniazdo.getOutputStream()));
```

Metody:

```
public final Object readObject() throws IOException,
ClassNotFoundException
public final void writeObject(Object obj) throws
IOException
```

Metody do obsługi danych binarnych:

```
readInt(), writeFloat(float v) ...
```

Dane serializowane

Klient TCP odbierający obiekt klasy **`Date`**:

```
public class Klient {
public static void main(String[] args) {
try{
Socket sock = new Socket("192.168.0.2",10);
ObjectInputStream in = new ObjectInputStream(
new BufferedInputStream(sock.getInputStream()));
Object ob = in.readObject();
Date d = (Date)ob;
System.out.println("data: " + d);
sock.close();
}
catch(ClassNotFoundException e) { System.out.println("Blad "+e); }
catch(IOException e) { System.out.println("Blad "+e); }
}
}
```

Serwer TCP

- Gniazdo serwerowe ma za zadanie nasłuchiwania na podanym porcie zgłoszeń klientów i obsługiwanie połączenia z nimi.
- Klasa **`ServerSocket`**
- Możliwość obsługi wielu klientów.
- Połączenie dwukierunkowe.
- Transmisja strumieniowa.
- Tylko jeden proces może słuchać na danym porcie w danym czasie.

Serwery

Rodzaje serwerów:

- serwer jednowątkowy
- serwer pozornie wielowątkowy
- serwer wielowątkowy

Serwery

Serwer jednowątkowy:

- prosta budowa
- możliwość obsługi tylko jednego klienta naraz
- kolejny klient musi czekać na swoją kolej

Serwer pozornie wielowątkowy:

- zaawansowana budowa
- możliwość obsługi kilku klientów naraz
- kolejny klient musi czekać aż wykonana zostanie tura obsługi podłączonych klientów

Serwer wielowątkowy:

- możliwość obsługi kilku klientów naraz
- nowi klienci obsługiwani są natychmiast

Serwer TCP

Schemat działania serwera:

1. open – tworzy gniazdo serwera
2. bind – łączy stworzone gniazdo z lokalnym portem
3. accept – nasłuchuje i akceptuje połączenia od klientów
4. read – pobiera dane ze strumienia połączenia z klientem
5. write – wysyła dane do strumienia połączenia z klientem
6. close – zamyka połączenie
7. close2 – zamyka serwer

Tworzenie serwera TCP

Konstruktory:

```
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws
IOException
public ServerSocket(int port, int backlog, InetAddress
bindAddr)
throws IOException
```

Najczęściej wykorzystuje się pierwszy konstruktor, podając numer portu, na którym serwer ma nasłuchiwać:

```
try {
ServerSocket s = new ServerSocket(80);
}
catch (IOException e) {
System.err.println(e);
}
```

Tworzenie serwera TCP

- Podczas tworzenia obiektu **`ServerSocket`** obiekt ten próbuje dowiązać się do wskazanego portu na lokalnym hoście (ang. binding).
- Jeśli port jest już zajęty wyrzucony zostanie wyjątek: **`java.net.BindException`** z klasy bazowej **`java.io.IOException`**
- Tworzenie własnych serwerów na portach >1024.
- Port nr 0 – automatyczny wybór pierwszego wolnego portu. Wybrany automatycznie numer portu można odczytać za pomocą metody **`getLocalPort()`**.

Serwer TCP – kolejka połączeń

- System operacyjny zapamiętuje połączenia przychodzące dla każdego portu w kolejce typu FIFO aż do jej zapełnienia.
- Po zapełnieniu kolejki następne połączenia przychodzące będą odrzucane, chyba, że w kolejce zwolni się miejsce.
- Długość kolejki zależy od systemu operacyjnego (5-50).
- Rozmiar kolejki połączeń może zostać zmieniony w zależności od potrzeb. Konstruktor:
  ```
  public ServerSocket(int port, int backlog) throws IOException
  ```
- Każdy system operacyjny ma określoną maksymalną długość kolejki. Jeśli zadeklarowana kolejka będzie dłuższa niż maksimum, to długość kolejki zostanie skrócona do wartości maksymalnej.
- Po utworzeniu gniazda serwera długość kolejki nie może być zmieniana.

Tworzenie serwera TCP

- Komputer może używać wielu interfejsów sieciowych.
- Poprzednio stworzone gniazda nasłuchują na wszystkich interfejsach.
- Do stworzenia gniazda serwera tylko dla wybranego interfejsu służy konstruktor:

```
public ServerSocket(int port, int backlog, InetAddress
bindAddr) throws IOException
```

Przykład:

```
InetAddress ia = InetAddress.getByName("192.168.0.2");
ServerSocket ss = new ServerSocket(80, 50, ia);
```

Serwer TCP – nasłuchiwanie

- Po stworzeniu gniazda serwer czeka na zgłoszenia klientów.
  Metoda: **`public Socket accept() throws IOException`**
- Metoda nasłuchuje i akceptuje połączenia z klientem czekającym w kolejce.
- Zwraca obiekt klasy **`Socket`**, reprezentujący połączenie klient-serwer.
- Metoda blokująca. Blokuje wykonywanie dalszych instrukcji dopóki nie nastąpi połączenie.
- Można zmienić opcje metody **`accept()`**:
  `public void setSoTimeout(int timeout) throws SocketException` gdzie **`timeout`** – czas oczekiwania na połączenie od klienta (domyślne 0 – nieskończoność).
  Wówczas metoda **`accept`** zwróci wyjątek **`SocketTimeoutException`** gdy czas nasłuchiwania minie.

Serwer TCP – połączenie

- Połączenie z klientem realizowane jest za pomocą obiektu gniazda **`Socket`**
- Odczytywanie informacji o połączeniu jak w przypadku Klienta TCP
- Wysyłanie i odbieranie danych takie samo jak w Kliencie TCP

Serwer TCP – zakończenie działania

- Metoda **`close()`** klasy **`Socket`** kończy konkretne połączenie klient-serwer.
- Serwer nadal działa.
- Aby zakończyć działanie serwera należy wykonać metodę **`close()`** na obiekcie klasy **`ServerSocket`**