Marcin Lis

Java

Wydanie II

ĆWICZENIA ZAAWANSOWANE

Mistrzostwo w Javie w zasięgu Twoich rąk!

- *Na czym polegają programowanie współbieżne i obsługa pracy wątków?*
- *Jak zapewnić właściwą komunikację sieciową z użyciem protokołu wymiany danych?*
- *Co zrobić, by współpraca z relacyjnymi bazami danych czy obsługa zapytań SQL przebiegały idealnie?*

Helion

Redaktor prowadzący: Ewelina Burska

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 032 231 22 19, 032 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?czjav2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wszystkich opublikowanych listingów można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/czjav2.zip

ISBN: 978-83-246-3497-2

Printed in Poland.

Spis treści

3

Programowanie sieciowe

Mechanizm gniazd

Mechanizm gniazd jest znany większości programistów, jednak dla tych czytelników, którzy się z nim nie zetknęli, krótkie wyjaśnienie.

Gniazda (ang. *sockets*) jest to mechanizm komunikacyjny, umożliwiający transmisję danych pomiędzy urządzeniami w sieci opartej na protokole IP. Obecnie jest to mechanizm powszechnie stosowany w komunikacji sieciowej. Gniazda można traktować jako końcówki połączeń znajdujące się w komputerach (ogólniej: urządzeniach sieciowych). Stąd też pochodzi nazwa „gniazdo" — czyli coś, do czego można włożyć wtyczkę. Oczywiście w tym przypadku chodzi o wtyczkę wirtualną.

Po utworzeniu gniazda można używać go do komunikacji z innym komputerem bądź też urządzeniem sieciowym. Dane wysłane do gniazda będą przesyłane do urządzenia, z którym zostało nawiązane połączenie. Transmisja jest oczywiście dwukierunkowa, zatem dane odsyłane przez odległe urządzenie sieciowe można również odbierać z gniazda. Jest to zatem swego rodzaju punkt komunikacyjny.

Aby połączyć się z odległym komputerem, niezbędne jest określenie dwóch wartości. Po pierwsze — jego adresu sieciowego IP, po drugie — numeru portu. Czym jest numer portu? Otóż na każdym komputerze może działać wiele usług, np. serwer WWW, poczty czy FTP. Trzeba zatem zakomunikować, z jaką usługą (aplikacją, procesem) ma nastąpić połączenie. Numer portu jest właśnie takim identyfikatorem. Liczba dostępnych portów jest zależna od rodzaju i wersji systemu operacyjnego. Jednak niezależnie od tego, ile ich oferuje system, dla typowych protokołów transmisyjnych TCP i UDP można użyć co najwyżej 65 535 portów[1], numerowanych od 1 do 65 535 (port zerowy nie jest używany do transmisji danych). Zatem teoretycznie

[1] Wynika to z tego, że w nagłówkach segmentów danych TCP i UDP na numer portu zarezerwowane jest 16 bitów.

dla jednego adresu IP tyle właśnie różnych usług i serwisów można zaoferować (w praktyce liczba ta będzie mniejsza, gdyż część portów jest zarezerwowana, a pojedyncza usługa może też korzystać z wielu z nich).

Gniazda z reguły dzielimy na **strumieniowe** — umożliwiające transmisję strumieniową (np. TCP), **datagramowe** — umożliwiające transmisję pakietową (np. UDP) oraz tzw. **raw sockets** — pozwalające na bezpośrednie wysyłanie pakietów (ramek) IP z pominięciem narzutu protokołów wyższych warstw sieciowych (spotykane tłumaczenie nazwy to „gniazda surowe"). W dalszej części rozdziału będzie poruszany jedynie temat gniazd strumieniowych.

Gniazda w Javie

W Javie dostępne są gniazda służące do komunikacji sieciowej. Ten, kto programował „czyste" gniazda, np. pod Uniksem, będzie jednak z pewnością mile zaskoczony, gdyż mechanizmy te w przypadku Javy są o wiele bardziej przyjazne użytkownikowi. Odpowiednie klasy znajdują się w pakiecie `java.net`. Do dyspozycji są **gniazda klienckie** (ang. *client socket*) i **serwerowe** (ang. *server socket*), zarówno strumieniowe, jak i datagramowe. Do komunikacji wykorzystywany jest protokół IP.

Gniazda klienckie strumieniowe reprezentowane są przez klasę `Socket`, która udostępnia konstruktory przedstawione w tabeli 3.1.

Tabela 3.1. *Konstruktory klasy Socket*

Konstruktor	Opis
`Socket()`	Tworzy gniazdo niepołączone z żadnym adresem.
`Socket(InetAddress address, int port)`	Tworzy nowe gniazdo podłączone do adresu `address` i portu `port`.
`Socket(InetAddress address, int port, InetAddress localAddr, int localPort)`	Tworzy nowe gniazdo podłączone do adresu `address` i portu `port` oraz do lokalnego adresu `localAddr` i lokalnego portu `localPort`.
`Socket(Proxy proxy)`	Tworzy nowe gniazdo, używające do komunikacji serwera pośredniczącego proxy wskazanego przez argument `proxy`. Konstruktor dostępny od wersji JDK 1.5.
`Socket(SocketImpl impl)`	Tworzy nowe gniazdo niepodłączone do żadnego adresu, o implementacji zadanej przez użytkownika.
`Socket(String host, int port)`	Tworzy nowe gniazdo podłączone do adresu wskazanego przez ciąg znaków `host` oraz portu wskazanego przez argument `port`.
`Socket(String host, int port, InetAddress localAddr, int localPort)`	Tworzy nowe gniazdo podłączone do komputera `host` i portu `port` oraz do lokalnego adresu `localAddr` i lokalnego portu `localPort`.

Oprócz konstruktorów wymienionych w tabeli 3.1 dostępne są jeszcze dwa inne: `Socket(InetAddress host, int port, boolean stream)` i `Socket(String host, int port, boolean stream)`, są one jednak przestarzałe i nie należy ich stosować (zostały zachowane jedynie w celu zachowania zgodności z wcześniejszymi JDK). Zawierają bowiem argument `stream` wskazujący, czy gniazdo ma być strumieniowe, czy datagramowe, a obecnie dla gniazd datagramowych należy stosować klasę `DatagramSocket`.

Przy tworzeniu obiektów typu `Socket`, w zależności od użytego konstruktora, mogą zostać zgłoszone następujące wyjątki:

- `IOException` — gdy wystąpił błąd wejścia-wyjścia,
- `UnknownHostException` — gdy nie można uzyskać adresu IP wskazanego hosta,
- `SecurityException` — gdy brak wystarczających uprawnień do utworzenia gniazda,
- `IllegalArgumentException` — gdy argument `port` zawiera wartość spoza dopuszczalnego zakresu (`0` – `65535`),
- `NullPointerException` — gdy argument wskazujący adres jest pusty (ma wartość `null`).

Spróbujmy zatem utworzyć obiekt typu `Socket` połączony z wybranym adresem zdalnym.

ĆWICZENIE

3.1 Tworzenie gniazda

Napisz program tworzący gniazdo strumieniowe połączone z wybranym adresem i portem zdalnym. Wyświetl informacje o połączeniu na ekranie.

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

public class Main
{
  public static void main(String args[])
  {
    Socket socket = null;
    try{
      socket = new Socket("helion.pl", 80);
    }
    catch(UnknownHostException e){
      System.out.println(e);
    }
    catch(IOException e){
      System.out.println(e);
    }
    if(socket != null){
      System.out.println(socket);
    }
  }
}
```

Na początku kodu importowane są pakiety `java.net` (do obsługi gniazd) oraz `java.io` (ze względu na obsługę wyjątku `IOException`). W klasie `Main` znajduje się zmienna `socket` typu `Socket`, której początkową wartością jest `null`. W bloku `try` następuje próba utworzenia nowego obiektu typu `Socket` i przypisania odniesienia do niego tej właśnie zmiennej. W konstruktorze przekazywany jest adres hosta, z którym ma nastąpić połączenie (`helion.pl`) oraz numer portu (`80` — standardowy port protokołu HTTP). Ponieważ konstruktor może zgłosić różne wyjątki, zostały również użyte dwa bloki `catch`, aczkolwiek jedyną czynnością w nich wykonywaną jest wyświetlanie danych obiektu wyjątku. W praktyce można oczywiście zróżnicować sposób reakcji na błąd w zależności od jego typu. Na zakończenie wyświetlane są dane obiektu `socket`, o ile taki obiekt udało się utworzyć, czyli gdy zmienna `socket` jest różna od `null`.

Po skompilowaniu i uruchomieniu programu na ekranie powinien pojawić się widok zaprezentowany na rysunku 3.1. Dostarczone zostaną informacje o adresie domenowym, adresie IP, numerze portu zdalnego oraz numerze portu lokalnego. Warto też zmienić w kodzie adres lub port, z którym ma nastąpić połączenie, na nieprawidłowy, po czym ponownie skompilować i uruchomić aplikację. Zostaną wtedy wyświetlone informacje o obiekcie wyjątku, np. takie jak na rysunku 3.2.

```
E:\WINDOWS\system32\cmd.exe

E:\java>javac Main.java

E:\java>java Main
Socket[addr=helion.pl/213.186.88.113,port=80,localport=2923]

E:\java>_
```

Rysunek 3.1. *Informacje o nawiązanym połączeniu*

```
E:\WINDOWS\system32\cmd.exe

E:\java>javac Main.java

E:\java>java Main
java.net.UnknownHostException: helion.pl1

E:\java>javac Main.java

E:\java>java Main
java.net.ConnectException: Connection timed out: connect

E:\java>_
```

Rysunek 3.2. *Komunikaty o błędach związanych z niewłaściwymi danymi*

W tabeli 3.1 widać, że niektóre z konstruktorów klasy `Socket` przyjmują adresy hostów w postaci obiektów klasy `InetAddress`. Obiektów tego typu nie tworzy się bezpośrednio, ale korzystając z metod statycznych tej klasy. Oprócz nich do dyspozycji jest także kilka innych metod pozwalających uzyskiwać informacje dotyczące adresów internetowych. Wybrane metody udostępniane przez `InetAddress` zostały przedstawione w tabeli 3.2.

Tabela 3.2. *Wybrane metody klasy InetAddress*

Typ zwracany	Metoda	Opis
`byte[]`	`getAddress()`	Zwraca adres IP w postaci tablicy bajtów.
`static InetAddress[]`	`getAllByName(String host)`	Zwraca wszystkie adresy IP urządzenia określonego przez argument `host`.
`static InetAddress`	`getByAddress(byte[] addr)`	Zwraca obiekt typu `InetAddress`, odpowiadający adresowi IP przekazanemu w postaci tablicy bajtów `addr`.
`static InetAddress`	`getByAddress(String host, byte[] addr)`	Zwraca obiekt typu `InetAddress`, odpowiadający adresowi IP przekazanemu w postaci tablicy bajtów `addr` i nazwie określonej przez argument `host`.
`static InetAddress`	`getByName(String host)`	Ustala adres IP urządzenia określonego przez argument `host`.
`String`	`getCanonicalHostName()`	Zwraca kwalifikowaną nazwę domenową dla danego adresu IP.
`String`	`getHostAddress()`	Zwraca adres IP w postaci ciągu znaków.
`String`	`getHostName()`	Zwraca nazwę hosta dla danego adresu IP.
`static InetAddress`	`getLocalHost()`	Ustala adres IP komputera lokalnego.
`static InetAddress`	`getLoopbackAddress()`	Ustala adres IP pętli lokalnej (ang. *loopback address*). Metoda dostępna od JDK 1.7.
`boolean`	`isMulticastAddress()`	Sprawdza, czy dany adres jest adresem typu multicast.
`boolean`	`isReachable(int timeout)`	Sprawdza, czy dany host jest osiągalny w sieci. Argument `timeout` określa (w milisekundach) maksymalny czas badania.
`boolean`	`isSiteLocalAddress()`	Sprawdza, czy adres jest adresem lokalnym.
`String`	`toString()`	Dokonuje konwersji adresu na ciąg znaków.

Wykorzystując dane przedstawione w tabeli 3.2, można w prosty sposób napisać program wyświetlający adres IP komputera, na którym został uruchomiony. Wystarczy użyć metody `getLocalHost`.

ĆWICZENIE

3.2 Uzyskiwanie lokalnego adresu IP

```
import java.net.*;

public class Main
{
  public static void main(String args[])
  {
    InetAddress inetAddress = null;
    try{
      inetAddress = InetAddress.getLocalHost();
    }
    catch(UnknownHostException e){
      System.out.println(
         "Nie można uzyskać adresu IP dla tego komputera.");
      System.exit(0);
    }
    String ip = inetAddress.getHostAddress();
    System.out.println("Adres IP tego komputera to: " + ip);
  }
}
```

Najpierw została utworzona zmienna `inetAddress` typu `InetAddress`, a następnie w bloku `try` nastąpiło wywołanie statycznej metody `getLocalHost` klasy `InetAddress`. Metoda ta zwraca obiekt zawierający adres IP komputera lokalnego (na którym został uruchomiony program). Blok `try` jest konieczny, jako że przy wywołaniu `getLocalHost` może wystąpić wyjątek `UnknownHostException`. Będzie tak w sytuacji, gdy pobranie adresu nie jest możliwe. Ewentualny wyjątek jest obsługiwany w bloku `catch` (wyświetlany jest stosowny komunikat i program kończy działanie). Uzyskany adres IP, zawarty w obiekcie `inetAddress`, jest uzyskiwany za pomocą metody `getHostAddress` (która zwróci go w postaci ciągu znaków — obiektu typu `String`) oraz wyświetlany na ekranie.

Skoro możliwe jest pobranie adresu IP komputera lokalnego, na pewno można też pobrać adres dowolnego urządzenia w sieci. W tym celu wystarczy użyć metody `getByName` klasy `InetAddress` i przekazać jej nazwę domenową. Zwrócony obiekt będzie zawierał poszukiwane dane, o ile oczywiście wywołanie metody zakończy się sukcesem. Jeżeli adresu nie uda się pobrać, wygenerowany zostanie wyjątek `UnknownHostException`. Warto zatem napisać program, któremu w wierszu poleceń będzie przekazywana nazwa domenowa, a w odpowiedzi na ekranie pojawi się odpowiadający jej adres IP (o ile taki istnieje).

ĆWICZENIE

3.3 Pobieranie dowolnego adresu IP

Napisz program, który będzie podawał adres IP komputera (urządzenia sieciowego) o nazwie domenowej przekazanej z wiersza poleceń.

```
import java.net.*;

public class Main
{
  public static void main(String args[])
  {
    if (args.length < 1){
      System.out.println("Wywołanie programu: Main nazwa_hosta");
      System.exit(0);
    }
    String host = args[0];
    InetAddress inetAddress = null;
    try{
      inetAddress = InetAddress.getByName(host);
    }
    catch(UnknownHostException e){
      System.out.println(
        "Nie można uzyskać adresu IP dla hosta " + host);
      System.exit(0);
    }
    String ip = inetAddress.getHostAddress();
    System.out.println("Adres IP komputera " + host +" to: " + ip);
  }
}
```

Pierwszą wykonywaną czynnością jest sprawdzenie liczby elementów tablicy `args` przekazanej metodzie `main`. Jeżeli wartość właściwości `length` jest mniejsza od 1, oznacza to, że w wywołaniu programu nie został podany żaden argument. W takiej sytuacji jedyną wykonywaną czynnością jest wyświetlenie komunikatu z informacją o prawidłowym sposobie wywołania i program kończy działanie (wywołanie statycznej metody `exit` z klasy `System`).

Jeżeli jednak aplikacja została uruchomiona z co najmniej jednym argumentem (czyli liczba elementów tablicy `lenght` jest większa od 0), pierwszy argument (o indeksie `0`) jest przypisywany pomocniczej zmiennej `host`, powstaje także zmienna `inetAddress` typu `InetAddress`. Wartość zapisana w `host` jest używana w wywołaniu statycznej metody `getByName` klasy `InetAddress`, a rezultat działania `getByName` (obiekt zawierający dane dotyczące adresu internetowego, w tym poszukiwany adres IP) jest przypisywany zmiennej `inetAddress`.

Wywołanie metody `getByName` jest ujęte w blok `try…catch`, jako że w przypadku niemożności ustalenia adresu jest generowany wyjątek `UnknownHostException`. Jeśli tak się stanie, na ekranie pojawi się odpowiedni komunikat. Jeżeli jednak adres da się uzyskać, zostanie on pobrany za pomocą metody `getHostAddress` i również wyświetlony na ekranie.

Ćwiczenie 3.3 pokazało, jak uzyskać adres IP dowolnego hosta w sieci. Jednak do jednego adresu domenowego może być przypisanych wiele adresów IP. Wszystkie mogą być odczytane za pomocą metody `getAllByName`. Rezultatem jej działania jest tablica obiektów typu `InetAddress`.

ĆWICZENIE

3.4 Pobranie wszystkich adresów przypisanych do wybranego hosta

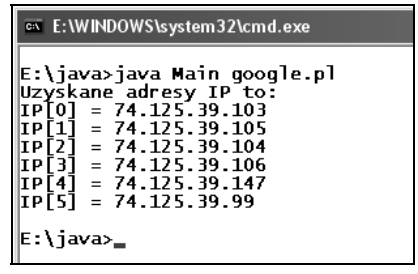
Napisz program, który wyświetli wszystkie adresy IP przypisane do urządzenia sieciowego o nazwie przekazanej w postaci argumentu w wierszu poleceń.

```
import java.net.*;

public class Main
{
  public static void main(String args[])
  {
    if (args.length < 1){
      System.out.println("Wywołanie programu: Main nazwa_hosta");
      System.exit(0);
    }
    InetAddress ips[] = null;
    String hostName = args[0];
    try{
      ips = InetAddress.getAllByName(hostName);
    }
    catch(UnknownHostException e){
      System.out.println(
        "Nie można uzyskać adresów IP dla komputera: " + hostName);
      System.exit(0);
    }
    System.out.println("Uzyskane adresy IP to:");
    for (int i = 0; i < ips.length; i++){
      String ip = ips[i].getHostAddress();
      System.out.println("IP[" + i + "] = " + ip);
    }
  }
}
```

Początek kodu jest taki sam jak w ćwiczeniu 3.3. Potem następuje badanie, czy z wiersza poleceń został przekazany parametr określający nazwę hosta. Dalej deklarowana jest zmienna tablicowa `ips` o początkowej wartości `null`. W bloku `try` zmiennej tej przypisywany jest wynik działania statycznej metody `getAllByName` klasy `InetAddress`. Jeżeli ta operacja zakończy się sukcesem, w tablicy `ips` znajdą się wszystkie adresy hosta określonego przez zmienną pomocniczą `hostName` (zmienna ta zawiera ciąg znaków przekazany jako argument z wiersza poleceń). Jeśli natomiast adresów nie uda się pobrać, jest generowany wyjątek przechwytywany następnie przez blok `catch`.

Zawartość tablicy `ips` jest odczytywana w pętli typu `for`. Adres IP uzyskuje się przez wywołanie metody `getHostAddress` — rezultat jej działania jest przypisywany zmiennej pomocniczej `ip`. Wartość zapisana w `ip` jest następnie wyświetlana na ekranie. Wynik przykładowego wywołania aplikacji został przedstawiony na rysunku 3.3.



Rysunek 3.3. *Odczytanie adresów IP przypisanych nazwie domenowej google.pl*

Gniazda serwerowe

Do tej pory zostały przedstawione jedynie gniazda klienckie (ang. *client sockets*). Pozwalają one jedynie na pisanie programów łączących się z działającymi serwerami. Jeżeli jednak chcemy samodzielnie napisać program serwera, musimy skorzystać z gniazd serwerowych (ang. *server sockets*). Gniazda takiego typu nasłuchują na wskazanym porcie i kiedy nadejdzie połączenie, tworzą dla niego gniazdo klienckie, służące do dalszej komunikacji.

Gniazda serwerowe w Javie są zaimplementowane przez klasę `ServerSocket`. Oferuje ona konstruktory przedstawione w tabeli 3.3.

Tabela 3.3. *Konstruktory klasy ServerSocket*

Konstruktor	Opis
`ServerSocket()`	Tworzy niepowiązane (nieprzypisane) gniazdo serwerowe.
`ServerSocket(int port)`	Tworzy gniazdo serwerowe nasłuchujące na porcie `port`.
`ServerSocket(int port, int backlog)`	Tworzy gniazdo serwerowe nasłuchujące na porcie `port`, z kolejką wejściową o długości wskazanej przez argument `backlog`.
`ServerSocket(int port, int backlog, InetAddress bindAddr)`	Tworzy gniazdo serwerowe nasłuchujące na porcie `port`, z kolejką wejściową o długości `backlog`, przypisane do adresu (powiązane z adresem) `bindAddr`.

Argument `port` może określać konkretny numer portu (od `1` do `65535`) lub też przyjąć wartość `0`. W tym drugim przypadku system sam przydzieli wolny numer. Ta opcja jest użyteczna, gdyż dzięki niej nie trzeba ręcznie sprawdzać, który port jest akurat wolny, jednak uniemożliwia przypisanie serwerowi (gniazdu serwera) wybranego numeru portu.

Argument `backlog` pozwala na ustalenie wielkości kolejki wejściowej. Jego dokładne znaczenie jest uzależnione od konkretnej implementacji Javy i systemu operacyjnego. Jeżeli podczas obsługi jednego zgłoszenia na dany port przychodzi kolejne wywołanie, zostaje ono ustawione w kolejce wejściowej. Jeśli wielkość tej kolejki przekroczy wartość podaną jako `backlog`, wywołanie to zostanie odrzucone. Podanie wartości `0` (lub mniejszej) oznacza, że zostanie użyta wartość domyślna dla danej implementacji systemu.

Argument `bindAddr` przypisuje dane gniazdo do konkretnego adresu IP. Jest to użyteczne w sytuacji, gdy komputer (urządzenie) posiada więcej niż jeden adres IP. W takiej sytuacji podanie parametru `bindAddr` pozwala na akceptowanie wyłącznie połączeń przychodzących na wybrany adres. Jeżeli argument ten będzie miał wartość `null`, gniazdo będzie akceptowało połączenia przychodzące na wszystkie dostępne adresy.

Przy tworzeniu obiektów typu `ServerSocket` może zostać zgłoszony jeden z następujących wyjątków:

- `IOException` — jeżeli wystąpi błąd wejścia-wyjścia,
- `IllegalArgumentException` — jeżeli argument określający port będzie miał wartość spoza dopuszczalnego zakresu (`0` – `65535`),
- `SecurityException` — jeżeli brak jest uprawnień do utworzenia gniazda.

Najważniejsze metody udostępniane przez klasę `ServerSocket` zostały zebrane w tabeli 3.4. Najbardziej przydatna w tej chwili będzie metoda `accept`, która powoduje przejście gniazda w stan nasłuchiwania, czyli oczekiwania na połączenie. Jeżeli takie połączenie nadejdzie, zwraca ona nowy obiekt klasy `Socket`, który może posłużyć do realizacji właściwej komunikacji serwera z klientem.

Tabela 3.4. *Wybrane metody klasy ServerSocket*

Typ rezultatu	Metoda	Opis
`Socket`	`accept()`	Oczekuje na połączenia i akceptuje je, tworząc nowe obiekty klasy `Socket`.
`void`	`bind(SocketAddress endpoint)`	Wiąże gniazdo z adresem i portem określonymi przez argument `endpoint`.
`void`	`bind(SocketAddress endpoint, int backlog)`	Wiąże gniazdo z adresem i portem określonymi przez argument `endpoint`. Argument `backlog` określa rozmiar kolejki wejściowej.
`void`	`close()`	Zamyka gniazdo.

Tabela 3.4. Wybrane metody klasy ServerSocket — ciąg dalszy

Typ rezultatu	Metoda	Opis
InetAddress	getInetAddress()	Zwraca lokalny adres IP, do którego przypisane jest gniazdo.
int	getLocalPort()	Zwraca lokalny port, na którym nasłuchuje gniazdo.
SocketAddress	getLocalSocketAddress()	Zwraca informacje o adresie, do którego jest podłączone gniazdo, lub wartość null, jeżeli gniazdo nie zostało powiązane.
int	getSoTimeout()	Zwraca parametr SO_TIMEOUT dla gniazda.
boolean	isClosed()	Zwraca true, jeżeli gniazdo zostało zamknięte.
int	setSoTimeout(int timeout)	Ustawia parametr SO_TIMEOUT dla gniazda.
String	toString()	Zwraca tekstowy opis gniazda.

Warto zwrócić uwagę na metodę setSoTimeout, ustawiającą parametr SO_TIMEOUT gniazda. Parametr ten określa, jak długo metoda accept ma czekać na przychodzące połączenie. Domyślnie jest to wartość nieskończona, czyli oczekiwanie nie zostanie przerwane. Możemy ten stan jednak zmienić, korzystając z wymienionej metody i podając czas oczekiwania w milisekundach. Wtedy, jeżeli po wywołaniu metody accept w podanym czasie nie nadejdzie żadne połączenie, zostanie wygenerowany wyjątek SocketTimeoutException.

Jak zatem utworzyć najprostszy serwer, którego jedynym zadaniem byłoby wyświetlanie parametrów połączenia z klientem? Zostało to zobrazowane w ćwiczeniu 3.5.

ĆWICZENIE

3.5 Tworzenie gniazda serwerowego

Napisz program serwera, który będzie oczekiwał na wybranym porcie na połączenie. Po nawiązaniu połączenia należy wyświetlić jego parametry i zakończyć działanie aplikacji.

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

public class Server
{
  public static void main(String args[])
  {
    ServerSocket serverSocket = null;
    Socket socket = null;
    try{
      serverSocket = new ServerSocket(6666);
```

```
      }
      catch(IOException e){
        System.out.println(
          "Błąd przy tworzeniu gniazda serwerowego.");
        System.exit(-1);
      }
      try{
        socket = serverSocket.accept();
      }
      catch(IOException e){
        System.out.println(e);
      }
      System.out.println(socket);
      try{
        serverSocket.close();
      }
      catch(IOException e){
        System.out.println(
          "Błąd przy zamykaniu gniazda serwerowego");
      }
    }
  }
```

Na początku funkcji `main` zostały umieszczone dwie zmienne `serverSocket` (dla gniazda serwerowego) oraz `socket` (dla gniazda klienckiego). Obiekt typu `ServerSocket` tworzony jest w bloku `try` za pomocą jednoargumentowego konstruktora, któremu w postaci parametru przekazywana jest wartość `6666`. To oznacza, że gniazdo, o ile uda się je utworzyć, będzie nasłuchiwało (oczekiwało na połączenia) na porcie o takim właśnie numerze. Blok `try` jest potrzebny, bowiem przy wywoływaniu konstruktora może wystąpić wyjątek. W takiej sytuacji jest on przechwytywany w bloku `catch`, na ekranie pojawia się związany z nim komunikat i serwer kończy działanie (dzięki wywołaniu statycznej metody `exit` klasy `System`).

Po utworzeniu gniazda wywoływana jest jego metoda `accept`, a rezultat jej działania przypisuje się zmiennej `socket` reprezentującej gniazdo klienckie:

```
socket = serverSocket.accept();
```

Od tego momentu serwer będzie oczekiwał na połączenia na porcie 6666. Gdy nadejdzie takie połączenie, metoda `accept` zakończy działanie i zwróci obiekt klasy `Socket`, który będzie mógł być użyty do transmisji danych z klientem. Powyższa instrukcja jest ujęta w blok `try…catch`, gdyż podczas oczekiwania może wystąpić wyjątek.

Po uzyskaniu gniazda klienckiego jego stan jest wyświetlany przez przekazanie obiektu `socket` metodzie `println` (`System.out.println(socket)`). To spowoduje wywołanie metody `toString` z klasy `Socket` i wyświetlenie uzyskanego ciągu znaków na ekranie. Na zakończenie gniazdo jest zamykane za pomocą metody `close`.

Do sprawdzenia poprawności działania aplikacji z gniazdem serwerowym potrzebny będzie program klienta. Będzie on wykonywał połączenie z adresem i portem określonymi w wierszu wywołania oraz, po nawiązaniu połączenia, wyświetlał dane dotyczące gniazda klienckiego. Działający w ten sposób kod został przedstawiony w ćwiczeniu 3.6.

ĆWICZENIE

3.6 Klient łączący się z serwerem

Napisz program klienta łączący się z adresem i portem podanymi jako argumenty wywołania. Program powinien wyświetlić parametry połączenia.

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

public class Client
{
  public static void main(String args[])
  {
    if (args.length < 2){
      System.out.println("Wywołanie programu: Client host port");
      System.exit(-1);
    }
    String host = args[0];
    int port = 0;
    try{
      port = new Integer(args[1]).intValue();
    }
    catch(NumberFormatException e){
      System.out.println("Nieprawidłowy argument: port");
      System.exit(-1);
    }
    Socket socket = null;
    try{
      socket = new Socket(host, port);
    }
    catch(UnknownHostException e){
      System.out.println("Nieznany host.");
    }
    catch(IOException e){
      System.out.println(e);
      System.exit(-1);
    }
    System.out.println(socket);
  }
}
```

Na początku badane jest, czy przy wywoływaniu programu zostały przekazane co najmniej dwa argumenty, czyli czy liczba elementów tablicy nie jest mniejsza od 2. Jeśli jest mniejsza, wyświetlany jest komunikat o prawidłowym sposobie wywołania i aplikacja kończy działanie. W przeciwnym przypadku wartość pierwszego argumentu wywołania (wartość komórki tablicy `args` o indeksie `0`) jest przypisywana

zmiennej pomocniczej `host`. Powstaje też zmienna `port` o początkowej wartości `0`. Potem następuje próba przetworzenia ciągu znaków z drugiego argumentu (wartość komórki tablicy `args` o indeksie `1`) na wartość typu `int` i przypisanie jej zmiennej `port`. W tym celu tworzony jest nowy obiekt typu `Integer`, któremu w konstruktorze jest przekazywana wartość `args[1]`, i wywoływana jest metoda `intValue`.

Jeżeli konwersja zakończy się sukcesem (ciąg zawarty w `args[1]` będzie reprezentował prawidłową liczbę), zostanie wykonana dalsza część programu. W przeciwnym razie zostanie zgłoszony wyjątek `NumberFormatException`, który zostanie przechwycony w bloku `catch`. Na ekranie pojawi się wtedy odpowiedni komunikat i aplikacja zakończy działanie.

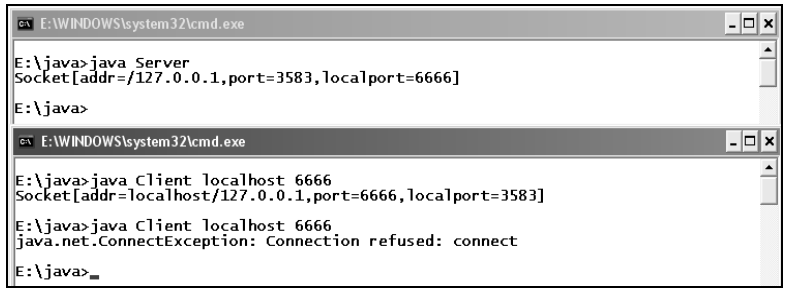
Po wykonaniu opisanych czynności następuje utworzenie gniazda klienckiego o adresie i porcie wskazywanych przez zmienne `host` i `port`. Odbywa się to na takich samych zasadach jak we wcześniejszych ćwiczeniach. Jeśli gniazdo uda się utworzyć, wyświetlane są jego parametry, jeżeli zaś wystąpi jeden z wyjątków, zostanie obsłużony przez odpowiedni blok `catch`.

ĆWICZENIE

3.7 Testowanie połączenia między klientem i serwerem

Przetestuj działanie klienta i serwera z ćwiczeń 3.5 i 3.6.

W jednej konsoli należy wywołać serwer (zacznie wtedy oczekiwać na połączenie), a w drugiej — klienta. Klientowi należy podać odpowiednie argumenty wywołania: jako nazwę `localhost` lub `127.0.0.1` (lub też przypisany do komputera inny adres IP), a jako port — wartość `6666`. Klient nawiąże wtedy połączenie z serwerem. Na konsoli serwera zostaną wyświetlone informacje z gniazda serwerowego, m.in. numer portu, z którego połączył się klient, a na konsoli klienta — informacje z gniazda klienckiego. Można też ponownie wywołać klienta bez uruchomionego serwera, aby zobaczyć obsługę wyjątku `ConnectException` powstałego ze względu na odrzucenie połączenia (rysunek 3.4).



Rysunek 3.4. *Klient i serwer działają zgodnie z założeniami*

Java

Ćwiczenia zaawansowane

Rzadko zdarza się, by jeden język programowania był w stanie zaspokoić niemal wszystkie potrzeby obsługi bardzo różnych urządzeń i systemów operacyjnych bez konieczności żmudnego dostosowywania. Java znakomicie spełnia te wymagania i właśnie dlatego każdy programista — nawet taki, który używa w swojej pracy innych języków programowania — powinien poznać ją bardzo dokładnie. A najłatwiejszym i najbardziej efektywnym sposobem poszerzenia wiedzy w dziedzinie programowania jest przećwiczenie rozmaitych przypadków konkretnych zastosowań danego języka. Jeśli zetknąłeś się już kiedyś z Javą, dobrze znasz jej podstawy, lecz zależy Ci na opanowaniu szerszego spektrum możliwości, powinieneś koniecznie sięgnąć po książkę *Java. Ćwiczenia zaawansowane. Wydanie II*.

Znajdziesz tu zestaw niemal stu ćwiczeń pomagających zrozumieć takie zagadnienia, jak programowanie współbieżne i obsługa wątków, budowanie aplikacji wielowątkowych z interfejsem graficznym, nawiązywanie połączeń sieciowych, komunikacja sieciowa z użyciem protokołu wymiany danych, serwery wielowątkowe, współpraca z relacyjnymi bazami danych czy obsługa zapytań SQL. Nauczysz się uruchamiać, przerywać i synchronizować wątki oraz poznasz sposób działania gniazd w Javie i gniazd serwerowych. Dowiesz się więcej o przesyłaniu danych w sieci, serwerach wielowątkowych i sterowaniu serwerem z konsoli. Zobaczysz, jak powinno wyglądać budowanie aplikacji sieciowych z interfejsem graficznym w Javie oraz o czym koniecznie trzeba pamiętać. Ponadto jasne staną się dla Ciebie wszystkie tajniki komunikacji z bazą danych oraz sposób obsługi zapytań w SQL.

Błyskawicznie opanuj nowe umiejętności i zaimponuj klientom!

- **Programowanie współbieżne**
- **Synchronizacja wątków**
- **Programowanie sieciowe**
- **Transmisja danych w modelu klient-serwer**
- **Aplikacje sieciowe z interfejsem graficznym**
- **Współpraca z bazami danych**

ISBN 978-83-246-3497-2

Cena 29,90 zł