# Programowanie rozproszone w języku Java

Wojciech Rząsa
wrzasa@prz-rzeszow.pl

Katedra Informatyki i Automatyki, Politechnika Rzeszowska

29 kwietnia 2016

# Plan

# Plan

# Programowanie rozproszone

- Osobne procesory
- Rozdzielona przestrzeń adresowa
- Komunikacja poprzez sieć
- Duży narzut komunikacyjny

# Modele programowania rozproszonego

- Message passing (przekazywanie wiadomości)
- Remote procedure call (zdalne wywoływanie procedur)

# Omówione mechanizmy

- Komunikacja sieciowa – gniazda (*sockets*)
- Zdalne wywołanie metod (RMI, odpowiednik RPC)

# Plan

# Możliwości

- Przesyłanie danych pomiędzy procesami

# Gniazdo

- Punkt końcowy kanału komunikacyjnego
- Każda strona ma własne gniazdo
- Pozwala wykonywać podstawowe operacje wejścia/wyjścia
- Operacje realizuje system operacyjny
- Komunikacja przez sieć
- Komunikacja lokalna

# Komunikacja przy użyciu gniazd

- Model klient-serwer

# Komunikacja przy użyciu gniazd

- Model klient-serwer
- Rozpoczęcie komunikacji

# Komunikacja przy użyciu gniazd

- Model klient-serwer
- Rozpoczęcie komunikacji
  - Pasywne (nasłuchiwanie) – serwer

# Komunikacja przy użyciu gniazd

- Model klient-serwer
- Rozpoczęcie komunikacji
  - Pasywne (nasłuchiwanie) – serwer
  - Aktywne (inicjowanie połączenia) – klient

# Komunikacja przy użyciu gniazd

- Model klient-serwer
- Rozpoczęcie komunikacji
  - Pasywne (nasłuchiwanie) – serwer
  - Aktywne (inicjowanie połączenia) – klient
- Typy gniazd

# Komunikacja przy użyciu gniazd

- Model klient-serwer
- Rozpoczęcie komunikacji
  - Pasywne (nasłuchiwanie) – serwer
  - Aktywne (inicjowanie połączenia) – klient
- Typy gniazd
  - Determinują protokół warstwy transportowej

# Komunikacja przy użyciu gniazd

- Model klient-serwer
- Rozpoczęcie komunikacji
  - Pasywne (nasłuchiwanie) – serwer
  - Aktywne (inicjowanie połączenia) – klient
- Typy gniazd
  - Determinują protokół warstwy transportowej
  - gniazda *strumieniowe*
    - protokół TCP
    - komunikacja niezawodna
    - dane jako strumień bajtów

# Komunikacja przy użyciu gniazd

- Model klient-serwer
- Rozpoczęcie komunikacji
  - Pasywne (nasłuchiwanie) – serwer
  - Aktywne (inicjowanie połączenia) – klient
- Typy gniazd
  - Determinują protokół warstwy transportowej
  - gniazda *strumieniowe*
    - protokół TCP
    - komunikacja niezawodna
    - dane jako strumień bajtów
  - gniazda *datagramowe*
    - protokół UDP
    - komunikacja zawodna
    - dane jako datagram

# Komunikacja przy użyciu gniazd

- Model klient-serwer
- Rozpoczęcie komunikacji
  - Pasywne (nasłuchiwanie) – serwer
  - Aktywne (inicjowanie połączenia) – klient
- Typy gniazd
  - Determinują protokół warstwy transportowej
  - gniazda *strumieniowe*
    - protokół TCP
    - komunikacja niezawodna
    - dane jako strumień bajtów
  - gniazda *datagramowe*
    - protokół UDP
    - komunikacja zawodna
    - dane jako datagram
  - gniazda *surowe* (ang. *raw sockets*), niedostępne w Javie

# Porty

- Adres IP – identyfikuje maszynę

# Porty

- Adres IP – identyfikuje maszynę
- Numer portu – identyfikuje proces na maszynie

# Porty

- Adres IP – identyfikuje maszynę
- Numer portu – identyfikuje proces na maszynie
  - numery od 0 do 65536

# Porty

- Adres IP – identyfikuje maszynę
- Numer portu – identyfikuje proces na maszynie
  - numery od 0 do 65536
  - 0 zarezerwowane

# Porty

- Adres IP – identyfikuje maszynę
- Numer portu – identyfikuje proces na maszynie
  - numery od 0 do 65536
  - 0 zarezerwowane
  - od 1 do 1024 tylko administrator

# Porty

- Adres IP – identyfikuje maszynę
- Numer portu – identyfikuje proces na maszynie
  - numery od 0 do 65536
  - 0 zarezerwowane
  - od 1 do 1024 tylko administrator
  - UDP i TCP mają osobną pulę portów

# Porty

- Adres IP – identyfikuje maszynę
- Numer portu – identyfikuje proces na maszynie
  - numery od 0 do 65536
  - 0 zarezerwowane
  - od 1 do 1024 tylko administrator
  - UDP i TCP mają osobną pulę portów
- Serwer – nasłuchuje na wybranym porcie

# Porty

- Adres IP – identyfikuje maszynę
- Numer portu – identyfikuje proces na maszynie
  - numery od 0 do 65536
  - 0 zarezerwowane
  - od 1 do 1024 tylko administrator
  - UDP i TCP mają osobną pulę portów
- Serwer – nasłuchuje na wybranym porcie
- Klient – używając jednego z lokalnych portów łączy się z wybranym IP i portem

# Gniazda klienckie (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Klient tworzy gniazdo

# Gniazda klienckie (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Klient tworzy gniazdo
  - Podaje adres IP i numer portu z którym się łączy

# Gniazda klienckie (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Klient tworzy gniazdo
  - Podaje adres IP i numer portu z którym się łączy
  - Może podać lokalny numer portu

# Gniazda klienckie (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Klient tworzy gniazdo
  - Podaje adres IP i numer portu z którym się łączy
  - Może podać lokalny numer portu
- Jeśli połączenie jest przyjęte, może pisać i czytać z gniazda

# Gniazda serwera (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Serwer tworzy gniazdo

# Gniazda serwera (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Serwer tworzy gniazdo
  - Podaje numer portu z którym ma się połączyć klient

# Gniazda serwera (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Serwer tworzy gniazdo
  - Podaje numer portu z którym ma się połączyć klient
- Czeka na połączenie klienta

# Gniazda serwera (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Serwer tworzy gniazdo
  - Podaje numer portu z którym ma się połączyć klient
- Czeka na połączenie klienta
- Po nadejściu i nawiązaniu nowego połączenia tworzy **nowe gniazdo**

# Gniazda serwera (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Serwer tworzy gniazdo
  - Podaje numer portu z którym ma się połączyć klient
- Czeka na połączenie klienta
- Po nadejściu i nawiązaniu nowego połączenia tworzy **nowe gniazdo**
- **nowe gniazdo** jest wykorzystywane do komunikacji z klientem

# Gniazda serwera (TCP)


[http://java.sun.com/docs/books/tutorial/networking/sockets/definition.html]

- Serwer tworzy gniazdo
  - Podaje numer portu z którym ma się połączyć klient
- Czeka na połączenie klienta
- Po nadejściu i nawiązaniu nowego połączenia tworzy **nowe gniazdo**
- **nowe gniazdo** jest wykorzystywane do komunikacji z klientem
- „stare" gniazdo może być równocześnie używane do dalszego nasłuchiwania

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

[1] http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html

[2] http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)
  - `Socket` gniazdo komunikacyjne klienta i serwera

[1] `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html`

[2] `http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html`

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

  `Socket` gniazdo komunikacyjne klienta i serwera

  `ServerSocket` gniazdo do nasłuchiwania (dla serwera)

[1] `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html`

[2] `http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html`

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

  `Socket` gniazdo komunikacyjne klienta i serwera

  `ServerSocket` gniazdo do nasłuchiwania (dla serwera)
- Komunikacja datagramowa (UDP)

[1] `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html`

[2] `http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html`

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

  `Socket` gniazdo komunikacyjne klienta i serwera

  `ServerSocket` gniazdo do nasłuchiwania (dla serwera)
- Komunikacja datagramowa (UDP)

  `DatagramSocket` gniazdo komunikacyjne UDP

[1] http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html

[2] http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

  `Socket` gniazdo komunikacyjne klienta i serwera

  `ServerSocket` gniazdo do nasłuchiwania (dla serwera)

- Komunikacja datagramowa (UDP)

  `DatagramSocket` gniazdo komunikacyjne UDP

  - Metody `send` i `receive`

[1] `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html`

[2] `http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html`

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

  `Socket` gniazdo komunikacyjne klienta i serwera

  `ServerSocket` gniazdo do nasłuchiwania (dla serwera)

- Komunikacja datagramowa (UDP)

  `DatagramSocket` gniazdo komunikacyjne UDP

  - Metody `send` i `receive`
  - `DatagramPacket`

[1] `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html`

[2] `http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html`

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

  `Socket` gniazdo komunikacyjne klienta i serwera

  `ServerSocket` gniazdo do nasłuchiwania (dla serwera)

- Komunikacja datagramowa (UDP)

  `DatagramSocket` gniazdo komunikacyjne UDP

  - Metody `send` i `receive`
  - `DatagramPacket`
  - Dokumentacja[1]

[1] `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html`

[2] `http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html`

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

  `Socket` gniazdo komunikacyjne klienta i serwera

  `ServerSocket` gniazdo do nasłuchiwania (dla serwera)

- Komunikacja datagramowa (UDP)

  `DatagramSocket` gniazdo komunikacyjne UDP

  - Metody `send` i `receive`
  - `DatagramPacket`
  - Dokumentacja[1]
  - Tutorial[2]

[1] `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html`

[2] `http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html`

# Gniazda w Javie

- Komunikacja strumieniowa (TCP)

  `Socket` gniazdo komunikacyjne klienta i serwera

  `ServerSocket` gniazdo do nasłuchiwania (dla serwera)

- Komunikacja datagramowa (UDP)

  `DatagramSocket` gniazdo komunikacyjne UDP

  - Metody `send` i `receive`
  - `DatagramPacket`
  - Dokumentacja[1]
  - Tutorial[2]

  `MulticastSocket` gniazdo komunikacyjne multicast

[1] `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html`

[2] `http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html`

# Użycie gniazda klienta

```
1        socket = new Socket(adres, port);
2        PrintWriter out =
3            new PrintWriter(socket.getOutputStream(), true);
4        BufferedReader in = new BufferedReader(new
5            InputStreamReader(socket.getInputStream()));
6        System.out.println(in.readLine());
7        out.println("Cześć! Udało się!");
8        System.out.println(in.readLine());
```

Użycie gniazda klienta

# Implementacja klienta

```
1  import java.net.*; import java.io.*;
2  public class Client {
3    public static void main(String args[]) {
4      String adres = args[0];
5      int port = Integer.parseInt(args[1]);
6      Socket socket = null;
7      try {
8        socket = new Socket(adres, port);
9        PrintWriter out =
10           new PrintWriter(socket.getOutputStream(), true);
11       BufferedReader in = new BufferedReader(new
12           InputStreamReader(socket.getInputStream()));
13       System.out.println(in.readLine());
14       out.println("Cześć! Udało się!");
15       System.out.println(in.readLine());
16     } catch(UnknownHostException e) { e.printStackTrace(); }
17     catch(IOException e) { e.printStackTrace(); }
18     finally { try { socket.close(); }
19               catch(IOException e) { e.printStackTrace(); }
20     }
21   }
22 }
```

# Użycie gniazda serwera

```
1        ssocket = new ServerSocket(1234);
2        while(true) {
3          Socket s = ssocket.accept();
4          serveClient(s);
5        }
```

Użycie gniazda serwera

# Użycie gniazda serwera

```
1        ssocket = new ServerSocket(1234);
2        while(true) {
3          Socket s = ssocket.accept();
4          serveClient(s);
5        }
```

Użycie gniazda serwera

```
1    private static void serveClient(Socket s) {
2      new Server(s).start();
3    }
```

Obsługa klienta w nowym wątku

# Implementacja serwera

```
1 import java.net.*;
2 import java.io.*;
3 public class Server extends Thread {
4   private Socket csocket;
5   Server(Socket csocket) {
6     super(); this.csocket = csocket;
7   }
```

Implementacja serwera (początek)

# Implementacja serwera

```
1        PrintWriter out = new PrintWriter(
2            csocket.getOutputStream(), true);
3        BufferedReader in = new BufferedReader(new
4            InputStreamReader(csocket.getInputStream()));
5        out.println("==> Cześć "+ip+"! Udało Ci się połączyć!");
6        String line;
7        while((line = in.readLine()) != null) {
8          out.println("==> Powiedziałeś: " + line);
9        }
```

Interakcja z klientem

# Implementacja serwera

```
1  public void run() {
2    InetAddress ip = csocket.getInetAddress();
3    System.out.println("Połączenie z "+ip);
4    try {
5      PrintWriter out = new PrintWriter(
6          csocket.getOutputStream(), true);
7      BufferedReader in = new BufferedReader(new
8          InputStreamReader(csocket.getInputStream()));
9      out.println("==> Cześć "+ip+"! Udało Ci się połączyć!");
10     String line;
11     while((line = in.readLine()) != null) {
12       out.println("==> Powiedziałeś: " + line);
13     }
14   } catch(IOException e) {
15     e.printStackTrace();
16   } finally {
17     try { csocket.close(); }
18     catch(IOException e) { e.printStackTrace(); }
19   }
20   System.out.println("Koniec połączenia z "+ip);
21 }
```

Kompletna metoda run

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej
- Pozwalają przesyłać komunikaty pomiędzy procesami

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej
- Pozwalają przesyłać komunikaty pomiędzy procesami
- Obsługują różne protokoły warstwy transportowej

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej
- Pozwalają przesyłać komunikaty pomiędzy procesami
- Obsługują różne protokoły warstwy transportowej
- Interfejs obiektowy w Javie

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej
- Pozwalają przesyłać komunikaty pomiędzy procesami
- Obsługują różne protokoły warstwy transportowej
- Interfejs obiektowy w Javie
- Nie zapomnieć o zamykaniu gniazd i strumieni (wycieki pamięci)!

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej
- Pozwalają przesyłać komunikaty pomiędzy procesami
- Obsługują różne protokoły warstwy transportowej
- Interfejs obiektowy w Javie
- Nie zapomnieć o zamykaniu gniazd i strumieni (wycieki pamięci)!
- Aplikacja rozproszona wymaga zdefiniowania *protokołu komunikacyjnego*

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej
- Pozwalają przesyłać komunikaty pomiędzy procesami
- Obsługują różne protokoły warstwy transportowej
- Interfejs obiektowy w Javie
- Nie zapomnieć o zamykaniu gniazd i strumieni (wycieki pamięci)!
- Aplikacja rozproszona wymaga zdefiniowania *protokołu komunikacyjnego*
  - Następstwa wymienianych komunikatów

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej
- Pozwalają przesyłać komunikaty pomiędzy procesami
- Obsługują różne protokoły warstwy transportowej
- Interfejs obiektowy w Javie
- Nie zapomnieć o zamykaniu gniazd i strumieni (wycieki pamięci)!
- Aplikacja rozproszona wymaga zdefiniowania *protokołu komunikacyjnego*
  - Następstwa wymienianych komunikatów
  - Znaczenia wymienianych komunikatów

# Gniazda i komunikacja

- Punkty końcowe komunikacji sieciowej
- Pozwalają przesyłać komunikaty pomiędzy procesami
- Obsługują różne protokoły warstwy transportowej
- Interfejs obiektowy w Javie
- Nie zapomnieć o zamykaniu gniazd i strumieni (wycieki pamięci)!
- Aplikacja rozproszona wymaga zdefiniowania *protokołu komunikacyjnego*
  - Następstwa wymienianych komunikatów
  - Znaczenia wymienianych komunikatów
- Protokoły tekstowe i binarne

# Plan

# Zdalne wywołanie metod (RMI)

- Remote Method Invocation

# Zdalne wywołanie metod (RMI)

- Remote Method Invocation
- Sposób tworzenia aplikacji rozproszonych

# Zdalne wywołanie metod (RMI)

- Remote Method Invocation
- Sposób tworzenia aplikacji rozproszonych
- Wyższy poziom niż gniazda

# Zdalne wywołanie metod (RMI)

- Remote Method Invocation
- Sposób tworzenia aplikacji rozproszonych
- Wyższy poziom niż gniazda
- Bazuje na gniazdach

# Zdalne wywołanie metod (RMI)

- Remote Method Invocation
- Sposób tworzenia aplikacji rozproszonych
- Wyższy poziom niż gniazda
- Bazuje na gniazdach
- Koncepcja oparta na RPC (Remote Procedure Call)

# Idea RMI

- Wywoływanie metod
- Na obiektach zdalnych
  - Na innej maszynie wirtualnej
  - Na innej maszynie fizycznej
- Implementacja w Javie

# Wywołanie metody na zdalnym obiekcie

- Zlokalizowanie obiektu – rejestr RMI

# Wywołanie metody na zdalnym obiekcie

- Zlokalizowanie obiektu – rejestr RMI
- Komunikacja z obiektem

# Wywołanie metody na zdalnym obiekcie

- Zlokalizowanie obiektu – rejestr RMI
- Komunikacja z obiektem
- Zdalne załadowanie definicji klasy dla obiektu

# Zlokalizowanie obiektu

- Rejestr `rmiregistry`

# Zlokalizowanie obiektu

- Rejestr `rmiregistry`
- Obiekty rejestrują zdalne interfejsy w rejestrze

# Zlokalizowanie obiektu

- Rejestr `rmiregistry`
- Obiekty rejestrują zdalne interfejsy w rejestrze
- Pobranie referencji z rejestru, wywołanie metody

# Zlokalizowanie obiektu

- Rejestr `rmiregistry`
- Obiekty rejestrują zdalne interfejsy w rejestrze
- Pobranie referencji z rejestru, wywołanie metody
- Użycie zdalnej referencji z poprzedniego wywołania

# Komunikacja z obiektem

- Realizowana przez RMI
- Niewidoczna dla użytkownika
- Ukryta za zwykłym wywołaniem metody na obiekcie

# Interfejsy

- Wywołujący zdalną metodę obiektu nie musi znać jego implementacji...

# Interfejsy

- Wywołujący zdalną metodę obiektu nie musi znać jego implementacji...
- ...ale musi znać jego interfejs

# Interfejsy

- Wywołujący zdalną metodę obiektu nie musi znać jego implementacji...
- ...ale musi znać jego interfejs
- Zdalne wywołanie wymaga znajomości zdalnego interfejsu

# Interfejsy

- Wywołujący zdalną metodę obiektu nie musi znać jego implementacji...
- ...ale musi znać jego interfejs
- Zdalne wywołanie wymaga znajomości zdalnego interfejsu
- Zdalne interfejsy – ustalone i współdzielone przez obie strony

# Zdalny interfejs w Javie

- Interfejs

# Zdalny interfejs w Javie

- Interfejs
- Dziedziczący po `java.rmi.Remote`

# Zdalny interfejs w Javie

- Interfejs
- Dziedziczący po `java.rmi.Remote`
- Każda metoda może wygenerować `java.rmi.RemoteException`

# Implementacja zdalnych metod

- Klasa Javy

# Implementacja zdalnych metod

- Klasa Javy
- Implementująca jeden lub więcej zdalny interfejs

# Implementacja zdalnych metod

- Klasa Javy
- Implementująca jeden lub więcej zdalny interfejs
- Może implementować wiele innych metod

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję
  - istnieje jedna kopia obiektu

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję
  - istnieje jedna kopia obiektu
- Obiekt lokalny

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję
  - istnieje jedna kopia obiektu
- Obiekt lokalny
  - serializowany, przesyłany i deserializowany

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję
  - istnieje jedna kopia obiektu
- Obiekt lokalny
  - serializowany, przesyłany i deserializowany
  - tworzona zdalna kopia obiektu

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję
  - istnieje jedna kopia obiektu
- Obiekt lokalny
  - serializowany, przesyłany i deserializowany
  - tworzona zdalna kopia obiektu
  - do użycia obiektu konieczna jest definicja jego klasy!

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję
  - istnieje jedna kopia obiektu
- Obiekt lokalny
  - serializowany, przesyłany i deserializowany
  - tworzona zdalna kopia obiektu
  - do użycia obiektu konieczna jest definicja jego klasy!
  - klasa musi implementować `java.io.Serializable`

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję
  - istnieje jedna kopia obiektu
- Obiekt lokalny
  - serializowany, przesyłany i deserializowany
  - tworzona zdalna kopia obiektu
  - do użycia obiektu konieczna jest definicja jego klasy!
  - klasa musi implementować `java.io.Serializable`
- W czasie działania konieczne jest załadowanie definicji nieznanej klasy!

# Obiekty zdalne i lokalne w wywołaniach

- Parametrem zdalnej metody może być obiekt
- Zdalna metoda może zwracać obiekt
- Do kompilacji wystarczy znać interfejs tego obiektu
- Obiekt zdalny
  - przekazany przez zdalną referencję
  - istnieje jedna kopia obiektu
- Obiekt lokalny
  - serializowany, przesyłany i deserializowany
  - tworzona zdalna kopia obiektu
  - do użycia obiektu konieczna jest definicja jego klasy!
  - klasa musi implementować `java.io.Serializable`
- W czasie działania konieczne jest załadowanie definicji nieznanej klasy!
- Dynamiczne ładowanie konieczne zarówno przez klienta jak i serwer

# Dynamiczne ładowanie kodu klas

- Z określonego miejsca na dysku (wymaga wielu kopii plików `class`)

# Dynamiczne ładowanie kodu klas

- Z określonego miejsca na dysku (wymaga wielu kopii plików `class`)
- RMI potrafi pobrać potrzebne klasy np. przez HTTP (tzw. `codebase`)

# Dynamiczne ładowanie kodu klas

- Z określonego miejsca na dysku (wymaga wielu kopii plików `class`)
- RMI potrafi pobrać potrzebne klasy np. przez HTTP (tzw. `codebase`)
- Poza rejestrem zdalnych metod konieczne repozytorium z implementacją

# `rmiregistry` i współdzielone interfejsy

- `rmiregistry` porzebuje plików `class` dla interfejsów współdzielonych

# `rmiregistry` i współdzielone interfejsy

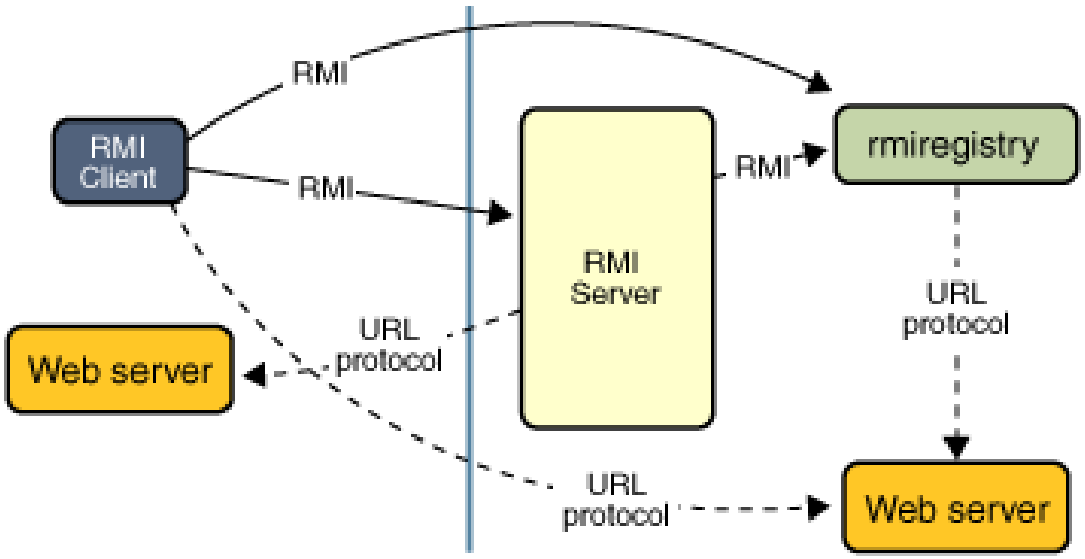
- `rmiregistry` porzebuje plików `class` dla interfejsów współdzielonych
- Mogą być dostępne lokalnie w `CLASSPATH`

# `rmiregistry` i współdzielone interfejsy

- `rmiregistry` porzebuje plików `class` dla interfejsów współdzielonych
- Mogą być dostępne lokalnie w `CLASSPATH`
- Mogą być ładowane dynamicznie (z `codebase` serwera)

# Działanie RMI

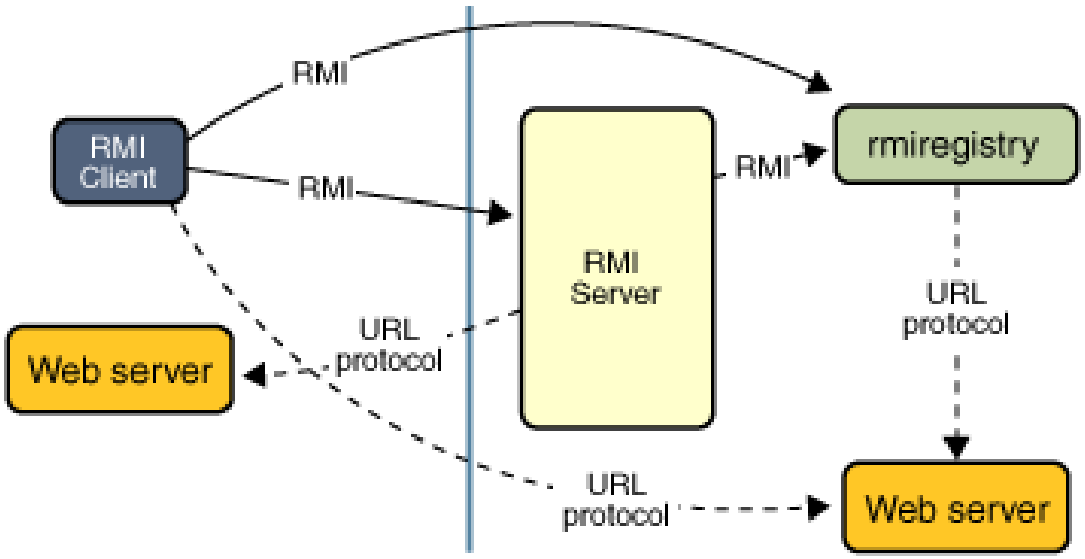
1. `rmiregistry`




[http://docs.oracle.com/javase/tutorial/rmi/overview.html]

# Działanie RMI

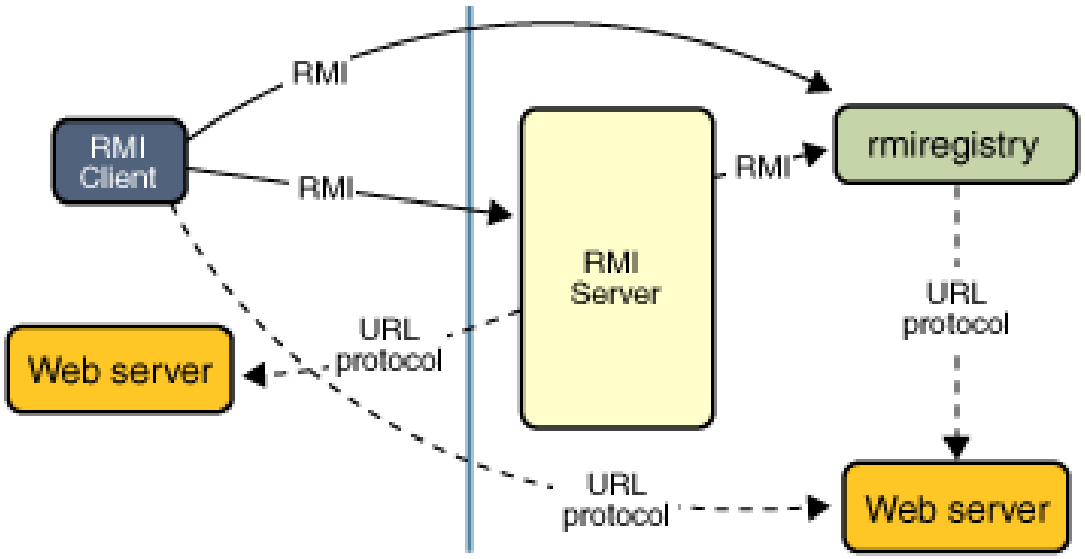
1. `rmiregistry`
2. `codebase` serwera na serwerze HTTP




[http://docs.oracle.com/javase/tutorial/rmi/overview.html]

# Działanie RMI

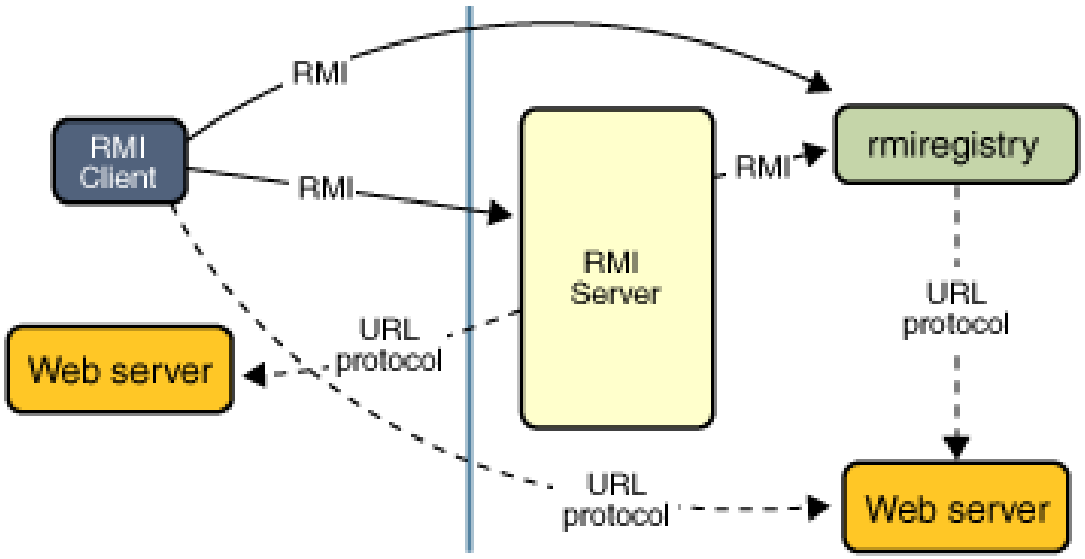
1. `rmiregistry`
2. `codebase` serwera na serwerze HTTP
3. Serwer




[http://docs.oracle.com/javase/tutorial/rmi/overview.html]

# Działanie RMI

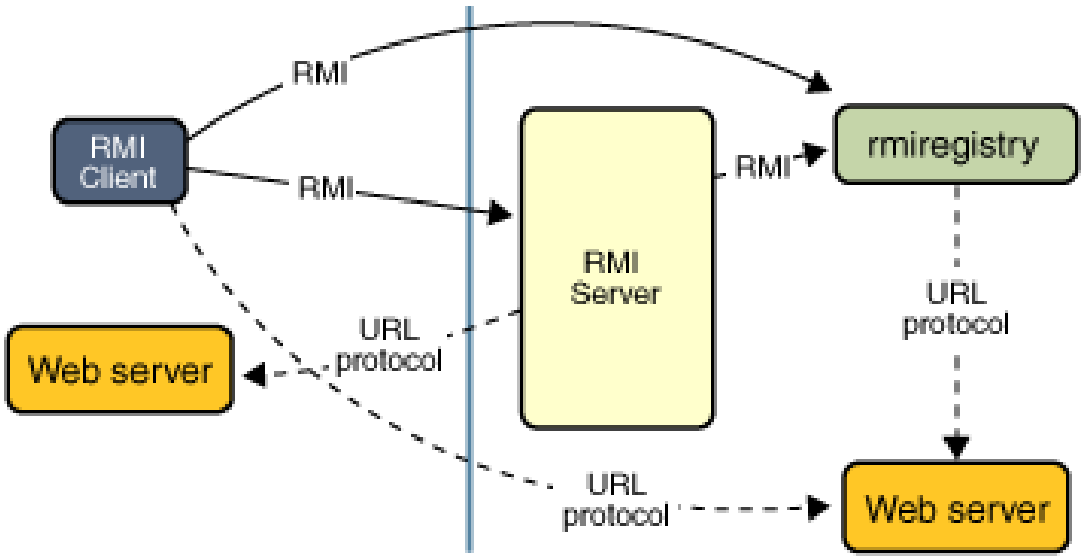
1. `rmiregistry`
2. codebase serwera na serwerze HTTP
3. Serwer
4. codebase klienta na serwerze HTTP




[http://docs.oracle.com/javase/tutorial/rmi/overview.html]

# Działanie RMI

1. `rmiregistry`
2. codebase serwera na serwerze HTTP
3. Serwer
4. codebase klienta na serwerze HTTP
5. Klient




[http://docs.oracle.com/javase/tutorial/rmi/overview.html]

# Komunikacja ze zdalnym obiektem

- Serwer rejestruje tzw. `stub` w `rmiregistry`

# Komunikacja ze zdalnym obiektem

- Serwer rejestruje tzw. `stub` w `rmiregistry`
- `stub` implementuje zdalny interfejs

# Komunikacja ze zdalnym obiektem

- Serwer rejestruje tzw. `stub` w `rmiregistry`
- `stub` implementuje zdalny interfejs
- `rmiregistry` zwraca `stub` do klienta

# Komunikacja ze zdalnym obiektem

- Serwer rejestruje tzw. `stub` w `rmiregistry`
- `stub` implementuje zdalny interfejs
- `rmiregistry` zwraca `stub` do klienta
- Wywołania metod `stuba` są przekazywane do zdalnego obiektu

# Komunikacja ze zdalnym obiektem

- Serwer rejestruje tzw. `stub` w `rmiregistry`
- `stub` implementuje zdalny interfejs
- `rmiregistry` zwraca `stub` do klienta
- Wywołania metod `stuba` są przekazywane do zdalnego obiektu
- Wyniki metod zdalnego obiektu są przekazywane do `stuba` i zwracane do klienta

# Komunikacja ze zdalnym obiektem

- Serwer rejestruje tzw. `stub` w `rmiregistry`
- `stub` implementuje zdalny interfejs
- `rmiregistry` zwraca `stub` do klienta
- Wywołania metod `stuba` są przekazywane do zdalnego obiektu
- Wyniki metod zdalnego obiektu są przekazywane do `stuba` i zwracane do klienta
- Komunikacja ukryta przez klientem

# Bezpieczeństwo

- Zdalnie załadowany kod może być niebezpieczny

# Bezpieczeństwo

- Zdalnie załadowany kod może być niebezpieczny
- Bezpieczeństwo zapewnia `SecurityManager`

# Bezpieczeństwo

- Zdalnie załadowany kod może być niebezpieczny
- Bezpieczeństwo zapewnia `SecurityManager`
- Wymagany do załadowania zdalnego kodu

# Bezpieczeństwo

- Zdalnie załadowany kod może być niebezpieczny
- Bezpieczeństwo zapewnia `SecurityManager`
- Wymagany do załadowania zdalnego kodu
- Konfigurowany z pliku

# Bezpieczeństwo

- Zdalnie załadowany kod może być niebezpieczny
- Bezpieczeństwo zapewnia `SecurityManager`
- Wymagany do załadowania zdalnego kodu
- Konfigurowany z pliku
- Ogranicza działania aplikacji

# Bezpieczeństwo

- Zdalnie załadowany kod może być niebezpieczny
- Bezpieczeństwo zapewnia `SecurityManager`
- Wymagany do załadowania zdalnego kodu
- Konfigurowany z pliku
- Ogranicza działania aplikacji
- Uruchamiając klienta i serwer trzeba podać ścieżkę do *policy file*

# Tworzenie aplikacji RMI

- Uzgodnienie zdalnych interfejsów

# Tworzenie aplikacji RMI

- Uzgodnienie zdalnych interfejsów
- Implementacja zdalnych interfejsów (serwer)

# Tworzenie aplikacji RMI

- Uzgodnienie zdalnych interfejsów
- Implementacja zdalnych interfejsów (serwer)
- Implementacja zdalnych wywołań (klient)

# Uruchamianie aplikacji RMI

- Kompilacja

# Uruchamianie aplikacji RMI

- Kompilacja
- Udostępnienie w sieci definicji klas klienta

# Uruchamianie aplikacji RMI

- Kompilacja
- Udostępnienie w sieci definicji klas klienta
- Udostępnienie w sieci definicji klas serwera

# Uruchamianie aplikacji RMI

- Kompilacja
- Udostępnienie w sieci definicji klas klienta
- Udostępnienie w sieci definicji klas serwera
- Start `rmiregistry`

# Uruchamianie aplikacji RMI

- Kompilacja
- Udostępnienie w sieci definicji klas klienta
- Udostępnienie w sieci definicji klas serwera
- Start `rmiregistry`
- Start serwera (rejestruje zdalne obiekty w `rmiregistry`)

# Uruchamianie aplikacji RMI

- Kompilacja
- Udostępnienie w sieci definicji klas klienta
- Udostępnienie w sieci definicji klas serwera
- Start `rmiregistry`
- Start serwera (rejestruje zdalne obiekty w `rmiregistry`)
- Start klienta (klientów)

# Przykładowa aplikacja

- Zgłaszanie obecności osoby

# Przykładowa aplikacja

- Zgłaszanie obecności osoby
- Klient tworzy nowy obiekt `Osoba` wraz z danymi

# Przykładowa aplikacja

- Zgłaszanie obecności osoby
- Klient tworzy nowy obiekt `Osoba` wraz z danymi
- Klient wywołuje zdalną metodę `reportPresenceOf` podając obiekt klasy `Osoba`

# Przykładowa aplikacja

- Zgłaszanie obecności osoby
- Klient tworzy nowy obiekt `Osoba` wraz z danymi
- Klient wywołuje zdalną metodę `reportPresenceOf` podając obiekt klasy `Osoba`
- Serwer zapamiętuje osobę na podstawie danych otrzymanych od klienta

# Współdzielone części kodu

- Przy programowaniu

# Współdzielone części kodu

- Przy programowaniu
  - Zdalny interfejs zgłaszania obeności (`RemotePresenceList`)

# Współdzielone części kodu

- Przy programowaniu
  - Zdalny interfejs zgłaszania obeności (`RemotePresenceList`)
  - Interfejs klasy `Osoba` (`OsobaInterface`)

# Współdzielone części kodu

- Przy programowaniu
  - Zdalny interfejs zgłaszania obeności (`RemotePresenceList`)
  - Interfejs klasy `Osoba` (`OsobaInterface`)
  - Tylko interfejsy!

# Współdzielone części kodu

- Przy programowaniu
  - Zdalny interfejs zgłaszania obeności (`RemotePresenceList`)
  - Interfejs klasy `Osoba` (`OsobaInterface`)
  - Tylko interfejsy!
- Przy uruchomieniu

# Współdzielone części kodu

- Przy programowaniu
  - Zdalny interfejs zgłaszania obeności (`RemotePresenceList`)
  - Interfejs klasy `Osoba` (`OsobaInterface`)
  - Tylko interfejsy!
- Przy uruchomieniu
  - Definicja klasy `Osoba` (skompilowana)

# Współdzielone interfejsy

```
1 package shared;
2 public interface RemotePresenceList
3                  extends java.rmi.Remote {
4
5   public boolean reportPresenceOf(OsobaInterface o)
6                  throws java.rmi.RemoteException;
7
8 }
```

Interfejs zdalnej rejestracji

# Współdzielone interfejsy

```
1 package shared;
2 public interface RemotePresenceList
3                  extends java.rmi.Remote {
4
5   public boolean reportPresenceOf(OsobaInterface o)
6                  throws java.rmi.RemoteException;
7
8 }
```

Interfejs zdalnej rejestracji

```
1 package shared;
2 public interface OsobaInterface {
3   public String getFirstName();
4   public String getLastName();
5   public int getIndexNumber();
6 }
```

Interfejs osoby

# Implementacja zdalnego interfejsu (serwer)

```
1  package server;
2
3  import java.rmi.RemoteException;
4  import java.rmi.registry.LocateRegistry;
5  import java.rmi.registry.Registry;
6  import java.rmi.server.UnicastRemoteObject;
7
8  import shared.*;
9
10 public class PresenceList implements RemotePresenceList {
```

Początek implementacji zdalnego interfejsu

# Implementacja zdalnego interfejsu (serwer)

```
12    public boolean reportPresenceOf(OsobaInterface o) {
13      System.out.print("Zapisano obecność osoby: ");
14      System.out.print(o.getFirstName() + " ");
15      System.out.print(o.getLastName() + " ");
16      System.out.println("indeks: " + o.getIndexNumber());
17
18      return true;
19    }
```

Implementacja metody wywoływanej zdalnie

# Implementacja zdalnego interfejsu (serwer)

```
21  public static void main(String args[]) {
22    if (System.getSecurityManager() == null) {
23      System.setSecurityManager(new SecurityManager());
24    }
25    try {
26      PresenceList list = new PresenceList();
27      RemotePresenceList remote_list = (RemotePresenceList)
28        UnicastRemoteObject.exportObject(list, 0);
29      Registry registry = LocateRegistry.getRegistry();
30      registry.rebind("RemotePresenceList", remote_list);
31      System.out.println("Remote presence list bound");
32    } catch(RemoteException e) {
33      e.printStackTrace();
34    }
35  }
```

Utworzenie obiektu i rejestracja zdalnej metody

# Wywołanie zdalnej metody (klient)

```
1  package client;
2
3  import java.rmi.RemoteException;
4  import java.rmi.NotBoundException;
5  import java.rmi.registry.LocateRegistry;
6  import java.rmi.registry.Registry;
7
8  import shared.*;
9  import client.shared.Student;
10
11 public class ReportStudent {
```

Początek implementacji części klienckiej

# Wywołanie zdalnej metody (klient)

```
13    public static void main(String args[]){
14      if (System.getSecurityManager() == null) {
15        System.setSecurityManager(new SecurityManager());
16      }
17
18      Student ja = new Student("Jan", "Kowalski", 1234);
```

Operacje wstępne

# Wywołanie zdalnej metody (klient)

```
19      try {
20        Registry registry = LocateRegistry.getRegistry(args[0]);
21        RemotePresenceList list = (RemotePresenceList)
22          registry.lookup("RemotePresenceList");
23
24        boolean result = list.reportPresenceOf(ja);
25
26        if (result) {
27          System.out.println("Obecność zarejestrowano poprawnie");
28        } else {
29          System.out.println("System odmówił przyjęcia zgłoszenia");
30        }
31      }
32      catch (RemoteException e) { e.printStackTrace(); }
33      catch (NotBoundException e) { e.printStackTrace(); }
34    }
```

Wywołanie zdalnej metody

# Wywołanie zdalnej metody (klient)

```
1  package client.shared;
2
3  import java.io.Serializable;
4  import shared.*;
5
6  public class Student implements OsobaInterface, Serializable {
7    private String fname, lname;
8    private int index;
9
10   public Student(String fname, String lname, int index) {
11     this.fname = fname; this.lname = lname;
12     this.index = index;
13   }
14
15   public String getFirstName() { return fname; }
16   public String getLastName() { return lname; }
17   public int getIndexNumber() { return index; }
18 }
```

Klasa przekazywana do zdalnej metody

# Przygotowanie do uruchomienia

- Kompilacja kodu współdzielonego, serwera i klienta

# Przygotowanie do uruchomienia

- Kompilacja kodu współdzielonego, serwera i klienta
- Utworzenie archiwów jar skompilowanego kodu:

# Przygotowanie do uruchomienia

- Kompilacja kodu współdzielonego, serwera i klienta
- Utworzenie archiwów jar skompilowanego kodu:
  `shared.jar` współdzielone interfejsy (serwer)

# Przygotowanie do uruchomienia

- Kompilacja kodu współdzielonego, serwera i klienta
- Utworzenie archiwów jar skompilowanego kodu:

  `shared.jar` współdzielone interfejsy (serwer)
  `client_shared.jar` klasy klient potrzebne serwerowi (klient)

# Przygotowanie do uruchomienia

- Kompilacja kodu współdzielonego, serwera i klienta
- Utworzenie archiwów jar skompilowanego kodu:

  `shared.jar` współdzielone interfejsy (serwer)
  `client_shared.jar` klasy klient potrzebne serwerowi (klient)
  ~~`server_shared.jar`~~ ~~klasy klient potrzebne klientowi (serwer)~~

# Przygotowanie do uruchomienia

- Kompilacja kodu współdzielonego, serwera i klienta
- Utworzenie archiwów jar skompilowanego kodu:

  `shared.jar` współdzielone interfejsy (serwer)
  `client_shared.jar` klasy klient potrzebne serwerowi (klient)
  ~~`server_shared.jar`~~ ~~klasy klient potrzebne klientowi (serwer)~~
- Umieszczenie plików jar na serwerach www

# Przygotowanie do uruchomienia

- Kompilacja kodu współdzielonego, serwera i klienta
- Utworzenie archiwów jar skompilowanego kodu:

  `shared.jar` współdzielone interfejsy (serwer)
  `client_shared.jar` klasy klient potrzebne serwerowi (klient)
  ~~`server_shared.jar`~~ ~~klasy klient potrzebne klientowi (serwer)~~
- Umieszczenie plików jar na serwerach www
- Konfiguracja `SecurityManagera` – `client.policy` i `server.policy`

```
1 grant codeBase "file:${user.home}/sciezka/-" {
2   permission java.security.AllPermission;
3 };
```

Przykładowa definicja polityki bezpieczeństwa

# Uruchomienie aplikacji

1. Uruchomenie `rmiregistry` (javase >=7)

```
rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

Uruchomienie `rmiregistry`

`http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/enhancements-7.html` *This change of default value may cause RMI-based applications to break unexpectedly.*

# Uruchomienie aplikacji

1. Uruchomenie `rmiregistry` (javase >=7)

```
rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

Uruchomienie `rmiregistry`

2. Uruchomienie serwera (javase >=7)

```
java -Djava.security.policy=server.policy \
     -Djava.rmi.server.codebase=http://s1/shared.jar \
     -Djava.rmi.server.useCodebaseOnly=false \
     server/PresenceList
```

Uruchomienie serwera

Może być potrzebne też `java.rmi.server.hostname=IP`

# Uruchomienie aplikacji

1. Uruchomenie `rmiregistry` (javase >=7)

```
rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

Uruchomienie `rmiregistry`

2. Uruchomienie serwera (javase >=7)

```
java -Djava.security.policy=server.policy \
     -Djava.rmi.server.codebase=http://s1/shared.jar \
     -Djava.rmi.server.useCodebaseOnly=false \
     server/PresenceList
```

Uruchomienie serwera

3. Uruchomienie klienta (javase >=7)

```
java -Djava.security.policy=client.policy \
     -Djava.rmi.server.codebase=http://s2/client_shared.jar \
     client/ReportStudent adres_serwera_z_rmiregistry
```

Uruchomienie klienta

# RMI – podsumowanie

- Proste wywołanie metody na zdalnym obiekcie

# RMI – podsumowanie

- Proste wywołanie metody na zdalnym obiekcie
- Wymaga sporo przygotowań

# RMI – podsumowanie

- Proste wywołanie metody na zdalnym obiekcie
- Wymaga sporo przygotowań
- Kwestia dostępności kodu

# RMI – podsumowanie

- Proste wywołanie metody na zdalnym obiekcie
- Wymaga sporo przygotowań
- Kwestia dostępności kodu
- Bezpieczeństwo

# RMI – podsumowanie

- Proste wywołanie metody na zdalnym obiekcie
- Wymaga sporo przygotowań
- Kwestia dostępności kodu
- Bezpieczeństwo
- Wysokopoziomowy model programowania rozproszonego

# Plan

# Programowanie rozproszone w Javie (omówione aspekty)

- Obsługa gniazd, wymiana danych
  - Strumieniowych (TCP)
  - Datagramowych (UDP), multicast
  - Brak gniazd „surowych” (*raw*)

# Programowanie rozproszone w Javie (omówione aspekty)

- Obsługa gniazd, wymiana danych
  - Strumieniowych (TCP)
  - Datagramowych (UDP), multicast
  - Brak gniazd „surowych” (*raw*)
- Remote Method Invocation (RMI)
  - Model wysokopoziomowy
  - Narzut „organizacyjny”
  - Gotowy protokół komunikacyjny
  - Rozproszone programowanie obiektowe