

Remote Method Invocation (RMI)

(B)

wg. Elliotte Rusty Harold, „JAVA Programowanie Sieciowe”, RM

(Wyłącznie do użytku lokalnego w celach porównawczych w ramach PZR 420 i PZR 380)

Wersja 1.2/2006

Co to jest zdalne wywoływanie metod ?

Interfejs API Remote Method Invocation pozwala obiektom Javy znajdującym się na różnych hostach na wywoływanie zdalnych metod. Zdalne obiekty są przechowywane w przestrzeni adresowej serwera. Każdy zdalny obiekt implementuje interfejs, który definiuje metody dostępne dla klientów. Klienci wywołują metody zdalnego obiektu tak, jak wywołują metody lokalne.

„Z punktu widzenia programisty zdalne obiekty i metody działają tak, jak ich lokalne odpowiedniki. Wszystkie szczegóły implementacyjne są ukryte. Należy zaimportować pakiet, wyszukać zdalny obiekt w rejestrze i zadbać o przechwytywanie wyjątku RemoteException podczas wywoływania metod tego obiektu. Od tej chwili używanie metod zdalnego obiektu przebiega zupełnie tak jak metod obiektu lokalnego.

Obiekt zdalny to taki obiekt, którego metody można wywoływać z innej maszyny wirtualnej, zwykle działającej w innym komputerze. Aby obiekt stał się zdalnym musi implementować jeden lub więcej zdalnych interfejsów deklarujących metody, które mogą zostać wywołane przez zdalne systemy oraz musi rozszerzać jeden ze zdalnych obiektów (np. UnicastRemoteObject). Szczegóły nawiązywania połączenia między hostami i przesyłania danych są ukryte w klasach RMI”

Bezpieczeństwo

Możliwość wywoływania metod w lokalnych obiektach przez zdalne hosty ma oczywisty wpływ na bezpieczeństwo. Podobnie jak można ograniczyć zakres operacji dozwolonych apletowi, host pozwalający na zdalne wywoływanie metod może ograniczyć operacje przeprowadzane przez zdalne hosty.

Standardowo obiekt SecurityManager sprawdza ładowany kod. Można też zdefiniować niestandardowe menedżery bezpieczeństwa i używać uwierzytelnień oraz autentyfikacji. Można precyzyjnie określać prawa do selektywnego korzystania z zasobów.

Serializacja obiektów.

W przypadku obiektów lokalnych przekazywanie takich obiektów odbywa się przez referencje do obiektu. Referencja do obiektu to w istocie podwójnie pośredni wskaźnik położenia obiektu w pamięci. Stąd niemożność bezpośredniego przekazywania referencji jako parametru zdalnego wywołania. Stosowane są trzy metody:

- W przypadku zmiennych prostych przekazuje się je przez wartość stosując serializację.
- W przypadku obiektów lokalnych zamiast referencji przekazuje się sam obiekt poddając go procesowi serializacji (- obiekt musi wówczas implementować interfejs `java.io.Serializable`).
- W przypadku przekazania referencji do zdalnego obiektu przekazuje się specjalną zdalną referencję (zdalne wskazanie na obiekt) z wykorzystaniem specjalnego protokołu opartego na przesyłaniu komunikatów.

„Serializacja obiektów to mechanizm przekształcania obiektów w strumień bajtów i wysyłania ich do innych komputerów, które odtwarzają pierwotny obiekt. Bajty te można również zapisać na dysku i odczytać później, co pozwala na zapisanie stanu programu (a nawet poszczególnych obiektów).

Ze względów bezpieczeństwa obiekty w Javie muszą spełnić pewne warunki, aby mogły być serializowane. Można serializować wszystkie podstawowe typy danych Javy, ale niezdalne obiekty Javy mogą być serializowane tylko wtedy, gdy implementują interfejs `java.io.Serializable`. Interfejs ten implementują na przykład klasy `String` i `Component`. Klasy pojemnikowe, takie jak `Vector`, są serializowalne wtedy, gdy zawierają serializowalne obiekty. Ponadto serializowalne są

także podklasy serializowalnej klasy. Na przykład klasy `java.lang.Integer` i `java.lang.Float` są serializowalne, ponieważ serializowalna jest klasa `java.lang.Number`, którą rozszerzają. Wyjątki, błędy i inne tego typu obiekty są zawsze serializowalne. Większość komponentów, pojemników oraz zdarzeń AWT i Swing jest serializowalna, natomiast adaptery zdarzeń, filtry obrazów i klasy równorzędne - nie. Strumienie, klasy czytające i zapisujące oraz niemal wszystkie pozostałe klasy wejścia-wyjścia nie są serializowalne. Klasy stanowiące nakładki na podstawowe typy danych są serializowalne, z wyjątkiem `Void`. Serializowalne są klasy pakietu `java.math`, natomiast klasy `java.lang.reflect` - nie. Klasa `URL` jest serializowalna, natomiast `Socket`, `URLConnection` i większość innych klas `java.net` - nie. „

Pod maską

„Podstawowa różnica między obiektami zdalnymi i lokalnymi polega na tym, że zdalne obiekty przebywają w innej maszynie wirtualnej. Zwykle argumenty obiektowe są przekazywane, a wartości zwracane, przez odniesienie do czegoś w maszynie wirtualnej. Nazywamy to *przekazywaniem referencji*. Sposób ten jednak nie działa, gdy metoda wywołująca i wywoływana znajdują się w innych maszynach wirtualnych, na przykład obiekt 243 w jednej maszynie nie ma nic wspólnego z obiektem 243 w drugiej. W istocie sposoby realizacji referencji przez różne maszyny wirtualne mogą być zupełnie różne i ze sobą niezgodne.”

„Do przekazywania argumentów i zwracania wyników służą więc trzy różne mechanizmy. Typy podstawowe (`int`, `boolean`, `double` i tak dalej) są przekazywane przez wartość, podobnie jak w przypadku lokalnych wywołań Javy. Referencje do zdalnych obiektów (to znaczy obiektów implementujących interfejs `Remote`) są przekazywane jako *zdalna referencja*, co pozwala odbiorcy na wywołanie metody zdalnego obiektu. Przypomina to przekazywanie lokalnych referencji obiektowych do lokalnych metod Javy. Obiekty nie implementujące interfejsu `Remote` są przekazywane przez wartość, to znaczy przekazuje się ich pełne kopie, wykorzystując serializację obiektów.”

„Aby zapewnić zgodność z istniejącymi programami i implementacjami Javy oraz usunąć cały proces z pola widzenia programisty, [komunikacja między zdalnym klientem i serwerem jest zaimplementowana jako seria warstw pokazanych na rysunku 18-1.](#)”

„Z punktu widzenia programisty klient porozumiewa się bezpośrednio z serwerem. W rzeczywistości program kliencki porozumiewa się tylko z trzonem, który przekazuje konwersację do warstwy zdalnych referencji, a ta z kolei porozumiewa się z warstwą transportową. Warstwa transportowa w kliencie przekazuje dane przez Internet do warstwy transportowej w serwerze. Warstwa transportowa serwera komunikuje się z warstwą zdalnych referencji serwera, która z kolei porozumiewa się z programem nazywanym szkieletem (ang. *skeleton*). Szkielet porozumiewa się z właściwym serwerem (serwery napisane w Javie 1.2 i nowszych wersjach nie są wyposażone w szkielet). Co do przeciwnego kierunku (serwer do klienta), przepływ danych jest po prostu odwrócony. Logiczny przepływ danych jest poziomy (od klienta do serwera i z powrotem), ale w rzeczywistości dane przepływają pionowo.

Zanim wywołasz metodę w obiekcie, musisz uzyskać referencję do tego obiektu. W tym celu pobierasz ją z rejestru według nazwy. Rejestr to program działający w serwerze. Zawiera on listę wszystkich zdalnych obiektów, które serwer może wyeksportować, oraz ich nazw. Klient łączy się z rejestrem i podaje nazwę zdalnego obiektu, którego potrzebuje. Rejestr odsyła klientowi referencję do tego obiektu, której klient może użyć do wywołania metod w serwerze.”

„W rzeczywistości klient wywołuje tylko lokalne metody w *trzonie* (ang. *stub*). Trzon to lokalny obiekt, który implementuje zdalne interfejsy zdalnego obiektu; oznacza to, że trzon zawiera metody odpowiadające prototypom wszystkich zdalnych metod eksportowanych

przez zdalny obiekt. W rezultacie klient sądzi, że wywołuje metodę w zdalnym obiekcie, ale naprawdę wywołuje równoważną metodę w trzonie.

Trzony są używane przez wirtualną maszynę klienta zamiast prawdziwych obiektów i metod przechowywanych w serwerze; możesz uważać trzon za namiastkę zdalnego obiektu. Kiedy klient wywołuje metodę, trzon przekazuje wywołanie do warstwy zdalnych referencji.

Warstwa zdalnych referencji posługuje się protokołem zdalnych referencji, niezależnym od typu trzonów klienta i szkieletów serwera. Warstwa zdalnych referencji jest odpowiedzialna za rozwikłanie znaczenia poszczególnych zdalnych referencji. Czasem zdalna referencja może odnosić się do wielu różnych maszyn wirtualnych w wielu różnych hostach. W innych sytuacjach referencja może odnosić się do jednej maszyny wirtualnej w lokalnym hoście albo do maszyny wirtualnej w zdalnym hoście. Warstwa zdalnych referencji tłumaczy lokalną referencję do trzonu na zdalną referencję do obiektu w serwerze, niezależnie od składni lub semantyki zdalnej referencji. Następnie przekazuje wywołanie do warstwy transportowej.”



Rysunek 18.1. Model warstwy RMI

..... „Po stronie serwera warstwa transportowa czeka na przychodzące połączenia. Po odebraniu wywołania warstwa transportowa przekazuje je do warstwy zdalnych referencji w serwerze. Warstwa zdalnych referencji przekształca referencje przesłane przez klienta w referencje odpowiadające lokalnej maszynie wirtualnej, a następnie przekazuje je do szkieletu. Szkielet odczytuje argumenty i przekazuje dane programowi serwera, który wywołuje metodę. Jeśli wywołanie zwróci wartość, wartość ta jest przesyłana z powrotem przez szkielet, warstwę zdalnych referencji i warstwę transportową w serwerze, potem przez Internet, a wreszcie przez warstwę transportową, warstwę zdalnych referencji i trzon w kliencie. *W Javie 1.2 i nowszych wersjach warstwa szkieletu jest pomijana, a serwer porozumiewa się bezpośrednio z warstwą zdalnych referencji. Poza tym protokół jest taki sam.*”

Implementacja

„Większość metod potrzebnych do pracy ze zdalnymi obiektami znajduje się w trzech pakietach: `java.rmi`, `java.rmi.server` oraz `java.rmi.registry`.

Pakiet `java.rmi` definiuje klasy, interfejsy i wyjątki, które pojawiają się po stronie klienta. Wykorzystasz je podczas pisania programów używających zdalnych obiektów, ale nie będących zdalnymi obiektami.

Pakiet `java.rmi` definiuje klasy, interfejsy i wyjątki, które pojawiają się po stronie serwera. Wykorzystasz je podczas pisania zdalnych obiektów, które będą wywoływane przez klienty.

Pakiet `java.rmi` definiuje klasy, interfejsy i wyjątki używane do lokalizowania i nazywania zdalnych obiektów. Pakiety te stały się częścią podstawowego API Javy od wersji 1.1.”

Strona serwera

„Aby utworzyć nowy zdalny obiekt, najpierw definiujesz interfejs, który rozszerza interfejs `java.rmi.Remote`. Interfejs `Remote` nie ma żadnych własnych metod; służy tylko do znakowania zdalnych obiektów, aby można je było zidentyfikować jako takie. Za zdalny

obiekt można uważać instancję każdej klasy implementującej interfejs Remote lub dowolny interfejs rozszerzający Remote.

Twój podinterfejs Remote określa, które metody zdalnego obiektu mogą być wywoływane przez klienty. Zdalny obiekt może mieć wiele metod publicznych, ale zdalnie wywoływać można tylko te, które są zadeklarowane w zdalnym interfejsie. Pozostałe metody można wywoływać tylko z maszyny wirtualnej, w której przebywa obiekt.

Każda metoda podinterfejsu musi deklarować zgłaszanie wyjątku RemoteException. RemoteException to klasa nadrzędna większości wyjątków, które mogą zostać zgłoszone podczas używania RMI. Wiele spośród nich ma związek z działaniem zewnętrznych systemów i *sieci*, więc nie masz nad nimi kontroli.”

„**Przykład 18-1** „to prosty interfejs zdalnego obiektu obliczającego liczby Fibonacciego (liczby Fibonacciego to ciąg o postaci 0, 1, 1, 2/3, 5/8/13, ... w którym każda następna liczba stanowi sumę poprzednich dwóch). Ten zdalny obiekt może działać w szybkim serwerze, aby obliczać wyniki na rzecz wolniejszych klientów. Interfejs deklaruje dwie predefiniowane metody getFibonacci() . Pierwsza przyjmuje argument typu int, druga typu BigInteger. Obie metody zwracają liczby typu BigInteger, ponieważ liczby Fibonacciego bardzo szybko osiągają duże wartości. Bardziej skomplikowany obiekt zdalny mógłby zawierać wiele więcej metod.”

Przykład 18-1. Interfejs Fibonacci

```
import java.rmi.*;
import java.math.BigInteger;

public interface Fibonacci extends Remote {

    public BigInteger getFibonacci(int n)
        throws RemoteException;
    public BigInteger getFibonacci(BigInteger n)
        throws RemoteException;
}
```

„Interfejs ten nie zawiera żadnych informacji o używanych metodach obliczeniowych. Liczby można wyznaczać bezpośrednio, korzystając z metod klasy java.math.BigInteger. Można je wyznaczać równie łatwo za pomocą bardziej wydajnej klasy com.ibm.BigInteger z pakietu alphaWorks IBM-a (<http://www.alphaworks.ibm.com/tech/bigdecimal>). Można wyznaczać je za pomocą liczb typu int dla małych wartości n i BigInteger dla dużych wartości n. Każde obliczenie można wykonywać natychmiast albo użyć stałej liczby wątków, aby ograniczyć obciążenie serwera przez zdalny obiekt. Obliczone wartości można buforować (aby przyspieszyć odpowiedzi na przyszłe żądania) albo wewnętrznie, albo zewnętrznie - w pliku lub bazie danych. Można skorzystać ze wszystkich tych rozwiązań razem lub z osobna. Klient nie musi wiedzieć, jak serwer otrzymuje wynik, jeśli tylko obliczenia są poprawne.”

„Następną czynnością jest zdefiniowanie klasy która implementuje ten zdalny interfejs. Klasa ta powinna rozszerzać klasę java.rmi.server.UnicastRemoteObject, bezpośrednio lub pośrednio (na przykład przez rozszerzenie innej klasy która rozszerza UnicastRemoteObject):

```
public class UnicastRemoteObject extends RemoteServer
```

... „Klasa UnicastRemoteObject zawiera kilka metod, dzięki którym RMI może działać. W szczególności szereguje i deszereguje zdalne referencje do obiektu. Szeregowanie (ang. *marshaling*) to proces przekształcania argumentów i wartości zwrotnych w strumień bajtów, które można przesłać siecią.

Deszeregowanie (ang. *unmarshaling*) to proces odwrotny: konwersja strumienia bajtów w grupę argumentów lub w wartość zwrótną.”

„Jeśli rozszerzanie klasy `UnicastRemoteObject` jest niedogodne - na przykład dlatego, że wolałbyś rozszerzyć inną klasę - wówczas możesz wyeksportować swój obiekt jako obiekt zdalny za pomocą jednej ze statycznych metod `UnicastRemoteObject.exportObject()`:

```
public static RemoteStub exportObject(Remote object)
    throws RemoteException

public static Remote exportObject(Remote object, int port)
    throws RemoteException // Java 1.2

public static Remote exportObject(Remote object, int port)
    RMIClientSocketFactory csf, RMIServerSocketFactory ssfj
    throws RemoteException // Java 1.2
```

Metody te tworzą zdalny obiekt, który wykorzystuje twój obiekt do wykonania określonych operacji. Przypomina to sposób, w jaki można użyć obiektu `Runnable` do określania zadań wątku, kiedy niedogodnie byłoby rozszerzać klasę `Thread`.

W Javie 1.2 można wybrać port, w którym działa serwer, a nawet fabryki gniazd używanych do nawiązywania połączeń. W Javie 1.1 można użyć tylko losowo wybranego portu, ale rejestr informuje klienty, w którym porcie działa serwer. Rejestr zwykle działa w dobrze znanym porcie.

W Javie 1.2 dodano nową odmianę zdalnego serwera, klasę `java.rmi.activation.Activatable`:

```
public abstract class Activatable extends RemoteServer //Java 1.2
```

Obiekt `UnicastRemoteObject` istnieje tylko dopóty, dopóki działa serwer, który go utworzył. Kiedy serwer kończy pracę, obiekt znika na zawsze. Obiekty `Activatable` pozwalają klientom na ponowne połączenie się z serwerem po zamknięciu i ponownym uruchomieniu serwera i uzyskanie dostępu do tych samych zdalnych obiektów. Klasa ta zawiera także statyczne metody `Activatable.exportObject()`, które możesz wywołać, jeśli nie chcesz wywodzić swojej klasy od klasy `Activatable`.”

Przykład 18-2. „klasa `FibonacciImpl`, implementuje zdalny interfejs `Fibonacci`.

Klasa ta zawiera konstruktor i dwie metody `getFibonacci ()`. Klient będzie miał dostęp tylko do metod `getFibonacci ()`, ponieważ tylko one są zdefiniowane w interfejsie `Fibonacci`. Konstruktor jest używany po stronie serwera, ale nie jest dostępny dla klienta.

Przykład 18-2. Klasa `FibonacciImpl`

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject; import
java.math.BigInteger;

public class FibonacciImpl extends UnicastRemoteObject
    implements Fibonacci {
    public FibonacciImpl() throws RemoteException {
        UnicastRemoteObject.exportObject(this);
    }
    public BigInteger getFibonacci(int n)
        throws RemoteException {
        return this.getFibonacci(new BigInteger(Long.toString(n)));
    }
    public BigInteger getFibonacci(BigInteger n)
        throws RemoteException {
        System.out.println("Wyznaczam" + n + ". liczbę Fibonacciego");
        BigInteger zero = new BigInteger("0");
        BigInteger one = new BigInteger("1");
        if (n.equals(zero)) return zero;
        if (n.equals(one)) return one;
        BigInteger i = one;
        BigInteger a = zero;
        BigInteger b = one;
        while (i.compareTo(n)==-1)
            BigInteger temp = b;
            b = b.add(a);
            a = temp;
            i = i.add(one);
        }
    return b;
}
```

Konstruktor `FibonacciImpl ()` eksportuje obiekt, to znaczy tworzy w pewnym porcie obiekt `UnicastRemoteObject`, który zaczyna czekać na połączenia. Deklarujemy, że konstruktor zgłasza wyjątek `RemoteException`, ponieważ metoda `UnicastRemoteObject.exportObject ()` może zgłosić ten wyjątek.

Metoda `getFibonacci (int n)` jest bardzo prosta. Zwraca po prostu wynik przekształcenia swojego argumentu w `BigInteger` i wywołania drugiej metody `getFibonacci ()`. Druga metoda przeprowadza właściwe obliczenia. W obliczeniach wykorzystuje obiekty `BigInteger`, aby umożliwić wyznaczenie dowolnie dużej liczby Fibonacciego o dowolnie dużym indeksie. Do wykonania tych obliczeń potrzeba sporo mocy obliczeniowej i pamięci. Właśnie dlatego warto umieścić tę metodę w wyspecjalizowanym serwerze obliczeniowym, zamiast wywoływać ją lokalnie.

Choć `getFibonacci ()` jest metodą zdalną, nie różni się ona niczym od metod lokalnych. Jest to prosty przykład, ale nawet bardzo skomplikowane metody zdalne nie różnią się algorytmicznie od swoich lokalnych odpowiedników. Jedyne różnica - że metoda zdalna

jest deklarowana w zdalnym interfejsie, a metoda lokalna nie - nie jest odzwierciedlona w samej metodzie.”

„Teraz musimy napisać serwer, który udostępni zdalny obiekt **Fibonacci** reszcie świata. Przykład 18-3 to właśnie taki serwer. Ma on tylko metodę `main()`. Zaczyna działanie od wejścia w blok `try`, który przechwytuje wyjątek `RemoteException`. Następnie konstruuje nowy obiekt `FibonacciImpl` i wiąże ten obiekt z nazwą `fibonacci`, używając klasy `Naming` do porozumienia się z lokalnym rejestrem. Rejestr przechowuje obiekty dostępne w serwerze RMI oraz nazwy, za pomocą których można uzyskać do nich dostęp. Kiedy tworzony jest nowy zdalny obiekt, dodaje on siebie i swoją nazwę do rejestru za pomocą metod `Naming.bind()` i `Naming.rebind()`. Klienci mogą wówczas pobrać ten obiekt według nazwy albo uzyskać listę wszystkich dostępnych obiektów. Zauważ, że nazwa obiektu w rejestrze wcale nie musi mieć związku z nazwą klasy. Moglibyśmy nazwać ten obiekt "Fred". W rejestrze może występować zresztą wiele egzemplarzy tej samej klasy, każdy pod inną nazwą. Po zarejestrowaniu się serwer wyświetla komunikat na `System.out`, informując, że jest gotów do przyjmowania zdalnych wywołań. Jeśli coś pójdzie źle, blok `catch` wyświetla krótki komunikat o błędzie.”

Przykład 18.3. Klasa `FibonacciServer`

```
import java.net.*;
import java.rmi.*;

public class FibonacciServer {

    public static void main(String[] args) {
        try {
            FibonacciImpl f = new FibonacciImpl();
            Naming.rebind("fibonacci", f); System.out.println("Serwer
            liczb Fibonacciego gotowy.");
        } catch (RemoteException re) {
            System.out.println("Wyjątek w FibonacciServer: " + re);
        } catch (MalformedURLException e) {
            System.out.println("Wyjątek MalformedURLException: " +
            e);
        }
    }
}
```

„Choć metoda `main()` w tym przykładzie dość szybko kończy pracę, serwer będzie nadal działał, ponieważ podczas eksportowania obiektu `FibonacciImpl` przez konstruktor `FibonacciImpl()` tworzony jest nowy wątek. Cały kod serwera jest już gotowy.”

Strona klienta

„Zanim klient będzie mógł wywołać zdalną metodę, musi pobrać zdalną referencję do zdalnego obiektu. Uzyskuje ją z rejestru w serwerze. Konwencja nazewnicza zależy od używanego rejestru; klasa `java.rmi.Naming` udostępnia metodę lokalizowania obiektów opartą na adresach URL. Jak widać w poniższym kodzie, adresy te zaprojektowano tak, aby przypominały adresy URL typu *http*. Protokół to *rmi*. Pole pliku w adresie URL określa nazwę zdalnego obiektu. Pola nazwy hosta i numeru portu pozostają bez zmian:

```
Object o1 = Naming.lookup("rmi://metalab.unc.edu/fibonacci");  
Object o2 =  
    Naming.lookup("rmi://metalab.unc.edu:2048/fibonacci");
```

Podobnie jak obiekty w tablicach mieszania, wektorach i innych strukturach danych przechowujących obiekty różnych klas, obiekt pobrany z rejestru traci informacje o typie. Zanim więc użyjesz obiektu, musisz rzutować go na zdalny interfejs implementowany przez ten obiekt (nie na rzeczywistą klasę, która jest ukryta przed klientami):

```
Fibonacci calculator = (Fibonacci) Naming.lookup("fibonacci");
```

Kiedy już pobierzesz referencję do obiektu i odtworzysz jego typ, klient może wywoływać metody zdalnego obiektu niemal tak samo, jak metody obiektu lokalnego. Jedyna różnica polega na tym, że musisz przechwytywać wyjątek `RemoteException` podczas każdego zdalnego wywołania:

```
try (  
    BigInteger f56 = calculator.getFibonacci(56);  
    System.out.println("56. liczba Fibonacciego to " + f56);  
    BigInteger f156 = calculator.getFibonacci(56);  
    System.out.println("156. liczba Fibonacciego to " + f156);  
}  
catch (RemoteException e) (  
    System.err.println(e)  
)
```

Przykład 18-4 to prosty klient dla interfejsu `Fibonacci` z poprzedniego podrozdziału.

Przykład 18-4. `FibonacciClient`

```
import java.rmi.*;
import java.net.*;
import java.math.BigInteger;

public class FibonacciClient {

    public static void main (String args[]) {
        if (args.length == 0 || !args[0].startsWith("rmi:")) {
            System.err.println(
                "Użycie: java FibonacciClient
                 rmi://host.domena:port/liczba Fibonacci");
            return;
        }

        try {
            Object o = Naming.lookup(args[0]);
            Fibonacci calculator = (Fibonacci) o;
            for (int i = 1; i < args.length; i++) {
                try {
                    BigInteger index = new BigInteger(args[i]); BigInteger f =
                        calculator.getFibonacci(index);
                    System.out.println(args[i] + "liczba Fibonacci to " + f);
                }
                catch (NumberFormatException e) {
                    System.err.println(args[i] + "nie jest liczbą
                                                                całkowitą.");
                }
            }
        }
        catch (MalformedURLException e) {
            System.err.println(args[0] + " jest błędnym adresem URL
            RMI");
        }
        catch (RemoteException e) {
            System.err.println("Zdalny obiekt zgłosił wyjątek" + e);
        }
        catch (NotBoundException e) {
            System.err.println(
                "Nie znalazłem w serwerze iadanego zdalnego obiektu");
        }
    }
}
```

Skompiluj tę klasę jak zwykle. Zauważ, że ponieważ obiekt zwracany przez metodę `Naming.lookup()` jest zwracany na typ `Fibonacci`, plik *Fibonacci.java* albo *Fibonacci.class* musi znajdować się w twoim komputerze. Ogólnie rzecz biorąc, w celu skompilowania klienta trzeba dysponować albo kodem źródłowym, albo kodem bajtowym zdalnego obiektu, z którym będziesz się łączył. „

Kompilowanie trzonów

„Zanim serwer zacznie przyjmować wywołania, musimy wygenerować trzony i szkielety niezbędne do pracy programu. Powiedzieliśmy już, do czego służą trzony i szkielety: trzon zawiera informacje z interfejsu `Remote` (w tym przykładzie obiekt z dwoma metodami `getFibonacci()`), a szkielet jest podobny, ale znajduje się po stronie serwera. Na szczęście, nie musisz pisać ich sam. Możesz wygenerować je automatycznie z kodu źródłowego zdalnych obiektów Javy za pomocą programu narzędziowego o nazwie *rmic* dołączonego do IOK. Aby wygenerować trzony i szkielety ze zdalnego obiektu `FibonacciImpl`, uruchom *rmic* z nazwą klasy zdalnego obiektu.

Na przykład:

```
% rmic FibonacciImpl
% ls Fibonacci*

Fibonacci.class
Fibonacci.java
FibonacciClient.class
FibonacciClient.java

FibonacciImpl.class
FibonacciImpl.java

FibonacciServer.class
FibonacciServer.java

FibonacciImpl_Skel.class
FibonacciImpl_Stub.class
```

Program *rmic* odczytuje pliki *.class* zdalnego obiektu i tworzy pliki *.class* trzonów i szkieletów niezbędnych do działania zdalnego obiektu. Argument linii polecenia *rmic* to pełna, pakietowa nazwa klasy zdalnego obiektu (na przykład *com. macfaq. rmi. examples. Chat*, a nie po prostu *Chat*).”

„Program *rmic* obsługuje te same opcje linii polecenia co kompilator *java*, na przykład *-classpath i -d*.

Jeśli na przykład klasa nie znajduje się na ścieżce klas, możesz określić jej położenie za pomocą argumentu *- class spa th*.

Poniższe polecenie szuka klasy *Fibonaccimpl.class* w katalogu *test/klasy*:

```
% rmic -classpath test/klasy Fibonaccimpl
```

Teraz skopiuj pliki

```
Fibonacci.class i  
Fibonaccimpl-Stub.class
```

do katalogu w zdalnym serwerze, z którego zostanie uruchomiony obiekt *Fibonacci* oraz do katalogu w lokalnym kliencie, z którego obiekt *Fibonacci* będzie wywoływany.

Uruchamianie serwera

Teraz jesteś gotów do uruchomienia serwera. W rzeczywistości będziesz musiał uruchomić dwa serwery: zdalny obiekt (w tym przykładzie *FibonacciServer*) oraz rejestr, który pozwala klientom na łączenie się ze zdalnym serwerem. Ponieważ serwer będzie próbował porozumieć się z rejestrem Naming, najpierw musisz uruchomić rejestr. Sprawdź, czy wszystkie klasy trzonu, szkieletu i serwera znajdują się na ścieżce klas serwera i wpisz:

```
% rmiregistry &
```

W Windows uruchom rejestr z linii poleceń DOS-a w następujący sposób:

```
c:> start rmiregistry
```

W obu powyższych przykładach rejestr zostanie uruchomiony w tle. Rejestr domyślnie próbuje monitorować port 1099. Jeśli to się nie uda (zwłaszcza gdy zostanie wyświetlony komunikat w rodzaju "java.net.SocketException: Address already in use"), oznacza to, że inny program używa portu 1099 - prawdopodobnie (choć niekoniecznie) inna usługa rejestrująca. Możesz uruchomić rejestr w innym porcie, dołączając do polecenia numer portu:

```
% rmiregistry 2048 &
```

Jeśli używasz innego portu, musisz dołączać jego numer do adresów URL odwołujących się do tego rejestru.

Wreszcie możesz uruchomić serwer. Uruchom go tak, jak każdą inną klasę Javy zawierającą metodę *main ()*:

```
% java FibonacciServer  
Serwer liczb Fibonacciego gotowy.
```

Serwer i rejestr są teraz gotowe do przyjmowania zdalnych wywołań.”

Uruchamianie klienta

„Przejdź do systemu klienckiego. Sprawdź, czy pliki *FibonacciClient.class*, *Fibonacci.class* i *Fibonacci_Impl.class* znajdują się na ścieżce klas. [Wpisz polecenie:](#)

```
C:>java FibonacciClient rmi://host.com/fibonacci 0 1 2 3 4 5
                                                56 156
0. liczba Fibonacciego to 0
1. liczba Fibonacciego to 1
2. liczba Fibonacciego to 1
3. liczba Fibonacciego to 2
4. liczba Fibonacciego to 3
5. liczba Fibonacciego to 5
56. liczba Fibonacciego to 225851433717
156. liczba Fibonacciego to
    178890334785183168257455287891792
```

Klient przekształca argumenty linii polecenia w obiekty `BigInteger`. Wysyła je przez sieć do zdalnego serwera. Serwer otrzymuje obiekty, wyznacza liczbę Fibonacciego o zadanym indeksie i odsyła obiekt `BigInteger` do klienta. W tym przykładzie użyłem komputera PC w charakterze klienta i zdalnego serwera uniksowego; można uruchomić oba programy w tym samym komputerze, choć jest to mniej interesujące.”

Ładowanie klas w czasie wykonania

Klient musi znać tylko interfejs zdalnego obiektu. Wszystko inne - na przykład klasy trzonowe - można załadować z serwera WWW (ale nie z serwera RMI) w czasie wykonania, wykorzystując program ładujący klasy. Możliwość ładowania klas z sieci jest zresztą jedną z unikalnych cech Javy. Przydaje się to zwłaszcza w apletach. Serwer WWW może wysłać przeglądarce aplet, który będzie porozumiewał się serwerem, aby na przykład pozwolić klientowi na odczytywanie i zapisywanie plików w serwerze. Jak zawsze w przypadku ładowania klas z niezaufanego hosta, musi je sprawdzić `SecurityManager`.

Niestety, choć łatwo pracować ze zdalnymi obiektami, kiedy możesz zainstalować niezbędne trzony i szkielety na ścieżce klas klienta, dynamiczne ładowanie trzonów i innych klas jest niewiarygodnie trudne. Choć Sun potrafi projektować dobre interfejsy programistyczne, to interfejsy użytkownika pozostawiają wiele do życzenia. Ścieżka klas, architektura zabezpieczeń i konieczność posługiwania się słabo udokumentowanymi zmiennymi środowiskowymi to zmory dręczące programistów Javy. Zmuszenie lokalnego klienta do pobrania zdalnych obiektów z serwera wymaga precyzyjnej manipulacji wszystkimi tymi mechanizmami. Nawet drobna pomyłka uniemożliwia uruchomienie programu, a gdy nieszczęsny programista chce sprawdzić, gdzie popełnił błąd, ma do dyspozycji tylko najogólniejsze wyjątki. Stopień trudności uruchomienia programów zależy od kontekstu, w którym działają zdalne obiekty. Ogólnie rzecz biorąc, nieco łatwiej zarządzać apletami używającymi RMI niż samodzielnymi aplikacjami. Napisanie samodzielnej aplikacji jest wykonalne wtedy, gdy klient ma dostęp do tych samych plików `.class` co serwer. Napisanie aplikacji, która musi ładować klasy z serwera, jest prawie niemożliwe.