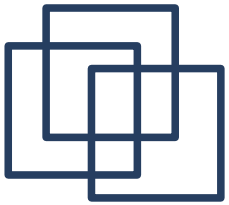
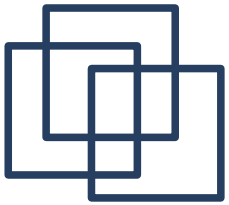


Remote Method Invocation

1. Interfejsy w Javie.
2. Zdalne wywołanie metod (*Remote Method Invocation*).

• interfejsy w RMI,

3. Przykładowa aplikacja korzystająca z RMI,
4. Własne gniazda w RMI.
5. Dystrybucja programu rozproszonego,
6. Przekazywanie parametrów.



Interfejsy w Javie

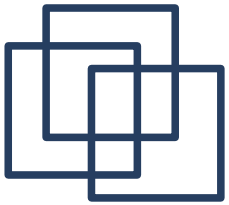
Interfejs (*Interface*) jest zbiorem metod, które mogą być implementowane przez dowolną klasę. Interfejs deklaruje metody natomiast nie zawiera ich implementacji. Klasa implementująca interfejs musi zawierać implementację każdej z metod wchodzących w skład interfejsu.

Podstawowe różnice pomiędzy interfejsem i klasą abstrakcyjną w Javie:

• interfejs nie może implementować żadnej z metod, klasa abstrakcyjna może zawierać implementację.

• dowolna klasa może posiadać maksymalnie jedną nadklasę oraz implementować dowolnie wiele interfejsów.

• interfejsy nie są związane hierarchią klas – dowolny zbiór klas może implementować ten sam interfejs.

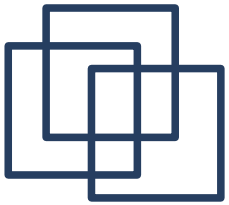


Interfejsy w Javie

Przykład zastosowania interfejsów:

```
public class Product{
    protected String name;
    protected float price;
    protected int stackSize;

    public void setName(String s){
        this.name = s;
    }
    public String getName(void){
        return this.name;
    }
    ...
}
```

Obiekty będące instancjami klasy **`Product`** mogą zawierać informacje o dowolnych rodzajach produktów. W niektórych sytuacjach korzystne może być operowanie na posortowanych zbiorach takich obiektów.



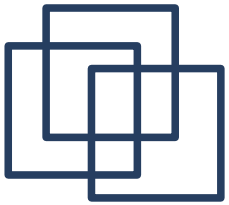
Interfejsy w Javie

Do rozwiązania zadania można użyć interfejsów np:

```
public interface Comparable{
   public int compareTo(Object o);
}
```

Interfejs **Comparable** zawiera jedną metodę, której argumentem jest dowolny obiekt. Metoda zwraca liczbę całkowitą. Umawiamy się, że:

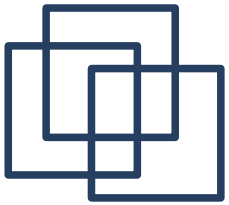
x < y jeśli **x.compareTo(y) < 0**,
x = y jeśli **x.compareTo(y) = 0**,
x > y jeśli **x.compareTo(y) > 0**,



Interfejsy w Javie

W kolejnym kroku piszemy ogólną metodę sortującą:

```
public class Sorter{

    public static sort(Comparable[] cArray){
        int i, j;
        Object tmp;

        for(i=0; i<cArray.length(); i++){
            for(j=1; j<cArray.length()-i; j++){
                if (cArray[j-1].compareTo(cArray[j])>0)
                    tmp = cArray[i];
                    cArray[i] = cArray[j];
                    cArray[j] = tmp;
            } //for
        } //for
    }// sort
}// class
```



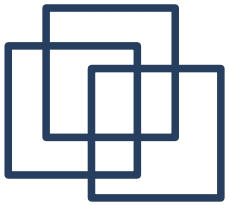
Interfejsy w Javie

Aby wykorzystać **Sorter** dla produktów wprowadzamy nową klasę:

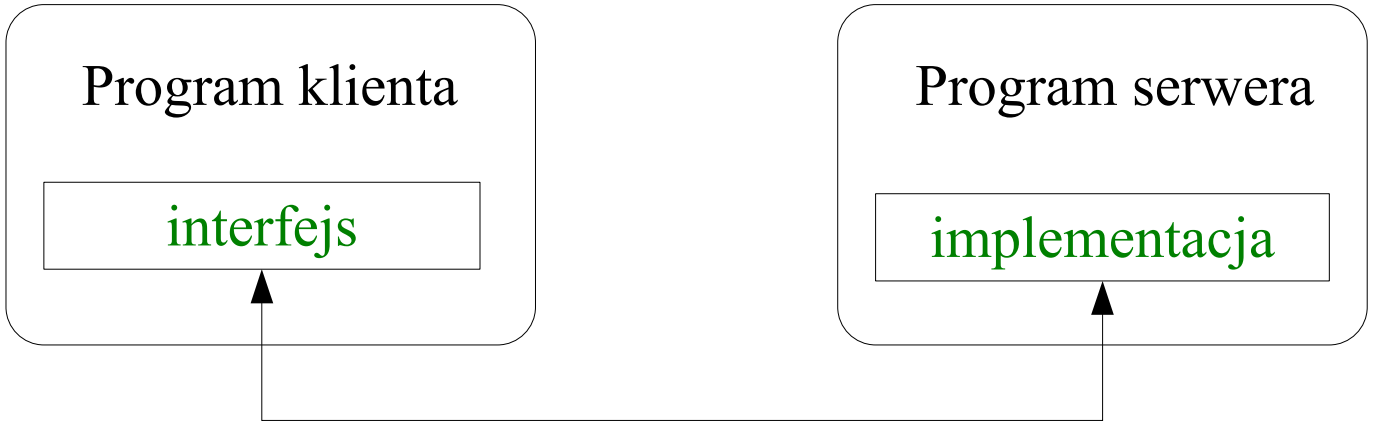
```
public class PriceSortedProduct extends Product implements Comparable{
    ...
    int compareTo(Object obj){
        Product p = (Product)obj;
        if (this.price < p.price) return -1;
        else if (this.price > p.price) return 1;
        else return 0;
    }
    ...
}
```

W programie wykorzystującym sortowanie produktów powinny znaleźć się linie:

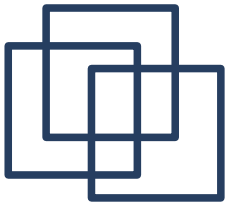
```
PriceSortedProduct[] pArray;
...
Sorter.sort(pArray);
...
```



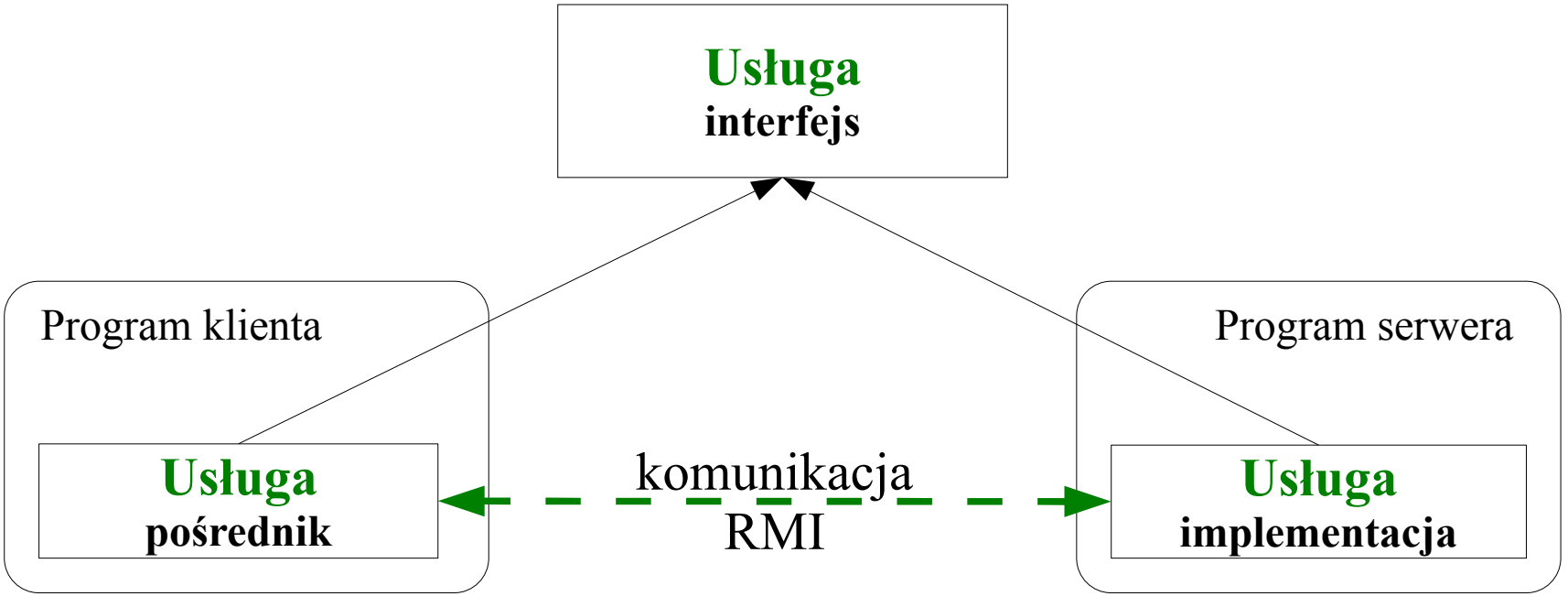
Interfejsy w RMI



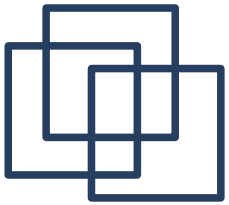
Architektura RMI opiera się na zasadzie rozdziału kodu zawierającego definicję (specyfikację) oraz implementację określonej funkcjonalności. Definicja jest zapisana pod postacią interfejsu. Implementacja jest wyrażona w ramach klasy. Interfejsy w Javie nie zawierają kodu wykonywalnego.



Interfejsy w RMI



RMI wykorzystuje dwie klasy implementujące wspólny interfejs. Pierwsza z nich zawiera właściwy kod wykonywalny i jest umieszczona na serwerze. Druga z klas jest pośrednikiem pomiędzy klientem i zdalną usługą na serwerze i działa w ramach programu klienta.

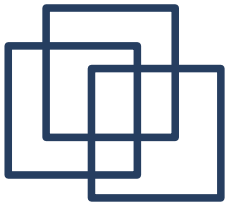


Elementy RMI

Wyszukiwanie zdalnych obiektów może odbywać się za pośrednictwem różnych usług. Najpopularniejsze to JNDI (*Java Naming and Directory Interface*) oraz RMI Registry.

Udostępnienie przez serwer usługi RMI za pomocą RMI Registry przebiega w kilku krokach:

- stworzenie lokalnego obiektu implementującego usługę,
- wyeksportowanie obiektu do systemu RMI,
- utworzenie usługi oczekującej na zgłoszenia klientów,
- zarejestrowanie nazwy usługi w RMI Registry.



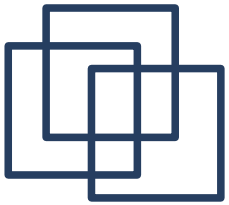
Elementy RMI

Do zarejestrowania usługi można użyć klasy **`java.rmi.Naming`**. Klasa ta zawiera metodę **`bind()`**:

```
public static void bind(String name, Remote obj) throws
AlreadyBoundException, MalformedURLException, RemoteException
```

`name` – ciąg w formacie:
`rmi://nazwa_serwera[:numer_portu]/nazwa_usługi`
`nazwa_serwera` – adres IP lub nazwa zarejestrowana w DNS,
`numer_portu` – domyślnie 1099,
`nazwa_usługi` – nazwa pod jaką zarejestrowano usługę.
`obj` – rejestrowany obiekt.

W przypadku błędu zwracany jest jeden z wyjątków:
`java.rmi.AlreadyBoundException` – wykorzystywana nazwa,
`java.rmi.RemoteException` – nie można się skontaktować z serwerem,
`java.net.MalformedURLException` – niewłaściwy adres URL.
`java.rmi.AccessException` – operacja niedozwolona,



Elementy RMI

Klient do wyszukania usługi poprzez RMI Registry używa klasy **`java.rmi.Naming`** i metody **`lookup()`**:

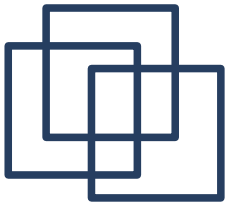
```
public static Remote lookup(String name) throws
NotBoundException, MalformedURLException, RemoteException
```

`name` – URL w formacie analogicznym jak w metodzie **`bind()`**:

`rmi://nazwa_serwera[:numer_portu]/nazwa_usługi`

Metoda zwraca referencję do obiektu **`java.rmi.Remote`**. W przypadku błędu zwracany jest jeden z wyjątków:

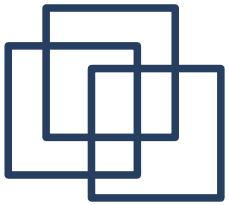
`java.rmi.NotBoundException` – nieznana nazwa,
`java.rmi.RemoteException` – nie można się skontaktować z serwerem,
`java.rmi.AccessException` – operacja niedozwolona,
`java.net.MalformedURLException` – niewłaściwy adres URL.



Przykład – aplikacja korzystająca z RMI

W celu stworzenia aplikacji korzystającej z RMI należy:

1. Napisać i skompilować kod opisujący interfejsy.
2. Napisać i skompilować kod klas implementujących zaprojektowane interfejsy.
3. Wygenerować za pomocą narzędzia **`rmic`** pliki klas **`Stub`** i **`Skeleton`** dla klas implementujących.
4. Napisać kod programu realizującego usługę.
5. Napisać kod klienta RMI.
6. Zainstalować i uruchomić system RMI (RMI Registry).



Krok 1: interfejsy

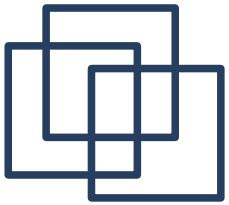
Program umożliwia przeprowadzenie podstawowych działań na liczbach całkowitych. Operacje te będą wykonywane z wykorzystaniem RMI - klient wysyła argumenty działania do serwera i odbiera wynik.

Plik **`Calculator.java`**

```
public interface Calculator extends java.rmi.Remote {
    public long add(long a, long b) throws java.rmi.RemoteException;
    public long sub(long a, long b) throws java.rmi.RemoteException;
    public long mul(long a, long b) throws java.rmi.RemoteException;
    public long div(long a, long b) throws java.rmi.RemoteException;
}
```

Interfejs musi rozszerzać **`java.rmi.Remote`**. Ponieważ każda zdalna metoda może potencjalnie zwrócić wyjątek **`java.rmi.RemoteException`**, więc musi on być zadeklarowany.

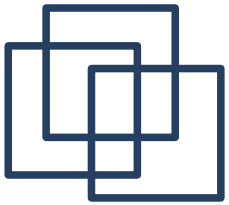
Kompilacja: **`javac Calculator.java`**



Krok 2: implementacja

Plik **`CalculatorImpl.java`**

```
public class CalculatorImpl extends java.rmi.server.UnicastRemoteObject
                                implements Calculator {


    // Implementacja musi zawierać konstruktor ponieważ utworzenie obiektu
    // może zwrócić wyjątek java.rmi.RemoteException
    public CalculatorImpl() throws java.rmi.RemoteException {
        super();
    }
    public long add(long a, long b) throws java.rmi.RemoteException {
        return a + b;
    }
    public long sub(long a, long b) throws java.rmi.RemoteException {
        return a - b;
    }
    public long mul(long a, long b) throws java.rmi.RemoteException {
        return a * b;
    }
    public long div(long a, long b) throws java.rmi.RemoteException {
        return a / b;
    }
}
```

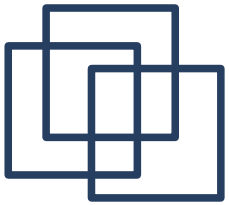


Krok 3: implementacja

Klasa **`CalculatorImpl`** używa **`java.rmi.server.UnicastRemoteObject`**, który definiuje niereplikowalny zdalny obiekt, którego referencje są ważne jedynie w trakcie działania usługi. Klasa **`UnicastRemoteObject`** umożliwia połączenia typu punt–punkt z wykorzystaniem transmisji strumieniowej poprzez TCP.

Rozszerzenie klasy **`UnicastRemoteObject`** nie jest obowiązkowe. Zdalne obiekty powinny jedynie rozszerzać **`java.rmi.server.RemoteObject`**. Wtedy dodatkową część kodu realizowaną przez **`UnicastRemoteObject`** należy zaimplementować własnoręcznie.

Kompilacja: **`javac CalculatorImpl.java`**



Krok 3: stub i skeleton

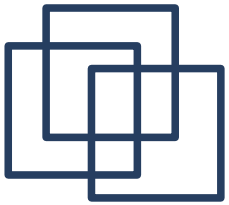
Używając klasy **`CalculatorImpl`** należy wygenerować automatycznie klasy **`Stub`** i **`Skeleton`**. W tym celu korzysta się z polecenia:

`rmic [opcje] klasa` np.

```
rmic CalculatorImpl
```

W wyniku jego działania w bieżącym katalogu pojawią się dwa nowe pliki:

`Calculator_Stub.class`
`Calculator_Skel.class` (od wersji 1.2)

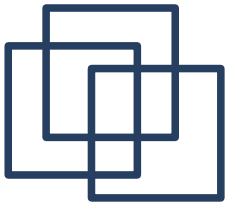


Krok 4: kod serwera

Program serwera tworzy zdalny obiekt i rejestruje go w systemie RMI.

Plik **CalculatorServer.java**

```
import java.rmi.Naming;

public class CalculatorServer {

  public CalculatorServer() {
    try {
      Calculator c = new CalculatorImpl();
      Naming.rebind("rmi://localhost:1099/CalculatorService", c);
    } catch (Exception e) {
      System.out.println("Problem: " + e);
    }
  }

  public static void main(String args[]) {
    new CalculatorServer();
  }
}
```

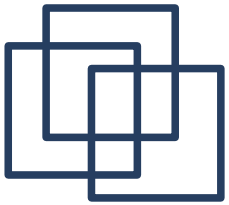


Krok 5: kod klienta

Przykładowy program kliencki mógłby wyglądać następująco.

Plik **`CalculatorClient.java`**

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Calculator c = (Calculator)
                Naming.lookup("rmi://remotehost/CalculatorService");
            System.out.println( c.sub(4, 3) );
            System.out.println( c.add(4, 5) );
            System.out.println( c.mul(3, 6) )
            System.out.println( c.div(9, 3) );
        }catch (Exception e) {System.out.println("Problem:" + e);}
    }
}
```



Krok 6: uruchomienie

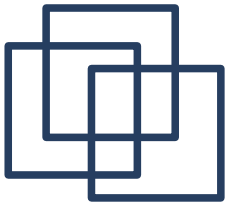
Kompilacja klienta i serwera:

```
javac CalculatorClient.java
javac CalculatorServer.java
```

Uruchomienie:

```
rmiregistry
java CalculatorServer

java CalculatorClient
```

Pomimo, że program jest uruchomiony na jednym komputerze, to do komunikacji między trzema Wirtualnymi Maszynami Javy wykorzystywany jest protokół TCP/IP.



Szyfrowanie w RMI

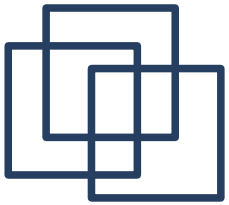
Technologia RMI została zaprojektowana tak, aby można było jej używać z dowolnymi mechanizmami zapewniającymi transport danych przez sieć, działającymi ponad protokołem TCP. W praktyce odbywa się to poprzez implementację własnego obiektu (tzw. s*ocket factory*) dostarczającego gniazda wykorzystywane do komunikacji RMI.

Implementacja własnego s*ocket factory* składa się z trzech kroków:

1. Implementacja własnych wersji klas **`ServerSocket`** i **`Socket`**.

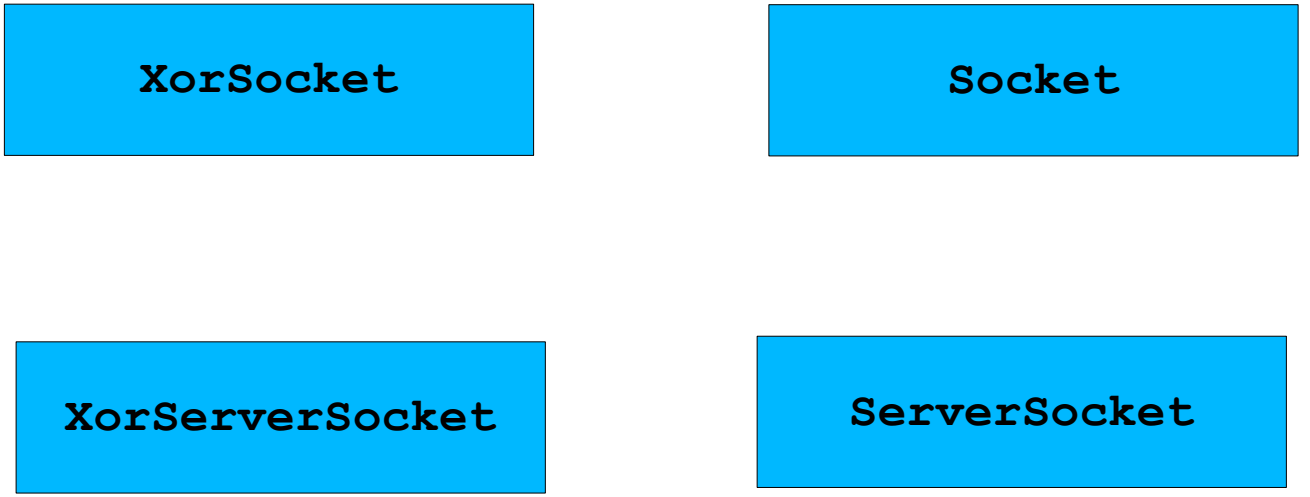
2. Implementacja własnej klasy **`ClientSocketFactory`**.

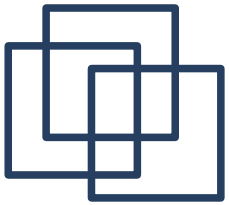
3. Implementacja własnej klasy **`ServerSocketFactory`**.



Implementacja własnych gniazd

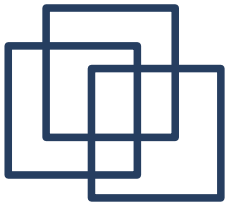
Przygotujemy gniazda, które będą umożliwiały przesyłanie przez siec danych kodowanych za pomocą operacji XOR i ustalonego, ośmiobitowego wzorca.





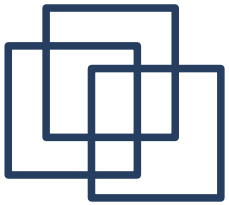
Implementacja XORSocket

```
import java.io.*;

import java.net.*;
class XorSocket extends Socket {
    private final byte pattern; // wzorzec kodowania
    private InputStream in = null;
    private OutputStream out = null;

    public XorSocket(byte pattern) throws IOException {
        super();
        this.pattern = pattern;
     }

    public XorSocket(String host, int port, byte pattern) throws IOException {
         super(host, port);
         this.pattern = pattern;
     }
```



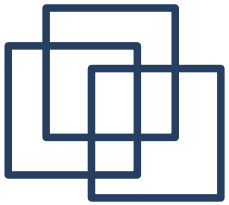
Implementacja XORSocket

```
    public synchronized InputStream getInputStream()throws IOException {
        if (in == null) {
            in = new XorInputStream(super.getInputStream(), pattern);
        }
        return in;
    }
    public synchronized OutputStream getOutputStream()throws IOException {
        if (out == null) {
            out = new XorOutputStream(super.getOutputStream(),pattern);
        }
        return out;
    }
}
```



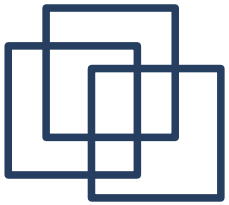
Implementacja XORServerSocket

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

class XorServerSocket extends ServerSocket {
    private final byte pattern;
    public XorServerSocket(int port, byte pattern)
                                                    throws IOException {
        super(port);
        this.pattern = pattern;
    }

    public Socket accept() throws IOException {
        Socket s = new XorSocket(pattern);
        this.implAccept(s);
        return s;
    }
}
```



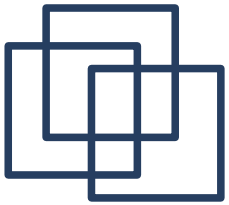
Implementacja strumieni

```
import java.io.*;

class XorOutputStream extends FilterOutputStream {
    private final byte pattern;

    public XorOutputStream(OutputStream out, byte pattern) {
        super(out);
        this.pattern = pattern;
    }

    public void write(int b) throws IOException {
        out.write((b ^ pattern) & 0xFF);
    }
}
```



Implementacja strumieni

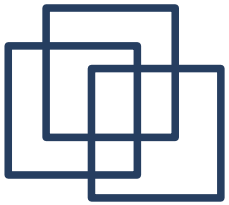
```
import java.io.*;


class XorInputStream extends FilterInputStream {
    private final byte pattern;

    public XorInputStream(InputStream in, byte pattern) {
        super(in);
        this.pattern = pattern;
     }

    public int read() throws IOException {
        int b = in.read();
        if (b != -1)  b = (b ^ pattern) & 0xFF;
        return b;
    }

    public int read(byte b[], int off, int len) throws IOException {
        int n = in.read(b, off, len);
        if (n <= 0) return n;
        for(int i = 0; i < n; i++)
            b[off + i] = (byte)((b[off + i] ^ pattern) & 0xFF);
        return n;
    }
}
```



Implementacja socket factories

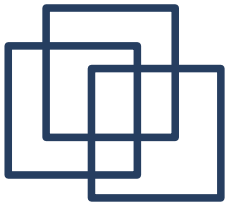
W technologi RMI serwer (zdalny obiekt) decyduje o rodzaju transmisji. Dzięki temu kod programu klienckiego jest niezależny od zmian (np. szyfrowanie) w protokole komunikacji. Z drugiej strony wszelkie dane potrzebne do zainicjowania połączenia są do klienta przesyłane przez sieć – czyli muszą być serializowalne.

Aby utworzyć gniazdo służące do komunikacji RMI korzysta z interfejsów **SocketFactory** oraz **ServerSocketFactory**, które udostępniają odpowiednio metody **createSocket(String host, int port)** i **createServerSocket(int port)**.



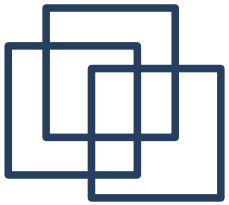
Implementacja RMIClientSocketFactory

```
import java.io.*;

import java.net.*;
import java.rmi.server.*;

public class XorClientSocketFactory
                        implements RMIClientSocketFactory, Serializable {
    private byte pattern;

    public XorClientSocketFactory(byte pattern) {
        this.pattern = pattern;
    }

    public Socket createSocket(String host, int port) throws IOException {
        return new XorSocket(host, port, pattern);
    }

    public int hashCode() { return (int) pattern; }
    public boolean equals(Object obj) {
        return (getClass() == obj.getClass() &&
                pattern == ((XorClientSocketFactory) obj).pattern);
    }
}
```



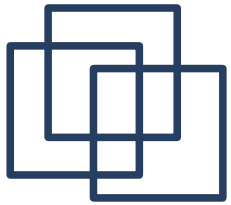
Implementacja RMIServerSocketFactory

```
import java.io.*;

import java.net.*;
import java.rmi.server.*;

public class XorServerSocketFactory implements RMIServerSocketFactory{
    private byte pattern;

    public XorServerSocketFactory(byte pattern) {
        this.pattern = pattern;
    }

    public ServerSocket createServerSocket(int port) throws IOException{
        return new XorServerSocket(port, pattern);
    }

    public int hashCode() { return (int) pattern; }
    public boolean equals(Object obj) {
        return (getClass() == obj.getClass() &&
                pattern == ((XorServerSocketFactory) obj).pattern);
    }
}
```



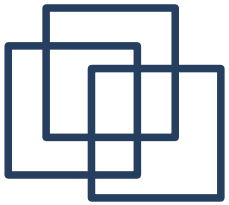
Program serwera

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloImpl implements Hello {

    public HelloImpl() {}

    public String getHello() {
        return "Hello World!";
    }

    public static void main(String args[]) {
        byte pattern = (byte) 0xC5; // 10100101
        try {
            HelloImpl obj = new HelloImpl();
            RMIClientSocketFactory csf =
                                    new XorClientSocketFactory(pattern);
            RMIServerSocketFactory ssf =
                                    new XorServerSocketFactory(pattern);
```



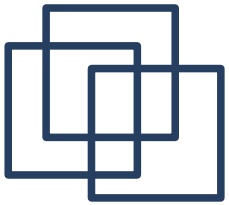
Program serwera

```
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj,
                                                                  0, csf, ssf);
            try {
                reg = LocateRegistry.getRegistry();
            } catch (RemoteException ex1) {
                try {
                    reg = LocateRegistry.createRegistry(1099);
                } catch (RemoteException ex2) {
                    return;
                }
            }
            reg.rebind("HelloService", stub);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Program klienta

```
import java.rmi.Naming;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloClient {

    public static void main(String args[]) {
        try {
            Hello obj = (Hello)
                        Naming.lookup("rmi://localhost/HelloService");
            System.out.println(obj.getHello());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Kompilacja i uruchomienie

1. Kompilacja:

```
javac *.java
```

2. Wygenerowanie klas łącznikowych:

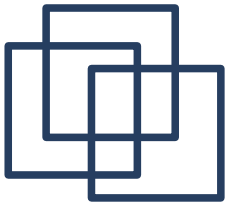
```
rmic HelloImpl
```

3. Uruchomienie programu serwera:

```
java HelloImpl
```

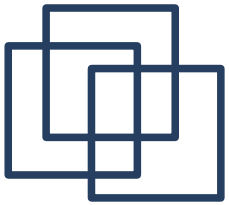
4. Uruchomienie programu klienta:

```
java HelloClient
```



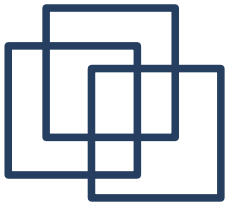
RMI i SSL

```
import java.io.IOException;

import java.io.Serializable;
import java.net.Socket;
import java.rmi.server.RMIClientSocketFactory;

import javax.net.ssl.SSLSocketFactory;

public class RMISSLClientSocketFactory
                    implements RMIClientSocketFactory, Serializable {

    public Socket createSocket(String arg0, int arg1)
                                                    throws IOException {

        SSLSocketFactory factory = (SSLSocketFactory)
                                            SSLSocketFactory.getDefault();
        return factory.createSocket(arg0, arg1);
    }
}
```



RMI i SSL

```
import java.io.IOException;

import java.io.Serializable;
import java.net.ServerSocket;
import java.rmi.server.RMIServerSocketFactory;

import javax.net.ssl.SSLServerSocketFactory;

public class RMISSLServerSocketFactory
                    implements RMIServerSocketFactory, Serializable{

    public ServerSocket createServerSocket(int arg0)
                                                throws IOException {

        SSLServerSocketFactory factory = (SSLServerSocketFactory)
                                    SSLServerSocketFactory.getDefault();
        return factory.createServerSocket(arg0);
    }
}
```

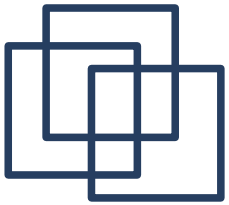


RMI i SSL

W programie serwera (**`HelloImpl.java`**) należy zmienić linie odpowiadające za tworzenie obiektów **`SocketFactory`** na:

```
RMIClientSocketFactory csf = new RMISSLClientSocketFactory();
RMIServerSocketFactory ssf = new RMISSLServerSocketFactory();
```

UWAGA: przy uruchomieniu programu korzystającego z SSL należy uwzględnić magazyn kluczy używanych do autoryzacji i szyfrowania.



Przekazywanie parametrów

Przekazywanie parametrów wywołania metod jak również wyników przez nie zwracanych odbywa się w sposób niezależny od platformy i architektury komunikujących się komputerów. Sposób przekazywania parametrów zależy od ich typu. Istnieją trzy rodzaje parametrów używanych w aplikacjach rozproszonych RMI .

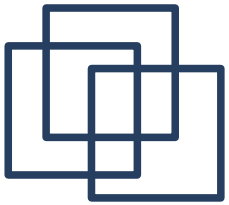
1. Proste typy danych (**`int`**, **`float`**, **`...`**) – przekazywane przez wartość.

2. Obiekty – przekazywane przez wartość, przed przesłaniem

wykonywana jest serializacja, po odebraniu deserializacja.

3. Zdalne obiekty

• w wyniku **`Naming.lookup()`** – przez referencję.



Przekazywanie parametrów

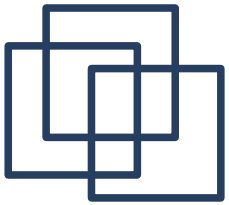
• w wyniku działania zdalnej metody – przykład:

```
// program klienta //
BankManager    bm;
Account        a;
try {
    bm = (BankManager)Naming.lookup("rmi://BankServer/BankManagerService");
    a  = bm.getAccount( "Jan Kowalski" );
    ...                    // kod używający obiektu Account
}
catch (RemoteException re) {
}
```

Metoda **`getAccount()`** działa po stronie serwera i zwraca lokalną referencję.

```
public Account getAccount(String accountName) {
    ...                    // wyszukanie odpowiedniego obiektu
    AccountImpl ai = ...   // utworzenie nowej referencji do tego obiektu
    return ai;             // zwrócenie referencji
}
```

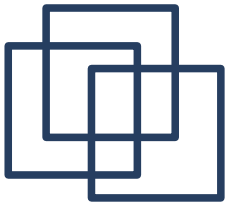
`ai` jest lokalną referencją do zdalnego obiektu.



Przekazywanie parametrów

Gdy metoda zwraca lokalną referencję do zdalnego obiektu RMI nie zwraca tego obiektu. Zamiast niego do zwracanego strumienia danych jest wstawiany inny obiekt (jego obiekt proxy).

Kiedy taki obiekt jest przesyłany między serwerem i klientem dwie Wirtualne Maszyny Javy używają swoich własnych kopii plików definiujących klasę tego obiektu. W związku z tym zmienne statyczne są aktualizowane niezależnie po stronie klienta i serwera – nie są przekazywane przez RMI.



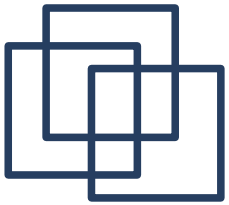
Dystrybucja programów wykorzystujących RMI

Pliki dostępne dla serwera:

- definicje zdalnych interfejsów (**`Calculator.class`**),
- implementacje zdalnych interfejsów (**`CalculatorImpl.class`**),
- klasy **`Skeletons`** dla klas implementujących (**`Calculator_Skel.class`**) do wersji 1.1,
- klasy **Stub** dla klas implementujących (**`Calculator_Stub.class`**),
- klasy programu serwera (**`CalculatorServer.class`**)

Pliki dostępne dla klienta

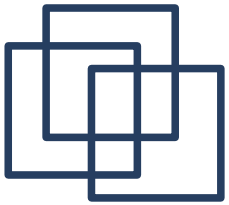
- definicje zdalnych interfejsów (**`Calculator.class`**),
- klasy **`Stub`** dla klas implementujących (**`Calculator_Stub.class`**),
- klasy obiektów zwracanych klientowi przez serwer ,
  klasy programu klienta (**`CalculatorClient.class`**),



Dystrybucja programów wykorzystujących RMI

Istnieje kilka podstawowych sposobów dystrybucji aplikacji rozproszonej:

1. *Closed* – wszystkie potrzebne pliki dla klienta i serwera są dostępne lokalnie.
2. *Server based* – program klienta wraz z potrzebnymi klasami jest w całości ładowany z serwera (applet)
3. *Client dynamic* – program klienta ładuje odpowiednie klasy z lokalizacji wskazanej przez serwer.
4. *Server dynamic* – program serwera ładuje odpowiednie klasy z lokalizacji wskazanej przez klienta.
5. *Bootstrap client* – cały kod klienta jest ładowany z sieci.
6. *Bootstrap server* – cały kod serwera jest ładowany z sieci.



Podsumowanie

Wywoływanie zdalnych metod w Javie jest technologią analogiczną do RPC w języku C. Odpowiednikiem zdalnego programu w RPC jest zarejestrowany, zdalny obiekt. Jego metody odpowiadają zdalnym procedurom w RPC.

RMI jest podstawą dla innych technologii (np. *Enterprise Java Beans*)wykorzystywanych do tworzenia współczesnych rozwiązań biznesowych opartych na języku Java.