Rozdział 8 Integracja Javy z innymi językami - JNI. Programowanie sieciowe

8.1 Integracja Javy z innymi językami - Java Native Interface (JNI)
8.1.1 Obsługa metod rodzimych w kodzie Javy
8.1.2 Kompilacja i generacja plików nagłówkowych
8.1.3 Implementacja metody rodzimej - funkcja a biblioteka
8.1.4 Dostęp do metod i pól zdefiniowanych w Javie
8.2 Programowanie sieciowe
8.2.1 Adresowanie komputerów w sieci (InetAddress i URL)
8.2.2 Komunikacja przez Internet (klient-serwer)
8.3 Serwlety
8.3.1 Model obsługi wiadomości
8.3.2 Środowisko wykonywania serwletów
8.3.3 Kontrola środowiska wymiany wiadomości
8.3.4 Metody wywoływania serwletów
8.3.5 Obsługa protokołu HTTP – pakiet javax.servlet.http.*
8.3.6 Bezpieczeństwo serwletów
8.4 Zdalne wywoływanie metod - RMI
8.4.1 Typy obiektów i relacje pomiędzy nimi w RMI
8.4.2 Komunikacja w procesie zdalnego wykonywania metod
8.4.3 Konstrukcja obiektu zdalnego – oprogramowanie serwera
8.4.4 Oprogramowanie klienta
8.4.5 Uruchamianie systemu.

8.1 Integracja Javy z innymi językami - Java Native Interface (JNI)

Tworząc programy w środowisku języka programowania Java napotyka się czasami na ograniczenia związane z dostępem do specyficznych dla danej platformy cech. Konieczne jest wówczas zastosowanie narzędzi obsługujących te cechy a następnie zaimplementowanie tych narządzi w kodzie programu tworzonego w Javie. Operacja taka jest możliwa poprzez wykorzystanie swoistego interfejsu pomiędzy kodem w Javie a kodem programu stworzonego w innym środowisku, np. C lub C++. Co więcej wykorzystanie istniejących już funkcji (napisanych wcześniej w innych językach programowania niż Java) może znacznie uprościć tworzenie nowej aplikacji w Javie, szczególnie wtedy gdy liczy się czas. Interfejs umożliwiający to specyficzne połączenie kodów został nazwany Java Native Interface, lub w skrócie JNI. Każda funkcja napisana w innym języku niż Java a implementowana bezpośrednio w kodzie Javy nosi nazwę metody rodzimej („native method”) i wymaga jawnej, sformalizowanej deklaracji. Metody rodzime mogą wykorzystywać obiekty Javy tak, jak to czynią metody tworzone w Javie, a więc tworzyć obiekty, używać je oraz modyfikować. Aby funkcjonalność metod rodzimych była pełna metody te mogą wywoływać metody tworzone w Javie, przekazywać im parametry i pobierać wartości lub referencje zwracane przez te metody. Możliwa jest również obsługa wyjątków metod rodzimych.
Czym jest więc JNI? JNI to interfejs zawierający:
- plik nagłówkowy środowiska rodzimego (np. plik jni.h dla środowiska C);
- generator pliku nagłówkowego metody rodzimej (np. javah -jni);
- formalizację deklaracji metody rodzimej,
- definicję i rzutowanie typów danych,

- zbiór metod (funkcji) umożliwiających wymianę danych i ustawianie stanów (np. wyjątków, monitora, itp.).

Implementacja JNI polega najprościej na wykonaniu następujących działań:
1. stworzenie programu w Javie zawierającego deklarację metody rodzimej (native);
2. kompilacja programu;
3. generacja pliku nagłówkowego środowiska rodzimego dla klasy stworzonego programu (javah -jni);
4. stworzenie implementacji metody rodzimej z wykorzystaniem plików nagłówkowych interfejsu JNI i klasy stworzonego programu ;
5. kompilacja metody rodzimej i umieszczenie jej w bibliotece;
6. uruchomienie programu korzystającego z metody rodzimej poprzez ładowanie biblioteki.

8.1.1 Obsługa metod rodzimych w kodzie Javy

Pierwszym krokiem w implementacji interfejsu JNI jest stworzenie kodu w Javie obsługującego metody rodzime. Najprostsza struktura obsługi może wyglądać następująco:

Przykład 8.1:

```
//Informacje.java:

class Informacje{
	//deklaracja metody rodzimej
	public native int infoSystemu(String parametr);

	//ładowanie biblioteki zawierającej implementację metody rodzimej
	static{
		System.loadLibrary(”sysinfo”);
	}

	//wykorzystanie metody rodzimej
	public static void main(String args[]){
		Informacje i = new Informacje();
		int status = i.infoSystemu(”CZAS”);
	}
}// koniec class Informacje
```

Powyższy szkic stosowania metod rodzimych zawiera trzy bloki. Pierwszy z nich deklaruje metodę rodzimą, która różni się od pozostałych metod tym, że używany jest specyfikator „native” w deklaracji. Drugi blok to kod statyczny ładowany przy interpretowaniu (kompilacji) kodu bajtów pobierający bibliotekę przechowujący realizację metody rodzimej. Ostatni blok to zastosowanie metody rodzimej. Zadeklarowanie metody jako „native” oznacza, że kompilator ma uznać daną metodę jako rodzimą zdefiniowaną i zaimplementowaną poza Javą. Podana w deklaracji metody rodzimej nazwa jest odwzorowywana później na nazwę funkcji w kodzie rodzimym zgodnie z regułą:
nazwa -> Java_NazwaPakietu_NazwaKlasy_nazwa, czyli np.

infoSystemu -> Java_Informacje_infoSystemu, (brak nazwy pakietu, gdyż klasa Informacje zawiera się w pakiecie domyślnym, który nie posiada nazwy).
Wykorzystanie bibliotek, w których znajduje się realizacja metod rodzimych wymaga, aby biblioteki te były dostępne dla uruchamianego programu, tzn. musi być odpowiednio ustalona ścieżka dostępu.

8.1.2 Kompilacja i generacja plików nagłówkowych

Kompilacja kodu Javy wykorzystującego metody rodzime odbywa się tak samo jak dla czystego kodu Javy, np. javac -g Informacje.java.
Nowością jest natomiast wygenerowanie pliku nagłówkowego, jaki zawarty będzie w kodzie metody rodzimej. Generacja taka wymaga zastosowania narzędzia javah, które generuje plik nagłówkowy o nazwie takiej jak podana nazwa klasy z rozszerzeniem .h (header - nagłówek) tworzony w tym samym katalogu gdzie znajduje się plik klasy programu. Przykładowo wywołanie polecenia:

javah -jni Informacje

spowoduje wygenerowanie następującego pliku nagłówkowego:

Przykład 8.2:

```
//Informacje.h:

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Informacje */

#ifndef _Included_Informacje
#define _Included_Informacje
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Informacje
 * Method:    infoSystemu
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_Informacje_infoSystemu
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Jak widać jest to nagłówek dla kodu metody rodzimej tworzonego w C/C++. Zasadniczo można tworzyć implementacje metod rodzimych w innych językach ale to wymaga indywidualnego podejścia (konwersji typów i nazw), stąd zaleca się korzystanie jedynie z C/C++ do obsługi metod rodzimych. W zasadniczej części nagłówka (poza analizą składni dla C czy C++) zawarto opis metody używając do

tego komentarza zawierającego nazwę klasy, w ciele której zadeklarowano metodę rodzimą, nazwę metody rodzimej w kodzie Javy oraz podpis (sygnatura) metody. Sygnatura metody ma format *(typy-argumentów)typy-zwracane*; gdzie poszczególne typy są reprezentowane przez swoje sygnatury, i tak:

sygnatura	znaczenie typu w Javie
Z	**boolean**
B	**byte**
C	**char**
S	**short**
I	**int**
J	**long**
F	**float**
D	**double**
Lpełna-nazwa-klasy	**pełna nazwa klasy**
[typ	**typ[]**

W powyższym przykładzie pliku nagłówkowego widnieje informacja, że metoda zawiera argument o sygnaturze *Ljava/lang/String* czyli obiekt klasy *String* oraz zwraca wartość typu o sygnaturze *I* czyli typu *int*. Można określić sygnatury typów argumentów i wartości zwracanych metod poprzez użycie narzędzia de-asemblacji kodu a mianowicie *javap*:

```
javap -s -p Informacje
```

co wygeneruje:

```
Compiled from Informacje.java
class Informacje extends java.lang.Object {
   static {};
      /*  ()V  */
   Informacje();
      /*  ()V  */
   public native int infoSystemu(java.lang.String);
      /*  (Ljava/lang/String;)I  */
   public static void main(java.lang.String[]);
      /*  ([Ljava/lang/String;)V  */
}
```

W powyższym wydruku w opcji komentarza zawarto sygnatury typów.

Po opisie metody rodzimej następuje deklaracja metody dla środowiska rodzimego czyli C/C++. Deklaracja ta oprócz omówionej już nazwy zawiera szereg nowych elementów i tak:

JNIEXPORT i JNICALL - oznaczenie funkcji eksportowanych z bibliotek (jest to odniesienie się do specyfikacji deklaracji funkcji eksportowanych, JNIEXPORT oraz JNICALL są zdefiniowane w jni_md.h);

jint lub inny typ - oznaczenie typu elementu zwracanego przez funkcję, np.

typ w Javie,	typ rodzimy,	rozmiar,	typ dla C/C++ (Win32)
boolean	**jboolean**	**8**	**unsigned unsigned char**
byte	**jbyte**	**8**	**signed char**
char	**jchar**	**16**	**unsigned unsigned short**
short	**jshort**	**16**	**short**
int	**jint**	**32**	**long**
long	**jlong**	**64**	**__int64**
float	**jfloat**	**32**	**float**
double	**jdouble**	**64**	**double**
void	**void**		**void**

*JNIEnv** wskaźnik interfejsu JNI zorganizowany jako tablica funkcji JNI o konkretnej lokalizacji. Metoda rodzima wykorzystuje funkcje poprzez odwołanie się do wskaźnika JNIEnv*. Przykładowo można pobrać rozmiar macierzy wykorzystując funkcję GetArrayLength() poprzez wskaźnik JNIEnv*:

```
(...) JNIEnv *env (...) jintArray macierz (...)
jsize rozmiar = (*env)->GetArrayLength(env, macierz);
```

jobject to kolejny element deklaracji funkcji w JNI. Argument tego typu stanowi referencję do bieżącego obiektu; jest odpowiednikiem this w Javie.
Warto tu zauważyć, że odpowiednikiem metody Javy zadeklarowanej jako "native" jest funkcja zawierająca co najmniej dwa argumenty, nawet wówczas gdy metoda nie zawiera żadnego.

jstring lub inne typy argumentów - kolejne argumenty funkcji (argumenty metody rodzimej).

8.1.3 Implementacja metody rodzimej - funkcja a biblioteka

Kolejny krok stosowania funkcji w kodzie Javy to stworzenie tych funkcji lub ich adaptacja do formy wymaganej przez JNI. Deklaracja realizowanej funkcji musi być taka sama jak założona (wygenerowana) w pliku nagłówkowym. Następujący przykład ukazuję przykładową realizację metody rodzimej deklarowanej we wcześniejszych przykładach:

Przykład 8.3:

```
//informacje.cpp

#include "jni.h"
 #include <stdio.h>
 #include <dos.h>
 #include "Informacje.h"


JNIEXPORT jint JNICALL Java_Informacje_infoSystemu(JNIEnv *env, jobject o, jstring str){
   struct  time t;
   const char *s=(env)->GetStringUTFChars(str,0);
```

```
    printf("Obsługiwana aktualnie opcja to: %s\n",s);
    if ((((*s=='C')&&(*(s+1)=='Z'))&&(*(s+2)=='A'))&&(*(s+3)=='S')){
      gettime(&t);
          printf("Bieżący czas to: %2d:%02d:%02d.%02d\n",t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
      return 1;
    } else {
          printf("Nie obsługiwana opcja");
      return 0;
    }
}
```

W powyższym kodzie funkcji w C++ włączono szereg plików nagłówkowych. Plik *stdio.h* oraz *dos.h* to standardowe pliki nagłówkowe biblioteki C. Pierwszy jest wymagany ze względu na stosowanie funkcji *printf*, drugi ze względu na dostęp do czasu systemowego poprzez funkcję *gettime()* oraz strukturę *time*. Ponadto zawarto dwa pliki nagłówkowe wymagane ze względu na implementację interfejsu JNI, czyli jni.h (definicja typów i metod) oraz Informacje.h (deklaracja funkcji odpowiadającej metodzie rodzimej). Pierwsze linia kodu funkcji zawiera deklarację zmiennej typu struktury *time* przechowującej informacje związane z czasem systemowym po wykorzystaniu funkcji *gettime*() w dalszej części kodu. Kolejna linia kodu jest bardzo ważna ze względu na zobrazowanie działania konwersji danych. Wykorzystano tam funkcję *GetStringUTFChars*() do konwersji zmiennej (argumentu funkcji) typu *jstring* do *const char* *. Konwersja ta jest wymagana gdyż nie występuje wprost odwzorowanie typów *String* na *const char *.* Jest to również spowodowane tym, że Java przechowuje znaki w Unicodzie i dlatego konieczna jest dodatkowa konwersja znaków z typu *String*(Unicode -UTF) na *char* *. Warto zwrócić uwagę na wywołanie funkcji konwertującej. Odbywa się ono poprzez wykorzystanie wskaźnika (*env*) do struktury przechowującej szereg różnych funkcji (określonych w jni.h). Ze względu na sposób zapisu funkcji w jni.h możliwe są dwa sposoby wywołania funkcji albo:
(env*)->GetStringUTFChars(env*, str ,0) dla C albo:
(env)->GetStringUTFChars(str,0) dla C++.
Obie funkcje są zapisane w jni.h, druga z nich jest realizowana poprzez zastosowanie pierwszej z pierwszym argumentem typu *this*.
Dalsza część kodu przykładu to wyświetlenie komunikatu zawierającego wartość argumentu tekstowego po konwersji, pobranie informacji o czasie systemowym i wyświetlenie go.
Tak przygotowany kod funkcji należy następnie skompilować i wygenerować bibliotekę (w prezentowanym przykładzie będzie to biblioteka typu Dynamic Link Library - DLL).
Należy pamiętać o lokalizacji plików jni.h oraz innych plików nagłówkowych wołanych przez jni.h. Pliki te są zapisane w podkatalogu *include* oraz *include/win32* głównego katalogu JDK.

Tak przygotowana funkcja i biblioteka umożliwiają uruchomienie programu w Javie wykorzystującego metodę rodzimą. W wyniku wywołania prezentowanego na początku kodu :
java Informacje
uzyskamy rezultat typu:

Obsługiwana aktualnie opcja to: CZAS
Bieżący czas to: 16:11:06.50

gdzie podawany czas zależy oczywiście od ustawień systemowych.

8.1.4 Dostęp do metod i pól zdefiniowanych w Javie

W celu wykorzystania w funkcji metod i metod statycznych zdefiniowanych w Javie należy wykonać trzy podstawowe kroki:
1. pobrać obiekt klasy (Class) w ciele której znajdować ma się żądana metoda:
jclass c = (env)->GetObjectClass(o); gdzie o jest zmienną typu jobject
2. pobrać identyfikator metody dla danej klasy poprzez podanie nazwy metody oraz sygnatur:
jmethodID id = (env)->GetMethodID(c, ”suma”, ”(II)I”); id przyjmuje wartość 0 jeżeli nie ma szukanej metody w klasie reprezentowanej przez obiekt c,
3. wywołać metodę poprzez podanie identyfikatora id oraz obiektu o, np.:
(env)->CallIntMethod(o,id,a1,a2); gdzie a1 i a2 to argumenty. W zależności od zwracanego typu można wykorzystać różne funkcje np. CallVoidMethod(), CallBooleanMethod(), itp.

Wykorzystanie metod statycznych jest analogiczne do powyższego schematu z tą różnicą, że w kroku 2 i 3 należy wywołać odpowiednie funkcje *GetStaticMethodID*() i *CallStaticIntMethod*() (lub podobne innych typów), przy czym funkcje CallStatic* jako argument wykorzystują zmienną typu *jclass* zamiast *jobject* (odwołanie się do obiektu klasy *Class* reprezentującego klasę zamiast do obiektu będącego wystąpieniem klasy).

Przykładowe fragmenty kodów w Javie i w C obrazujące stosowanie metod mogą być następujące:

```
//Liczenie.java:

(...)
public native void oblicz(int a, int b);
public int suma(int a, int b){
        return (a+b);
}
(...)

//policz.cpp:

(...) JNIEXPORT void JNICALL Java_Liczenie_oblicz(JNIEnv *env, jobject o, jint a, jint b){
        jclass c = (env)->GetObjectClass(o);
        jmethodID id =  (env)->GetMethodID(c, ”suma”, ”(II)I”);
        if (id==0)
                return;
        printf(„Oto wartość sumy argumentów: %d”, (env)->CallIntMethod(o,id,a,b));
}
```

Dostęp do pól obiektów i zmiennych statycznych klas Javy z poziomu funkcji jest wykonywany w podobny sposób jak dla metod. Wyróżnia się trzy kroki postępowania:

1. pobrać obiekt klasy (Class) w ciele której znajdować ma się żądana zmienna:

jclass c = (env)->GetObjectClass(o); gdzie o jest zmienną typu jobject
2. pobrać identyfikator zmiennej dla danej klasy poprzez podanie nazwy zmiennej oraz sygnatury:
jfielddID id = (env)->GetFieldID(c, ”a”, ”I”); id przyjmuje wartość 0 jeżeli nie ma szukanej zmiennej w klasie reprezentowanej przez obiekt c,
3. pobrać lub ustawić wartość zmiennej poprzez podanie identyfikatora id oraz obiektu o, np.:
jint a = (env)->GetIntField(o,id);
lub
(env)->SetIntField(o,id,a);
W zależności od typu zmiennej można wykorzystać różne funkcje np. SetObjectField(), GetObjectField(), GetLongField(), SetDoubleField(), itp.

Wykorzystanie pól statycznych jest analogiczne do powyższego schematu z tą różnicą, że w kroku 2 i 3 należy wywołać odpowiednie funkcje *GetStaticFieldID*() i *SetStaticIntField*() (lub podobne innych typów), przy czym funkcje *{Set,Get}Static** jako argument wykorzystują zmienną typu jclass zamiast jobject (odwołanie się do obiektu klasy *Class* reprezentującego klasę zamiast do obiektu będącego wystąpieniem klasy).

Przykładowe fragmenty kodów w Javie i w C ukazujące dostęp do pól mogą być następujące:

```
//Liczenie.java:
class Liczenie{
static String str=”Operacja zakonczona”;
int suma = 10;
(...)
public native void oblicz(int a, int b);
public int suma(int a, int b){
        return (a+b);
}
(...)
}

//policz.cpp:

(...) JNIEXPORT void JNICALL Java_Liczenie_oblicz(JNIEnv *env, jobject o, jint a, jint b){
        jclass c = (env)->GetObjectClass(o);
        jfieldID id =  (env)->GetFieldID(c, ”suma”, ”I”);
        if (id==0)
                return;
        jint s = (env)->GetIntField(o,id);
        printf(„Oto wartość sumy argumentów: %d”,s);
        (env)->SetIntField(o,id, (env)->CallIntMethod(o,id,a,b));
        id =  (env)->GetStaticFieldID(c, ”str”, ”Ljava/lang/String;”);
        if (id==0)
                return;
        jstring tekst = (env)->GetStaticObjectField(c,id);
        printf(tekst);

}
```

Na zakończenie tej sekcji warto przedstawić w skrócie zagadnienia związane z wyjątkami. Wyjątki mogą pojawić się w różnych okolicznościach w pracy z metodami rodzimymi np. przy braku wzywanej metody, przy braku wzywanego pola, przy złym argumencie, itp. JNI dostarcza klika funkcji do obsługi zdarzeń w kodzie funkcji. Podstawowe funkcje to *ExceptionOccurred*(), *ExceptionDescribe*() i *ExceptionClear*(). Pierwsza funkcja bada czy wystąpił wyjątek; jeśli tak to zwraca wartość logiczną odpowiadającą prawdzie. Wówczas można w bloku instrukcji warunkowej zawrzeć pozostałe dwie funkcje (wysłanie informacji o wyjątku na ekran, oraz wyczyszczenie wyjątku). Dodatkowo można wywołać wyjątek, który zostanie zwrócony do kody Javy, gdzie należy go obsłużyć. Można to zrobić za pomocą *funkcji Throw(jthrowable o)* lub *ThrowNew(jclass, const char *)*.

Należy pamiętać również o tym, że JNI umożliwia korzystanie z Maszyny Wirtualnej w ramach aplikacji rodzimej. W tworzeniu takiej aplikacji zanim wywoływane będą metody i pola (tak jak to ukazano do tej pory) konieczne jest wywołanie Maszyny Wirtualnej Javy oraz uzyskanie obiektu klasy, np.:

```
JDK1_1InitArgs vm;
JavaVM *jvm;
JNIEnv *env;
jclass c;

vm.version=0x00010001;
//określenie wersji Maszyny Wirtualnej jako 1.1.2 i wyższych
JNI_GetDefaultJavaVMInitArgs(&vm);
jint test = JNI_CreateJavaVM(&jvm,&env,&vm);

c=(env)->FindClass(”NazwaKlasy”);

// np. GetStatic{Method,Field}Int(c,...) i tak dalej
```

8.2 Programowanie sieciowe

Tworzenie programów działających w sieciach komputerowych było jednym z celów stworzenia języka Java. Możliwości aplikacji sieciowych tworzonych za pomocą Javy są różne. Podstawowe, proste rozwiązania wykorzystujące mechanizmy wysokiego poziomu pracy w sieci dostarczane są przez podstawowe klasy połączeń (adresowania) jak *InetAddress* i *URL*.

8.2.1 Adresowanie komputerów w sieci (InetAddress i URL)

Klasa *InetAddress* opisuje adres komputera w sieci poprzez nazwę/domenę, np. *www-med.eti.pg.gda.pl* oraz poprzez numer IP, np. 153.19.51.66. Istnieje szereg metod statycznych klasy *InetAddress* wykorzystywanych do tworzenia obiektu klasy, brak jest bowiem konstruktorów. Podstawowe metody wykorzystywane do tworzenia obiektów to:

InetAddress.getByName(String nazwa),

InetAddress.getAllByName(String nazwa),
InetAddress.getLocalHost();

Pierwsza metoda tworzy obiekt klasy bazując na podanej nazwie komputera lub adresie. Druga metoda jest wykorzystywana wówczas kiedy komputer o danej nazwie ma wiele adresów IP. Zwracana jest wówczas tablica obiektów typu *InetAddress*. Ostania metoda jest wykorzystywana do uzyskania obiektu reprezentującego adres komputera lokalnego. Wszystkie metody muszą zawierać deklaracje lub obsługę wyjątku *UnknownHostException* powstającego w przypadku braku identyfikacji komputera o podanej nazwie lub adresie. Poniżej zaprezentowano przykładowy program wykorzystujący prezentowane metody klasy *InetAddress*.

Przykład 8.4:

```
//Adresy.java

//Adresy.java

import java.net.*;

public class Adresy{

 public static void main(String args[]){

   try{
     InetAddress a0 = InetAddress.getLocalHost();
     System.out.println("Adres komputera "+a0.getHostName()+" to: " +a0);
     InetAddress a1 = InetAddress.getByName("biomed.eti.pg.gda.pl");
     System.out.println("Adres komputera biomed to: "+a1);
     InetAddress a2[] = InetAddress.getAllByName("www.eti.pg.gda.pl");
     System.out.println("Adres komputera www.eti.pg.gda.pl to:");
     for(int i=0; i<a2.length; i++){
            System.out.println(a2[i]);
     }
   } catch (UnknownHostException he) {
     he.printStackTrace();
   }

 }

}// koniec public class Adresy
```

W wyniku działania powyższego programu wyświetlone zostaną informacje na temat adresów sieciowych wybranych komputerów. W programie zastosowano również jedną z metod klasy *InetAddress* umożliwiającą operacje na adresie a mianowicie *getHostName*(). Metoda ta zwraca nazwę komputera jako obiekt klasy String. Istnieją również metody umożliwiające filtrację adresu komputera w celu uzyskania jedynie numeru IP: *byte[] getAddress(), String getHostAddress().*

Inną klasą wykorzystywaną w Javie do adresowania komputerów jest klasa *URL* oraz jej pochodne (*URL, URLClassLoader, URLConnection, URLDecoder, URLEncoder, URLStreamHandler). URL* czyli Uniform Resource Locator jest specjalną formą adresu zasobów w sieci. *URL* posiada dwa podstawowe elementy:

identyfikator protokołu oraz nazwę zasobów. Identyfikator protokołu to np*. http, ftp, gopher, rmi* czy *jdbc*. Nazwę zasobów stanowią takie elementy jak: nazwa hosta, nazwa pliku, numer portu, nazwa odwołania (w danym pliku). Tworząc więc obiekt klasy URL otrzymujemy gotowy wskaźnik, który jest wykorzystywany przez liczne metody Javy (np. otwieranie obrazka *getImage*(), tworzenie połączenia w JDBC - Connection). W odróżnieniu od klasy *InetAddress* tworzenie obiektów klasy *URL* odbywa się poprzez wykorzystanie jednego z licznych konstruktorów. Każdy z nich związany jest z koniecznością obsługi wyjątku *MalformedURLException* powstającym w wypadku problemów z identyfikacją wskazanego w wywołaniu konstruktora protokołu. Liczne konstruktory umożliwiają podania parametrów adres URL albo poprzez odpowiednik adresu będący tekstem (*String*) albo poprzez tworzenie tego adresu z takich elementów jak nazwa protokołu, nazwa komputera, port, nazwa pliku, itp. Przykładowe konstruktory mają postać:

URL(String adres) ,
URL(String protokół, String host, int port, String plik)

Klasa *URL* zawiera szereg metod umożliwiających filtrację adresu, a więc pobranie nazwy protokołu (*getProtocol*()), nazwy komputera (*getHost*()), pliku (*getFile*()) czy numeru portu (*getPort*()). Dodatkowo klasa *URL* zawiera metody umożliwiające wykonywanie połączenia z hostem (tworzone jest gniazdo – o czym dalej) i przesyłanie danych. Przykładowy program ukazuje możliwość wykorzystania funkcjonalności klasy *URL*:

Przykład 8.5:

```
//Pobiez.java

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

public class Pobiez{

 public static void main(String args[]){
   URL url;
   String tekst;
   if (args.length !=1) {
          System.out.println("Wywołanie: Pobiez URL; gdzie URL to adres zasobów");
          System.exit(1);
   }
   try{
          url = new URL(args[0]);
          BufferedReader br = new BufferedReader(new InputStreamReader(url.openStream()));
          while( (tekst=br.readLine()) !=null){
                 System.out.println(tekst);
          }
   } catch (Exception e) {
          e.printStackTrace();
   }

 }

}// koniec public class Pobiez
```

Powyższy program umożliwia pobranie źródła wskazanego pliku (*html*) i wyświetlenie go na ekranie. Można oczywiście tak skonstruować strumień aby pobierany plik był nagrywany do lokalnego pliku i w ten sposób stworzyć narzędzie do kopiowania stron z Internetu. W przykładzie zastosowano konstrukcję obiektu klasy *URL* poprzez podanie argumentu do konstruktora jako tekstu (*String*) będącego argumentem wywołania aplikacji. Przykładowe wywołanie aplikacji może być następujące:

```
java Pobiez http :// biont.eti.pg.gda.pl/java.htm
```

w wyniku czego uzyskamy wydruk:

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
  <meta name="Author" content="Jacek Ruminski">
  <meta name="GENERATOR" content="Mozilla/4.6 [en-gb] (WinNT; I) [Netscape]">
  <title>Jezyk JAVA - wyklad</title>
</head>
<body text="#FFFFFF" bgcolor="#330033" link="#0000EE" vlink="#551A8B" alink="#FF0000">

<center><b><i><font size=+3>Jŕzyk <img SRC="javalogo.gif" height=88 width=52
align=ABSCENTER>
wyk│ad</font></i></b>
<br> 
<p><img SRC="jav_logo.jpg" height=125 width=260>
<p><b><font color="#CC0000"><font size=+1>Plan wyk│adu:</font></font></b></center>

(...)
```

8.2.2 Komunikacja przez Internet (klient-serwer)

Do komunikacji poprzez Internet programy Javy wykorzystują protokoły TCP i UDP. Klasy *URL** oraz *Socket* i *ServerSocket* wykorzystują Transfer Control Protocol klasy takie jak *DatagramPacket*, *DatagramSocket*, oraz *MulticastSocket* korzystają z User Datagram Protocol. W pakiecie java.net zdefiniowane są jeszcze inne klasy, z których warto przytoczyć te związane z autoryzacją połączeń i nadawaniem uprawnień: *Authenticator*, *NetPermission*, *PasswordAuthentication*, *SocketPermission*.

W aplikacjach klient-serwer, serwer dostarcza określonej usługi np. przetwarza zapytanie skierowane do bazy danych, zapisuje serię obrazów diagnostycznych. Klient wykorzystuje usługi świadczone przez serwer i jest odpowiedzialny za żądanie usługi oraz obsługę wyników. Funkcje pełnione przez każdą ze stron można ująć następująco:
- połączenie z urządzeniem zdalnym (przygotowanie wysyłania i odbioru danych),
- wysyłanie danych,
- odbiór danych,
- zamknięcie połączenia,
- przywiązanie portu (dla danej aplikacji na danym hoście),

- nasłuch,
- akceptacja połączeń na danych portach.

Pierwsze cztery funkcje są właściwe dla klienta, serwer rozszerza je o dodatkowe trzy. W Javie obsługę klienta, a więc realizację pierwszych czterech funkcji dostarcza klasa *Socket*; dla serwera przeznaczono klasę *ServerSocket*. *Socket* (gniazdo) to dobrze znane pojęcie wynikające z planów stworzenia dostępu do sieci jako do strumienia danych (strumień wejścia, strumień wyjścia). Koncepcja strumieni miała być uniwersalna dla każdego możliwego przepływu danych (z urządzenia, z pliku, z sieci). *Socket* jest więc pewną abstrakcją umożliwiającą przejście na wyższy poziom i nie zajmowaniem się takimi zagadnieniami jak rozmiar pakietu, retransmisja pakietu, typ mediów, itp. Rozważmy teraz zagadnienia związane ze stroną klienta procesu komunikacji sieciowej, a więc zapoznajmy się z klasą *Socket* pakietu java.net.

Klasa *Socket* posiada szereg różnych konstruktorów, z których najbardziej popularne są:
Socket(InetAddress address, int port) - gdzie address to obiekt klasy InetAddress będący adresem IP lub nazwą hosta, port - numer portu (0-65535),
Socket(String host, int port) - gdzie host to tekst oznaczający nazwę hosta, port - numer portu (0-65535).

Ponieważ połączenie wykonywane przez obiekt klasy *Socket* może nie powieść się z różnych przyczyn (np. nie znany host) konieczna jest obsługa wyjątków.
Klasyczny fragment kodu obsługi klasy *Socket* pokazano poniżej:

```
try {
      Socket  gniazdo= new Socket("www.amg.gda.pl", 80);
    }
    catch (UnknownHostException e) {
      System.err.println(e);
    }
    catch (IOException e) {
      System.err.println(e);
   }
```

W powyższym przykładzie podjęta jest próba (*try*) połączenia z serwerem www.amg.gda.pl na porcie 80. Jeżeli nazwa hosta jest nieznana lub serwer nazw nie działa zostanie zwrócony wyjątek *UnknownHostException* (wyjątek nieznanego hosta). Jeśli z innych przyczyn nie uda się uzyskać połączenia zostanie zwrócony wyjątek *IOException* (wyjątek wejścia-wyjścia). Najprostszym przykładem programu implementującego klasę *Socket* może być prosty skaner portów serwera:

Przykład 8.6:

```
//SkanerPortow.java:

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


public class SkanerPortow {
```

```
  public static void main(String[] args) {

    Socket gniazdo;
    String host = "localhost";

    if (args.length > 0) {
      host = args[0];    //jeśli nie podano argumentu programu hostem bedzie komputer lokalny
    }
    for (int n = 0; n < 1024; n++) {   //skanuj wszystkie porty "roota"
      try {
        gniazdo = new Socket(host, n);
        System.out.println("Znalazłem serwer na porcie " + n + " komputera: " + host);
      }
      catch (UnknownHostException e) {
        System.err.println(e);
        break; //koniec pętli for w razie nieznanego hosta
      }
      catch (IOException e) {
        // System.err.println(e);  - nie drukuj informacji o braku serwera
       }
    }

  }

}  // koniec public class SkanerPortow
```

Program powyższy jest skanerem portów na których działają aplikacje (serwery). Skanowane porty 0-1023 należą do zakresu portów, które na maszynach UNIX-owych przynależą do administratora, to znaczy tylko *root* może uruchamiać serwery na tych portach. Odwzorowanie portów na aplikację można znaleźć w pliku ustawień /etc/services. Klasa *Socket* posiada wiele metod umożliwiających między innymi uzyskanie informacji związanych z obiektem tej klasy takich jak:
- getLocalAddress() - zwraca lokalny adres, do którego dowiązane jest gniazdo,
- getLocalPort() - zwraca lokalny port, do którego dowiązane jest gniazdo,
- getInetAddress() - zwraca zdalny adres, do którego gniazdo jest podłączone,
- getPort() - zwraca zdalny numer portu, do którego gniazdo jest podłączone.

Kolejny przykład obrazuje wykorzystanie metody *getLocalPort*().

Przykład 8.7:

```
// SkanerPortow2.java:

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

public class SkanerPortow2 {

  public static void main(String[] args) {

    Socket gniazdo;
    String host = "localhost";
```

```
    if (args.length > 0) {
      host = args[0];
    }
    for (int n = 0; n < 1024; n++) {
      try {
        gniazdo = new Socket(host, n);
         int lokalPort = gniazdo.getLocalPort();
         System.out.println("Numer lokalnego portu to:" + lokalPort);
        System.out.println("Znalazłem serwer na porcie " + n + " komputera: " + host);
      }
      catch (UnknownHostException e) {
        System.err.println(e);
        break;
      }
      catch (IOException e) {
        //System.err.println(e);
       }
    }

  }
}// koniec public class SkanerPortow2
```

Powiedziano wcześniej, że idea gniazd związana była ze stworzeniem strumienia, do którego można pisać, i z którego można czytać. Podstawową, uniwersalną a zarazem abstrakcyjną klasą obsługującą strumień wejściowy jest klasa *InputStream* dla strumienia wyjściowego jest to *OutputStream*. Obydwie klasy są rozszerzane i stanowią podstawę przepływu danych (np. z i do pliku, z i do urządzenia, z i do sieci). W celu obsługi sieci w klasie *Socket* zdefiniowano metody zwracające obiekty klas *InputStream* oraz *OutputStream*, co umożliwia stworzenie strumieni do przesyłania danych przez sieć. Najczęściej wykorzystywane klasy nie abstrakcyjne to *BufferedReader* obsługujące czytanie danych przez bufor oraz *PrintStream* dla obsługi zapisu nie zwracająca wyjątku *IOException*. W celu zobrazowania pracy ze strumieniami posłużmy się przykładami. Pierwszy przykład prezentuje działanie typowe dla klienta, a więc czytanie ze strumienia.
Przykładowy program łączy się z serwerem czasu a następnie wyświetla ten czas na ekranie.

Przykład 8.8:

```
//Zegar.java:

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

public class Zegar {

  public static void main(String[] args) {

    Socket gniazdo;
    String host = "localhost";
    BufferedReader strumienCzasu;

    if (args.length > 0) {
```

```
      host = args[0];
    }

      try {
        gniazdo = new Socket(host, 13);
        strumienCzasu = new BufferedReader(new InputStreamReader(gniazdo.getInputStream()));
        String czas = strumienCzasu.readLine(); //wprowadź linię znaków z bufora strumienia
        System.out.println("Na "+host+" jest: "+czas);

      }
      catch (UnknownHostException e) {
       System.err.println(e);
      }
      catch (IOException e) {
       System.err.println(e);
       }

  }
}//koniec public class Zegar
```

Kolejny przykład umożliwi pokazanie zarówno stworzenie programu klienta jak i programu serwera.

Przykład 8.9:

```
//KlientEcho.java:

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

public class KlientEcho {

  public static void main(String[] args) {

    Socket gniazdo;
    String host = "localhost";
    BufferedReader strumienEcha, strumienWe;
    PrintStream strumienWy;
    String echo;

    if (args.length > 0) {
      host = args[0];
    }

      try {
       gniazdo = new Socket(host, 7);  //port 7 jest standardowym portem obslugi echa
        strumienWe = new BufferedReader(new InputStreamReader(gniazdo.getInputStream()));
          //czytaj z serwera
        strumienWy = new PrintStream(gniazdo.getOutputStream());
       strumienEcha = new BufferedReader(new InputStreamReader(System.in));
       //czytaj z klawiatury
        while(true){
          echo=strumienEcha.readLine();
           if (echo.equals(".")) break; //znak . oznacza koniec pracy
```

```
            strumienWy.println(echo); //wyslij do serwera
            System.out.println(strumienWe.readLine());  //wyslij na monitor
          }
        }
        catch (UnknownHostException e) {
          System.err.println(e);
        }
        catch (IOException e) {
          System.err.println(e);
        }

    }
}//koniec public class KlientEcho
```

Program serwera

Przykład 8.10:

```
//SerwerEcho.java:

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

public class SerwerEcho {

  public static void main(String[] args) {

    ServerSocket serwer;
    Socket     gniazdo;
    String host = "localhost";
    BufferedReader strumienEcha, strumienWe;
    PrintStream strumienWy;
    String echo;

    if (args.length > 0) {
      host = args[0];
    }

      try {
       serwer = new ServerSocket(7); //stworz serwer pracujacy na porcie 7 biezacego komputera
         while(true){ //główna pętla serwera
            try{
              while(true){ //główna pętla połączenia
                gniazdo = serwer.accept();  //przyjmuj połączenia i stworz gniazdo
                System.out.println("Jest polaczenie");
                while(true){
                  strumienWe = new BufferedReader(new InputStreamReader(gniazdo.getInputStream()));
                  strumienWy = new PrintStream(gniazdo.getOutputStream());
                  echo=strumienWe.readLine();
                  strumienWy.println(echo); //wyslij to co przyszlo
                }//od while

            }//od while
```

```
        }//od try
        catch (SocketException e){
          System.out.println("Zerwano polaczenie"); //klient zerwał połączenie
        }
        catch (IOException e) {
           System.err.println(e);
        }
      }//od while
    }// od try
    catch (IOException e) {
      System.err.println(e);
    }
  }
}//koniec public class SerwerEcho
```

Przykład SerwerEcho.java demonstruje zastosowanie klasy *ServerSocket* do tworzenia serwerów w architekturze klient-serwer. Klasa *ServerSocket* dostarcza kilka konstruktorów umożliwiających stworzenie serwera na danym porcie, z określoną maksymalną liczbą połączeń (kolejka), na danym porcie danego komputera o konkretnym IP (ważne w przypadku wielu interfejsów) z określoną maksymalną liczbą połączeń (kolejka). Kolejnym istotnym elementem implementacji klasy *ServerSocket* jest oczekiwanie i zamykanie połączeń. W tym celu wykorzystuje się metody *ServerSocket.accept*() oraz *Socket.close*(). Metoda *accept*() blokuje przepływ instrukcji programu i czeka na połączenie z klientem. Jeśli klient podłączy się do serwera metoda *accept*() zwraca gniazdo, na którym jest połączenie. Zamknięcie połączenia odbywa się poprzez wywołanie metody *close*() dla stworzonego gniazda. Wówczas połączenie jest zakończone i aktywny jest nasłuch *accept*() oczekujący na następne podłączenie. Jeżeli jest potrzeba zwolnienia portu serwera to wówczas należy wywołać metodę *ServerSocket.close*(). Inne przydatne metody klasy *ServerSocket* to *getLocalPort*() umożliwiająca uzyskanie numeru portu na jakim pracuje serwer oraz metoda *setSoTimeout*() umożliwiająca danemu serwerowi ustawienie czasu nasłuchu metodą *accept*(). Metoda *setSoTimeout*() musi być wywołana przed *accept*(). Czas podaje się w milisekundach. Jeżeli w zdefiniowanym okresie czasu *accept*() nie zwraca gniazda (brak połączenia) zwracany jest wyjątek *InterruptedException*.

8.3 Serwlety

Serwlet jest definiowaną przez programistę klasą implementującą interfejs javax.servlet.Servlet. Zadaniem serwletu jest wykonywanie działań w środowisku serwera. Oznacza to, że serwlet nie stanowi oddzielnej aplikacji, lecz jego wykonywanie jest zależne od serwera. Serwlet stanowi zatem odmianę apletu, z tym, że działa nie po stronie klienta (w otoczeniu np. przeglądarki WWW) lecz po stronie serwera. Taka forma działanie serwleta wymaga od serwera zdolności do interpretacji kodu pośredniego Javy oraz możliwości wykorzystania produktu wykonanego kodu Javy w obrębie funkcjonalności serwera. Interpretacja kodu pośredniego odbywa się najczęściej za pośrednictwem zewnętrznej względem serwera maszyny wirtualnej co wymaga zaledwie prostej konfiguracji serwera. Zdolność do wykorzystania serwletów przez serwer deklarują producenci danego

serwera. Należy podkreślić, że biblioteki kodów konieczne do tworzenia servletów (javax.servlet.*) nie stanowią części standardowej dystrybucji i muszą być pobrane od producenta (SUN) oddzielnie jako standardowe rozszerzenie JDK. Istnieją różne wersje bibliotek kodów servletów (JSDK - Java Servlet Development Kit), które są obecnie wykorzystywane w różnych serwerach: 2.0 (np. jserv - dodatek do serwera WWW Apache 1.3.x), 2.1, 2.2 (np. tomcat). Przed przystąpieniem do tworzenia serwletów konieczne jest zatem ustalenie wersji biblioteki jakiej należy użyć, aby być zgodnym z obsługiwanym serwerem.

8.3.1 Model obsługi wiadomości.

Serwlety wykorzystują model obsługi wiadomości typu żądanie – odpowiedź (request – response). W modelu tym klient wysyła wiadomość zawierającą żądanie usługi, natomiast serwer wykonuje usługę i przesyła do klienta żądane informacje. Model taki jest związany z większością popularnych protokołów komunikacyjnych jak FTP czy HTTP, stąd serwlety można teoretycznie stworzyć dla różnych typów usług. Działający serwer komunikuje się z serwletem (stanowiącym część procesu serwera) w oparciu o trzy podstawowe metody zadeklarowane w interfejsie javax.servlet.Server:

- **init(),**
- **service()**
- **destroy()**

Rolę i moment wywołania tych metod ukazuje następująca sekwencja zadań wykonywana w czasie uruchamiania serwleta:

a.) serwer ładuje kod pośredni serwletu,
b.) serwer wywołuje obiekt załadowanego kodu,
c.) serwer woła metodę init() serwletu, która umożliwia wszelkie wymagane przez serwlet ustawienia. Metoda ta stanowi odpowiednik metody init() apletu, oraz z funkcjonalnego punktu widzenia odpowiada również konstruktorowi klasy obiektu.
d.) serwer na podstawie danych dostarczonych przez klienta usługi tworzy obiekt żądania (request object), implementujący interfejs ServletRequest,
e.) serwer tworzy obiekt odpowiedzi (response object) implementujący interfejs ServletResponse,
f.) serwer woła metodę serwletu service(), która realizuje zadania (usługi) stworzone przez programistę,
g.) metoda service() przetwarza obiekt żądania i korzystając z metod obiektu odpowiedzi wysyła określoną informację do klienta,
h.) jeżeli serwer nie potrzebuje serwletu woła jego metodę destroy(), której obsługa umożliwia zwolnienie zasobów (jak np. zamknięcie plików, zamknięcie dostępu do bazy danych, itp.).

Model obsługi serwletu jest więc bardzo prosty. Interfejs javax.servlet.Servlet wymaga oczywiście pełnej definicji wszystkich metod stąd tworząc kody klas serwletów nie korzysta się bezpośrednio z implementacji tego interfejsu lecz dziedziczy się po klasach interfejs ten implementujących. Do klas tych należy zaliczyć klasy:

GenericServlet
HttpServlet

definiowane kolejno w dwóch dostępnych pakietach JSDK API:

javax.servlet
javax.servlet.http.

Korzystając z przedstawionego modelu można skonstruować pierwszy przykład serwletu:

Przykład 8.11:

```
//WitajcieServlet.java:

import javax.servlet.*;
import java.io.*;

public class WitajcieServlet extends GenericServlet{

 static String witajcie= "Witajcie ! Jestem serwletem!\n";

 public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException {
  response.setContentLength(witajcie.length());
  response.setContentType("text/plain");
  PrintWriter pw=response.getWriter());
  pw.print(witajcie);
  pw.close();
 }

}// koniec public class WitajcieServlet
```

Powyższy przykład definiuje nową klasę WitajcieServlet, która dziedziczy po GenericServlet, czyli automatycznie implementuje interfejs javax.servlet.Servlet. W ciele klasy serwletu zdefiniowana jedną metodę (przepisano metodę service() klasy GenericServlet) service(), która na podstawie pobranych argumentów będących obiektami żądania i odpowiedzi, wykonuje określoną usługę i przesyła informacje do klienta. W przypadku omawianej metody usługa polega na przesłaniu wiadomości o określonym rozmiarze (setContentLength()), typie (setContentType()) i treści (print(witajcie)) do klienta. Zdefiniowana klasa serwletu korzysta z domyślnych definicji metod init() oraz destroy() występujących w klasie rodzica (GenericServlet).

8.3.2 Środowisko wykonywania serwletów:

W celu uruchomienia serwletu należy go najpierw skompilować do czego wymagana jest oczywiście platforma Javy oraz dodatkowo pakiety kodów serwletów (JSDK). Biblioteki JSDK należy zainstalować, tak aby były widoczne dla kompilatora Javy (np. zawrzeć pakiety kodów serwletów w ustawieniach CLASSPATH lub przegrać archiwum *.jar servletów do katalogu „ext” stanowiącego część biblioteki („lib”) maszyny wirtualnej („jre”). Udana kompilacja kodu umożliwia jego wywołanie. Uruchomienie serwletu wymaga jednak odpowiedniego serwera zdolnego do

prowadzenia dialogu z serwletem. Możliwe tutaj są różne rozwiązania. Najprostszym z nich jest zastosowaniem dostarczanego w ramach JSDK prostego programu servletrunner (odpowiednik funkcjonalny programu appletviewer dla apletów). Ten prosty serwer umożliwia określenie podstawowych parametrów przy jego wywołaniu jak numer portu, katalog serwletów itp. Standardowa konfiguracja serwera wygląda następująco:

```
./servletrunner -v
servletrunner starting with settings:
  port = 8080
  backlog = 50
  max handlers = 100
  timeout = 5000
  servlet dir = ./examples
  document dir = ./examples
  servlet propfile = ./examples/servlet.properties
```

Bardzo ważnym elementem konfiguracji programu servletrunner jest pliku właściwości serwletów: servlet.properties. Wymagane jest aby w pliku tym podana została nazwa servletu reprezentując plik jego klasy, np.:

```
//servlet.properties:

# Servlets Properties
#
# servlet.<name>.code=class name (foo or foo.class)
# servlet.<name>.initArgs=comma-delimited list of {name, value} pairs
#                  that can be accessed by the servlet using the
#                  servlet API calls
#

# Witajcie servlet
servlet.witajcie.code=WitajcieServlet
```

Tak przygotowany plik ustawień własności serwletu oraz uruchomienie programu servletrunner umożliwia wywołanie servletu, np.:

lynx medvis.eti.pg.gda.pl:8080/servlet/witajcie
albo
http://medvis.eti.pg.gda.pl:8080/servlet/witajcie

W konsoli serwera (servletrunner) pojawi się wówczas komunikat:

WitajcieServlet: init

oznaczający wywołanie metody init() serwletu przez serwer.
Warto zwrócić uwagę, że przy wywołaniu serwletu pojawił się w ścieżce dostępu katalog o nazwie servlet. Jest to katalog logiczny, definiowany w ustawieniach serwera (w powyższym przykładzie katalog logiczny „servlet” to katalog fizyczny

„examples”). W wyniku poprawnego działania serwletu klient uzyska wyświetloną następującą informację:

Witajcie ! Jestem serwletem!

Wadą wykorzystania programu servletrunner jest konieczność jego powtórnego wywoływania w przypadku zmiany kodu serwletu. Na rynku dostępne są obecnie różne wersje serwerów, głównie WWW, obsługujących serwlety. Do najbardziej jednak popularnych należy zaliczyć rozszerzenie serwerów WWW opracowanych w ramach projektu Apache – ApacheJServ oraz nowy serwer obsługujący servlety oraz Java Server Pages (dynamiczna konstrukcja stron WWW). Podstawowa różnica pomiędzy tymi produktami jest taka, że pierwszy z nich jest dodatkiem do serwera WWW obsługującym kody serwletów zgodne ze specyfikacją 2.0, natomiast drugi jest referencyjną implementacją specyfikacji serwletów 2.2 pracującą jako samodzielny serwer WWW.

Odwołanie do środowiska wywołania serwletu może następować poprzez wykonanie metod zdefiniowanych dla klasy ServletContext. W czasie inicjowania serwletu (init()) informacje środowiska podawane są w ramach obiektu ServletConfig. Odwołanie się do tego obiektu poprzez metodę getServletContext() w czasie wykonywania serwletu (service()) zwraca obiekt klasy ServletContext. W ten sposób można wywołać na nim szereg metod uzyskując cenne często informacje o środowisku pracy. Poniższy przykład demonstruje wykorzystanie metody getServerInfo() obiektu klasy ServletContext w celu uzyskania informacji o nazwie serwera servletów i jego wersji.

Przykład 8.12:

```
//SerwerServlet.java:

import javax.servlet.*;
import java.io.*;

public class SerwerServlet extends GenericServlet{

	static String witajcie= "Jestem serwletem! Dzia\u0142am na:\n";
	private ServletConfig  ustawienia;
	public void init(ServletConfig cfg){
		ustawienia=cfg;
	}
	public void service(ServletRequest request, ServletResponse response) throws ServletException,
IOException {
		String wiad=witajcie+"";
		ServletContext sc = ustawienia.getServletContext();
		wiad+=sc.getServerInfo();
		response.setContentLength(wiad.length());
		response.setContentType("text/plain; charset=iso-8859-2");
		PrintWriter pw=response.getWriter();
		pw.print(wiad);
		pw.close();
	}

}// koniec public class SerwerServlet
```

Efektem działania powyższego serwletu może być komunikat:

Jestem serwletem! Działam na:
servletrunner/2.0

lub np.:

Jestem serwletem! Działam na:
Tomcat Web Server/3.1 (JSP 1.1; Servlet 2.2; Java 1.2; SunOS 5.5.1 sparc; java.vendor=Sun Microsystems Inc.)

8.3.3 Kontrola środowiska wymiany wiadomości

W modelu wymiany wiadomości typu żądanie-odpowiedź ważną rolę pełni dostęp do informacji konfiguracyjnych występujących w żądaniu oraz możliwość ustawień podobnych informacji dla odpowiedzi. W przypadku serwletów funkcje te możliwe są poprzez obsługę obiektów podawanych jako argumenty w głównych metodach usług serwletu. W przypadku pakietu javax.servlet.* obiekty te posiadają interfejs odpowiednio o nazwach: ServletRequest (dane żądania) i ServletResponse (dane odpowiedzi). Dla pakietu javax.servlet.http.*, w którym zdefiniowano klasy i metody obsługi protokołu HTTP przez servlety, występując interfejsy dziedziczące po powyższych: HttpServletRequest i HttpServletResponse. Obiekty obu typów posiadają do dyspozycji szereg metod umożliwiających dostęp do parametrów wywołania, ustawień konfiguracyjnych i innych. Przykładowe metody obiektów typu *Request to:

getCharacterEncoding() – pobranie informacji o stronie kodowej
getContentLength() – pobranie informacji o rozmiarze wiadomości
getContentType() – pobranie informacji o typie (MIME) wiadomości
getParameter(java.lang.String name) – pobranie wartości parametru o danej nazwie
getProtocol() – pobranie informacji o protokole komunikacji
getRemoteAddr() – pobranie informacji o adresie komputera zdalnego,
getReader() – utworzenie obiektu strumienia do odczytu (znaków)
getInputStream()– utworzenie obiektu strumienia do odczytu (bajtów)

Przykładowe metody obiektów typu *Response to:

getCharacterEncoding() – pobranie informacji o aktualnej stronie kodowej,
getOutputStream()-utworzenie obiektu strumienia do zapisu (bajtów)
getWriter()– utworzenie obiektu strumienia do zapisu (znaków)
setContentLength(int len) – ustawienie rozmiaru wiadomości
setContentType(java.lang.String type) – ustawienie typu (MIME) wiadomości.

Niektóre z powyższych metod stosowane były we wcześniejszych przykładach. Inną ilustracją metod obiektów typu *Request i *Response może być następujący program:

Przykład 8.13:

```
//KlientServlet.java:

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;


public class KlientServlet extends HttpServlet {
        static String wiad = "Oto nag\u0142\u00F3wki: \n";;

        public void doGet(HttpServletRequest request, HttpServletResponse response)   throws IOException,
ServletException{
                String nazwa, wartosc, nag="";
                response.setContentType("text/plain; charset=iso-8859-2");
                // dla wersji nowszej od JSDK 2.0 można ustalić lokalizacje:
                //response.setLocale(new Locale("pl","PL"));
                PrintWriter pw = response.getWriter();
                Enumeration e = request.getHeaderNames();
                while (e.hasMoreElements()) {
                nazwa = (String)e.nextElement();
                        wartosc = request.getHeader(nazwa);
                          nag+=(nazwa + " = " + wartosc+"\n");
                }
                pw.print(wiad+nag);
                pw.close();
        }

}// koniec public class KlientServlet
```

Powyższy serwlet jest zdefiniowany dla klasy typu HttpServlet definiującej szereg metod obsługujących protokół HTTP. Jedną z usług tego protokołu jest pobranie nagłówka i danych – GET. Obsługa tej usługi jest możliwa poprzez zastosowanie metody doGet(), do której są przekazywane obiekty typu HttpServletRequest i HttpServletResponse. Ponieważ wraz z przesyłaniem wiadomości w HTTP przesyła się również nagłówek zawierający informacje sterujące można te informacje pobrać i wyświetlić. Omawiany program pobiera zestaw informacji konfiguracyjnych pochodzących z nagłówków wygenerowanych przez klienta usługi (request.getHeaderNames()). Pobrane informacje są formatowane w postaci tekstowej zgodnie ze stroną kodową „iso-8859-2” i są przesyłane jako treść informacji do klienta (respone). W wyniku działania tego serwletu klient może obserwować w swojej przeglądarce następujący efekt:

Oto nagłówki:
Connection = Keep-Alive
User-Agent = Mozilla/4.6 [en-gb] (WinNT; I)
Host = med16:8080
Accept = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding = gzip
Accept-Language = en-GB,en,en-*
Accept-Charset = iso-8859-1,*,utf-8

8.3.4 Metody wywoływania serwletów.

Serwlety mogą być również (o ile umożliwia to serwer – np. Java Web ServerTM) wywoływane z kodu HTML poprzez zastosowanie mechanizmu SSI (Serwer-Side Include). Technika opisu serwletu w kodzie HTML jest podobna do opisu apletu. W najprostszej formie opis ten może wyglądać następująco:

<servlet name=”witajcie” code=”WitajcieServlet” NAZWA_1=”WARTOSC_1” ... NAZWA_n=”WARTOSC_n”>
<PARAM NAME=”nazwa_1” value=”wartosc_1”>
.
.
.
<PARAM NAME=”nazwa_n” value=”wartosc_n”>
</servlet>

W przedstawionym opisie wykorzystano znacznik <servlet> do oznaczenia zawartego w kodzie HTML wywołania serwletu. W obrębie znacznika zastosowano szereg atrybutów jak:
name – nazwa servletu
code – nazwa pliku kodu serwletu
NAZWA_1, WARTOSC_1 – pary parametrów wywołania serwletu.

Parametry wywołania serwletu mogą być pobrane przez serwlet poprzez użycie metod klas GenericServlet i HttpServlet jak np. getInitParameter(java.lang.String name) dostarczającą wartość parametru o podanej nazwie. W obrębie znacznika <servlet> można zastosować inne znaczniki takie jak <PARAM NAME> umożliwiające zdefiniowanie parametrów dostępnych przez serwlet za pomocą metod obiektów żądania jak np. getParameter(java.lang.String name).

Przykład 8.14:

```
//LicznikServlet.java:

import javax.servlet.*;
import java.io.*;
import java.util.*;

public class LicznikServlet extends GenericServlet{

	static Date pocz = new Date();
	static int licznik=0;

	public void service(ServletRequest request, ServletResponse response) throws ServletException,
IOException {
		int licz_tmp;
		response.setContentType("text/plain; charset=iso-8859-2");
		PrintWriter pw=response.getWriter();
		synchronized(this){
			licz_tmp=licznik++;
		}
```

```
            pw.print("Liczba odwiedzin strony od "+pocz+" wynosi: "+licz_tmp);
            pw.close();
      }

}// koniec public class LicznikServlet

//serwlety.shtml:

<html>
<head>
<title> Serwlety: wywołanie serwletu za pomocą znaczników w kodzie HTML</title>
</head>
<body>
<H1>Strona demonstracyjna</H1>
<H2>Pokaz wywołania serwletu za pomocą znaczników z kodu HTML</H2>
<servlet name="LicznikServlet" code="LicznikServlet">
</servlet>
</body>
</html>
```

Niestety zastosowanie tego typu jest ograniczone ponieważ tylko niektóre serwery je umożliwiają (Java Web ServerTM). Stosowanie dynamicznej kompozycji stron w obrębie kodu HTML jest obecnie realizowane poprzez technologię Java –Server Pages (JSP).

8.3.5 Obsługa protokołu HTTP – pakiet javax.servlet.http.*

Protokół HTTP jest jednym z najbardziej popularnych ze względu na obsługę WWW. Dlatego trudno się dziwić, że opracowano pakiet klas (javax.servlet.http.*) umożliwiający obsługę usług tego protokołu w ramach serwletów. Wyróżnia się następujące usługi (metody) protokołu HTTP:

- HEAD – żądanie informacji od serwera w formie danych nagłówka
- GET- żądanie informacji od serwera w formie danych nagłówka i treści (np. pliku), występuje ograniczenie rozmiaru danych do przesłania (kilkaset bajtów)
- POST – żądanie umożliwiające przesłanie danych od klienta do serwera
- PUT – żądanie umożliwiające przesłanie przez klienta pliku do serwera (analogia do ftp)
- DELETE – żądanie usunięcia dokumentu z serwera
- TRACE – żądanie zwrotnego przesłania nagłówków wiadomości do klienta (testowanie)
- OPTIONS – żądanie od serwera identyfikacji obsługiwanych metod, informacja o których jest przesyłana zwrotnie w nagłówku
- Inne.

W celu obsługi powyższych usług stworzono zestaw metod dostępnych poprzez klasę javax.servlet.http.HttpServlet dziedziczącą po javax.servlet.Servlet, a mianowicie:

doGet() – dla żądań HEAD i GET
doPost() – dla żądania POST

doPut() – dla żądania PUT
doDelete() – dla żądania DELETE
doTrace() – dla żądania TRACE
doOptions() – dla żądania OPTIONS.

Każda z wymienionych metoda jest zadeklarowana jako chroniona (protected). Jedyną metodą dostępną (public) jest metoda service() poprzez którą przechodzą wysyłane przez klienta żądania i są kierowane do obsługi w ramach wyżej pokazanych metod doXXX(). Najprostszą realizacją usługi na żądanie ze strony klienta jest dynamiczne stworzenie strony HTML, którą odczytuje klient.

Przykład 8.15:

```
//StronaServlet.java:

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;


public class StronaServlet extends HttpServlet {
	static String tytul = "Strona g\u0142\u00F3wna serwisu nauki JAVY \n";
	String dane="";

	public void init(ServletConfig config){
		dane=config.getServletContext().getServerInfo();
	}

	public String naglowek(String tytul){
		String nag="<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 3.2//PL\">"
		+ "<HTML>"
		+ "<HEAD>"
		+ " <TITLE>" + tytul + "</TITLE>"
		+" <META NAME=\"Autor\" CONTENT=\"Jacek Rumi\u0144ski\">"
		+ " <META NAME=\"Serwer\" CONTENT=\"" + dane + "\">"
		+ "</HEAD>";
		return nag;
	}
	public String stopka(){
		String stop ="</BODY>"+ "</HTML>";
		return stop;
	}

	public void doGet(HttpServletRequest request, HttpServletResponse response)   throws IOException,
ServletException{
		Date data = new Date();
		response.setContentType("text/html; charset=iso-8859-2");
		PrintWriter pw = response.getWriter();
		String wiad=naglowek(tytul);
		wiad=wiad + "<BODY BGCOLOR=\"#C0C0FF\">"
		+ " <CENTER>"
		+ "  <H1>" + tytul + "</H1>"
		+ "  <H2>Witamy na stronie kursu Javy!</H2>"
```

```
                + "  <H3>[ Czas: <font color='#FFFFC0'>" + data + "</font> ]</H3>"
                + "  <FONT SIZE=\"-2\">Zapraszam na:"
                + "  <A HREF=\"http://biont.eti.pg.gda.pl/java.htm\">J\u0119zyk JAVA</a><br>"
                + "  </FONT>"
                + " </CENTER>";
                wiad+=stopka();

                pw.print(wiad);
                pw.close();
        }

}// koniec public class StronaServlet
```

W wyniku działania powyższego programu można uzyskać następujący (sformatowany) tekst:

Strona główna serwisu nauki JAVY
Witamy na stronie kursu Javy!
[Czas: Tue May 09 16:35:13 GMT+02:00 2000]
Zapraszam na: Język JAVA

Rozważając konstruowanie dynamicznych stron HTML w oparciu o serwlety konieczne jest przedstawienie łączenia ich z formularzami.

Przykład 8.16:

```
//OgloszenieServlet.java

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;


public class OgloszenieServlet extends HttpServlet {
        static String tytul = "Strona testowa serwisu nauki JAVY \n";
        String dane="";

        public void init(ServletConfig config){
                dane=config.getServletContext().getServerInfo();
        }

        public String naglowek(String tytul){
                String nag="<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 3.2 //PL\">"
       + "<HTML>"
       + "<HEAD>"
       + " <TITLE>" + tytul + "</TITLE>"
                + " <META NAME=\"Autor\" CONTENT=\"Jacek Rumi\u0144ski\">"
       + " <META NAME=\"Serwer\" CONTENT=\"" + dane + "\">"
       + "</HEAD>";
                return nag;
        }
        public String stopka(){
```

```
		String stop ="</BODY>"+ "</HTML>";
		return stop;
	}

	public void doGet(HttpServletRequest request, HttpServletResponse response)   throws IOException,
ServletException{
		Date data = new Date();
		response.setContentType("text/html; charset=iso-8859-2");
		PrintWriter pw = response.getWriter();
		String wiad=naglowek(tytul);
		wiad=wiad + "<BODY BGCOLOR=\"#C0C0FF\">"
	+ " <CENTER>"
	+ "  <H1>" + tytul + "</H1>"
	+ "  <H2>[ Lokalny czas to: <font color='#FFFFC0'>" + data + "</font> ]</H2>"
	+ "  <H3>Mi\u0142o jest mi zakomunikowa\u0107, \u017Ce:</H3>"
		+ "  <BR><FONT SIZE=\"+2\">" + (String)request.getParameter("nazwisko")
		+ "  to "+ (String)request.getParameter("opinia")
	+ "  </FONT><BR>"
	+ "  <FONT SIZE=\"-2\">Zapraszam na:"
	+ "  <A HREF=\"http://biont.eti.pg.gda.pl/java.htm\">J\u0119zyk JAVA</a><br>"
	+ "  </FONT>"
	+ " </CENTER>";
		wiad+=stopka();

		pw.print(wiad);
		pw.close();
	}

}// koniec public class OgloszenieServlet

//formularz.html:

<html>
<head>
<title> Serwlety: wywołanie serwletu dla formularza w kodzie HTML</title>
</head>
<body>
<H1>Strona demonstracyjna</H1>
<H2>Pokaz wywołania serwletu dla formularza w kodzie HTML</H2>
<form  action="/servlet/OgloszenieServlet" method="get">
<p> Wykładowaca
	<select name="nazwisko">
		<option value="Kowalski" selected>Kowalski</option>
		<option value="Nowak" selected>Nowak</option>
		<option value="Misiak" selected>Misiak</option>
	</select>
	to:
	<input type="text" name="opinia">
</p>
<p>
	<input type="submit" name="publikuj" value="Publikuj">

</p>
</form>
</body>
```

</html>

W wyniku podania parametrów w formularzu możliwe jest wygenerowanie następującej strony:

Strona testowa serwisu nauki JAVY
[Lokalny czas to: Wed May 10 11:38:08 CEST 2000]
Miło jest mi zakomunikować, że:

Nowak to dobry nauczyciel
Zapraszam na: Język JAVA

W prezentowanym przykładzie zastosowano formularz w kodzie HTML zawierający:
- pole wyboru o nazwie „nazwisko" umożliwiające wybór jednego z trzech nazwisk
- pole tekstowe o nazwie „opinia" umożliwiające wpisanie dowolnego tekstu opinii o wykładowcy
- pole wykonania operacji o nazwie „publikuj" wywołujące zadaną akcję, w tym przypadku żądanie „GET" obsługiwane przez /servlet/OgloszenieServlet.

W obsłudze żądania „GET" realizowanego przez serwlet pobierane są informacje o wartościach kolejnych dwóch parametrów:

(String)request.getParameter("nazwisko")
(String)request.getParameter("opinia")

które są wyświetlane w ramach tworzonej strony HTML.

Oczywiście wywołanie serwletu jako programu obsługującego określoną operację może pochodzić z innych źródeł, np. z apletu. Łatwo można również stworzyć serwlet obsługujący bazę danych dla potrzeb serwisu WWW (inicjowanie połączenia w metodzie init() serwletu; polecenia SQL w ramach formularza lub apletu; zamknięcie połączenia w ramach metody destroy()) łącząc go dodatkowo ze zdalnym wykonywaniem metod (na serwerze aplikacji obsługującej bazę danych) poprzez RMI.

8.3.6 Bezpieczeństwo serwletów

Serwlety mają dostęp do informacji o kliencie i protokole, natomiast informacje i dostęp do sieci oraz plików mają takie na ile umożliwia im do system bezpieczeństwa (Security Manager). Działalność systemu bezpieczeństwa w stosunku do servletów jest analogiczna jak dla apletów, co jest omówione w ostatnim rozdziale tego podręcznika. Warto jednak wspomnieć, że wprawdzie uprawnione serwlety nie spowodują błędów dostępu do pamięci (co jest cechą programów w Javie) lecz mogą wywołać polecenie System.exit(), co może prowadzić do zakończenia pracy serwera. Innym elementem zabezpieczania serwera przez niewłaściwym działaniem niektórych serwletów jest stosowanie list dostępu (ACL – access code lists) wskazujących kto i co może na danym serwerze wywołać (uruchomić). Można w ten sposób ograniczyć wykonywanie niektórych serwletów.

8.4 Zdalne wywoływanie metod - RMI

Programowanie sieciowe skupia się zazwyczaj na tworzeniu aplikacji w dwóch zasadniczych grupach:
- aplikacji wymiany plików (np. obsługa protokołów FTP, HTTP, NFS, SMTP)
- aplikacji wywoływanych i wykonywanych zdalnie (np. obsługa baz danych, telnet, RPC, CGI).

Zdalne wykonywanie zadań jest istotnym elementem w rozwiązywaniu zadań poprzez programy działające równolegle (oddzielne procesy na oddzielnych procesorach) w środowisku rozproszonym (oddzielne procesory są elementami różnych maszyn współpracujących ze sobą poprzez sieć komputerową). Oczywiście przetwarzanie zadań z wykorzystaniem zdalnych procedur nie musi mieć charakteru przetwarzania równoległego, lecz może występować jedynie w formie rozproszonej (różne zadania są kolejno wykonywane przez różne jednostki). Ogólnie zdalne wykonywanie zadań polega:
- wysłaniu przez klienta żądania wykonania procedury (wraz z podaniem odpowiednich argumentów) do serwera aplikacji,
- wykonaniu procedury przez serwer,
- przesłanie przez serwer wyniku działania procedury do klienta
- wykorzystanie przez klienta otrzymanego wyniku działania zdalnej procedury do dalszych zadań.

Jednym z pierwszych rozwiązań tego typu była opracowana przez Suna technologia zwana RPC – Remote Procedure Call (zdalne wywoływanie procedur). W RPC zastosowano podejście strukturalne (niezależne procedury) z jednoczesną realizacją żądań niezależną od języka programowania i od typu platformy. Uniwersalność RPC wymaga jednak realizacji wielu dodatkowych zadań jak np. konwersja typów danych (różne rozmiary liczb całkowitych) argumentów; konwersja reprezentacji typów danych (różny zapis liczb całkowitych – little endian, big endian). Ponieważ RPC stosuje podejście strukturalne nie można przesłać jako argumentów złożonych typów danych takich jak dla klas obiektów. Ze względu na wymienione wady oraz ze względu na powstanie nowej platformy Javy, Sun stworzył kolejną technologię zdalnego wywoływania procedur o nazwie RMI – Remote Method Invocation (zdalne wywoływanie metod). Ponieważ RMI stosuje podejście obiektowe zmieniono w nazwie technologii słowo procedura (procedure) na metoda (method) wskazując w ten sposób na możliwość zdalnego wykonywania zadań przypisanych dla obiektu. RMI stanowi więc dobre rozwiązanie do zdalnego wykonywania zadań realizowanych w Javie pomiędzy platformami różnego typu. RMI ogranicza się zatem jedynie do metod tworzonych w Javie.

8.4.1 Typy obiektów i relacje pomiędzy nimi w RMI

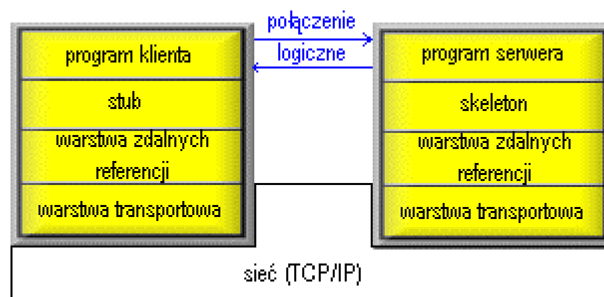
W zdalnych wykonywaniu zadań wyróżnia się dwa typy obiektów: obiekty lokalne (klient) i obiekty zdalne (serwer). Każdy obiekt zdalny musi implementować specjalny interfejs, określający wszystkie metody możliwe do wykonywania przez klienta. Interfejs te musi dziedziczyć po zdefiniowanym w API interfejsie Remote. Interfejs Remote ma charakter znacznika zbioru metod zdalnych (podobnie jak interfejs Serializable określa obiekty, które mogą podlegać utrwalaniu np. poprzez zapis lub wymianę przez sieć). Oprogramowanie lokalne (klienta) może wywoływać

metody obiektu zdalnego tak, jakby były to metody lokalne. Mechanizm ten jest więc bardzo wygodny dla programisty. Pojawia się jednak pewien problem. Otóż jeśli obiekt jest przekazywany (argument) lub zwracany przez metodę zwracana jest w rzeczywistości referencja realizowana jako wskaźnik do pamięci organizowanej przez daną maszynę wirtualną. Wymiana obiektów, a więc referencji pomiędzy dwoma maszynami wirtualnymi przysparza pewien problem: referencja obiektu A, np. 12345 w danej maszynie wirtualnej nie wskazuje tego samego obiektu na innej maszynie wirtualnej. Rozwiązanie tego problemu może być wykonane na dwa sposoby. Po pierwsze można przekazać zdalną referencję obiektu wskazującą na pamięć zdalnej maszyny wirtualnej. Oznacza to, że jeżeli klient przekazuje jako argument obiekt zdalny (realizowany na maszynie zdalnej) to w rzeczywistości przekazuje jego zdalną referencję czyli zdalny obiekt nigdy nie opuszcza swojej maszyny wirtualnej. Jeżeli, co stanowi drugie rozwiązanie, natomiast klient przekazuje swój obiekt w wywołaniu metody zdalnej tworzy jego kopię, która jest przesyłana. W celu przesłania kopii obiektu lokalnego konieczna jest jego konwersja do takiej formy zbioru bajtów, aby była odtwarzalna po stronie serwera aplikacji. Konieczne jest zatem, dostępne w Javie, zastosowanie mechanizmu serializacji (klasy obiektów muszą implementować interfejs Serializable). Prawie wszystkie obiekty mogą być w ten sposób podlegać konwersji, z wyjątkiem tych, które mają charakter lokalny (np. odnoszą się do plików lokalnych). Przesyłanie obiektów poprzez ich kopie wykonywane jest również dla tych obiektów po stronie serwera aplikacji (zdalnej), które nie implementują interfejsu Remote, czyli są obiektami lokalnymi serwera lecz nie stanowią w świetle RMI obiektów zdalnych. Podsumowując, możliwe są następujące formy przekazywania argumentów i wyników działania metod:

- przez wartość: w komunikacji dwustronnej poprzez wartość przekazywane są proste typy danych jak: int, boolean, iitp),
- przez referencję zdalną: w komunikacji klient -> serwer przekazywana jest zdalna referencja zdalnego obiektu,
- przez kopię: - czyli również przez wartość – w komunikacji dwustronnej poprzez kopie przekazywane są obiekty inne niż zdalne (tzn. te dziedziczące po interfejsie Remote).

8.4.2 Komunikacja w procesie zdalnego wykonywania metod.

W celu analizy procesu komunikacji pomiędzy klientem a serwerem aplikacji posłużmy się modelem zaprezentowanym na rysunku:



Rysunek 8.1. Model warstwowy procesu komunikacji w RMI

Z prezentowanego modelu widać, że logicznie oprogramowanie klienta pracuje tak, jakby było bezpośrednio powiązane z programami serwera. W rzeczywistości

komunikacja jest bardziej złożona. Klient wywołując metody zdalne używa specjalnego obiektu – stub – będącego odpowiednikiem, lokalnym reprezentantem obiektu zdalnego. Stub implementuje interfejs obiektu zdalnego dzięki czemu posiada pełen opis (sygnatury) metod tego obiektu. Obiekt stub stanowi więc namiastkę (stub jest często tłumaczony przez słowo namiastka) obiektu zdalnego po stronie lokalnej. Wywoływane przez klienta metody zdalne są bezpośrednio odnoszone do obiektu stub, a nie do obiektu zdalnego. Stub przekazuje dalej wywołanie do warstwy zdalnych referencji. Głównym zadaniem tej warstwy jest odniesienie lokalnych referencji obiektu stub do referencji obiektu zdalnego po stronie serwera. Kolejnym krokiem jest przesłanie danych wywołania metody do warstwy transportowej. Warstwa ta jest odpowiedzialna za przesyłanie danych przez sieć i realizuje: ustalanie połączeń, śledzenie obiektów zdalnych, zarządzanie połączeniem, itp. Po stronie serwera warstwa transportowa realizuje nasłuch danych przenoszących żądanie wykonania metody zdalnej. Po uzyskaniu referencji obiektu zdalnego, będącego teraz obiektem lokalnym serwera dane przekazywane są do kolejnej warstwy o nazwie skeleton. Skeleton (namiastka) jest odpowiednikiem stub' a po stronie klienta. Warstwa ta odpowiada za przypisanie do wywoływanych metod przez stub ich implementacji realizowanych w oprogramowaniu serwera. Po wykonaniu zadania przez serwer ewentualne wyniki mogą być zwrotnie transportowane do klienta (jako: wartości, kopie obiektów lub obiekty zdalne).
Zgodnie z zaprezentowanym modelem ważną rolę w procesie wymiany parametrów odgrywają warstwy stub oraz skeleton. Nazwy stub oraz skeleton są tu przyjęte z technologii RPC. Warstwy pośredniczą i sterują procesem wykonywania zdalnych procedur. Jakkolwiek w wersjach Javy od JDK1.2 nie jest już wymagana oddzielna realizacja (poza oprogramowaniem serwera) warstwy skeleton. Fizyczna realizacja oprogramowania warstw stub'a i skeleton'u powstaje w wyniku generacji dla nich kodu pośredniego Javy na podstawie klasy obiektu zdalnego. Generacja ta wykonywana jest za pomocą dodatkowego kompilatora o nazwie rmic (RMI compiler).

Powyższe rozważania nie uwzględniają problemu adresowania zdalnego serwera aplikacji. Możliwe jest przecież, że w sieci istnieć będą tysiące komputerów serwujących setki obiektów zdalnych (ich metod) każdy. Jak zatem zaadresować po stronie klienta konkretny obiekt zdalny na konkretnym serwerze? Otóż w tym celu tworzony jest mechanizm rejestracji obiektów zdalnych. RMI wykorzystuje w tym celu interfejs java.rmi.registry.Registry oraz klasę java.rmi.registry.LocateRegistry do obsługi (rejestracja obiektów i przeszukiwanie rejestru) rejestru obiektów zdalnych poprzez przypisane im nazwy. Rejestr pełni zatem rolę prostej bazy danych wiążącej ze sobą nazwy obiektów z ich realizacją. Przykładową realizacją rejestru (implementująca interfejs Registry) jest program dostarczany przez Suna – rmiregistry, który po wywołaniu pracuje na danym komputerze na domyślnym porcie dla RMI zdefiniowanym w interfejsie Registry:

public static final int REGISTRY_PORT = 1099.

Oczywiście można podać jako argument do programu rmiregistry inny numer portu, należy jednak pamiętać o konsekwencjach tego posunięcia, czyli o właściwym adresowaniu rejestru z poziomu programów go użytkujących. Lokalizacja rejestru jest obsługiwana przez metody klasy LocateRegistry. Użytkowanie rejestru odbywa się poprzez zastosowanie metod klasy java.rmi.Naming. Metody tej klasy wykonując

operacje na rejestrze odwołując się pośrednio przy pobieraniu referencji do rejestru do metody getRegistry() klasy LocateRegistry. Występujące w klasie Naming metody są realizacją deklarowanych w interfejsie Registry metod, przeznaczoną do obsługi adresów podawanych w formie URL:

- lookup() – zwraca obiekt zdalny związany z podaną nazwą,
- bind() – wiąże obiekt zdalny z podaną nazwą – jeśli podana nazwa już występuje zwracany jest wyjątek klasy, AlreadyBoundExcepton,
- rebind() – wiąże obiekt zdalny z podaną nazwą – jeśli podana nazwa już występuje przypisany do niej obiekt jest zastępowany nowym,
- unbind() – usuwa powiązanie nazwy z obiektem zdalnym w rejestrze,
- list() – zwraca listę nazw występujących w rejestrze (lista obiektów typu String).

Podawana nazwa przyjmuje format URL:

//host:port/nazwa

gdzie:

host – nazwa lub adres komputera gdzie uruchomiony jest rejestr,
port – numer portu (domyślnie 1099),
nazwa – nazwa przyjęta dla obiektu zdalnego w rejestrze,

i jest reprezentowana jako obiekt łańcucha znaków – String.

Podsumowując przeprowadzone rozważania można przedstawić uproszczony model przepływu danych:

1. Serwer rejestruje obiekt zdalny w rejestrze,
2. Klient przeszukuje rejestr w celu uzyskania obiektu zdalnego (lookup()),
3. Klient żąda przesłania klasy stub'u obiektu zdalnego (dynamiczne ładowanie klasy – RMIClassLoader),
4. Klient poprzez stub zapisuje i transmituje parametry z żądaniem wykonania metody obiektu zdalnego
5. Serwer poprzez skeleton realizuje wykonanie metod podawanych przez stub
6. Serwer przesyła wynik działania metody zdalnej do klienta.

8.4.3 Konstrukcja obiektu zdalnego – oprogramowanie serwera.

Tworząc obiekt zdalny należy najpierw zdefiniować interfejs dziedziczący po interfejsie Remote zawierający deklarację wszystkich metod zdalnych. Przykładowo interfejs taki może wyglądać następująco:

Przykład 8.17:

```
//Srednia.java:

import java.rmi.*;

public interface Srednia extends Remote{
	public Double policzSrednia(int i) throws RemoteException;
}
```

Każda metoda zdalna interfejsu musi mieć deklarację wyjątku RemoteException. Mając zdefiniowany interfejs można stworzyć klasę obiektu zdalnego:

Przykład 8.18:

```
//ObliczeniaSrednia.java:

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

public class ObliczeniaSrednia extends UnicastRemoteObject implements Srednia{

        public ObliczeniaSrednia() throws RemoteException{
                super();
        }
        public Double policzSrednia(int i) throws RemoteException{
                        double s=0.0;
                        for (int j=0;j<i;j++){
                                s=s+(int)j;
                        }
                        return (new Double(s/i));
        }
        public static void main(String args[]){
                        try{
                                ObliczeniaSrednia ob = new ObliczeniaSrednia();
                                Naming.rebind("srednia", ob);
                                System.out.println("Serwer gotowy");
                        }catch (RemoteException re){
                                System.out.println("Wyjątek: "+re);
                        }catch (MalformedURLException ue){
                                System.out.println("Wyjątek adresowania: "+ue);
                }
        }

}// koniec public class ObliczeniaSrednia
```

W powyższym kodzie zdefiniowano klasę ObliczeniaSrednia dla obiektów zdalnych o definicji metody policzSrednia podanej w interfejsie Srednia. Definiowana klasa dziedziczy dodatkowo po klasie UnicastRemoteObject. Klasa ta pochodzi z pakietu java.rmi.server w którym zdefiniowano m.in. klasy takie jak:

- RemoteObject – klasa abstrakcyjna dziedzicząca po java.lang.Object definiująca zachowanie obiektów zdalnych na podobieństwo obiektów zwykłych dziedziczących po java.lang.Object. Wykorzystanie tej klasy jest konieczne ponieważ obiekt zdalny jest definiowany tylko przez te metody podawane w interfejsie dziedziczącym po Remote. Niezbędne jest więc zdefiniowanie dla niego metod takich jak np. equals(),
- RemoteServer – klasa abstrakcyjna dziedzicząca po RemoteObject, umożliwiająca obsługę zdalnych referencji, np. eksport obiektów zdalnych,

- UnicastRemoteObject – klasa dziedzicząca po RemoteServer, realizująca zadania klas abstrakcyjnych, z których dziedziczy.

Stosowanie UnicastRemoteObject nie jest konieczne, lecz wówczas trzeba stworzyć własną implementację usług przez nią dostarczanych.

W ciele metody głównej omawianej aplikacji przykładowej zawarto w bloku obsługi wyjątków trzy polecenia:
- stworzenie obiektu danej klasy:
  ObliczeniaSrednia ob = new ObliczeniaSrednia();
- stworzenie nazwy dla obiektu i jego rejestracja w rejestrze
  Naming.rebind(”srednia”, ob);
- wydruk o gotowości serwera do pracy
  System.out.println(”Serwer gotowy”);

Tak przygotowany kod można następnie skompilować:

javac –g Srenia.java
javac –g ObliczneniaSrednia.java

Po kompilacji kodu konieczne jest stworzenie klasy stub'u i ewentualnie skeleton'u. Do tego konieczne jest zastosowanie kompilatora rmic z podaniem jako argumentu pełnej nazwy (całej ścieżki jeśli dana lokalizacja nie występuje w ustawieniach zmiennej środowiska - CLASSPATH) pliku kodu pośredniego serwera, np.:

rmic –classpath www/classes ObliczeniaSrednia

gdzie:
 www/classes to odpowiednie podkatalogi z kodem wykorzystywanych klas i interfejsów.

W ten sposób wygenerowany zostanie kod klas dla stub'a i skeleton'a:
ObliczeniaSrednia_Stub.class i ObliczeniaSrednia_Skel.class.

Oczywiście ze względu na funkcję oprogramowania klienta kody te muszą być umieszczone w dostępnym sieciowo katalogu.

8.4.4 Oprogramowanie klienta.

Zanim klient będzie mógł wywołać zdalną metodę musi pobrać obiekt zdalny, dla którego tą metodę zdefiniowano. Wykonywane jest to poprzez zastosowanie implementacji metody lookup(), np. w klasie java.rmi.Naming:

Object o = Naming.lookup(”rmi://medvis.eti.pg.gda.pl/srednia”),

gdzie:
 rmi – to nazwa protokołu komunikacji,
 medvis.eti.pg.gda.pl - adres serwera na którym pracuje rejestr,
 srednia - nazwa przywiązana do obiektu zdalnego przez serwer.

Uzyskiwana w ten sposób zdalna referencja (stub) wskazuje na obiekt klasy Object, którego typ trzeba rzutować na typ danego interfejsu obiektu zdalnego:

Srednia s = (Srednia) o;

lub razem:

Srednia s = (Srednia) Naming.lookup(”rmi://medvis.eti.pg.gda.pl/srednia”).

Uzyskując w ten sposób referencję obiektu zdalnego można wywołać zdefiniowaną metodę zdalną. Przykładowy program klienta przedstawiono poniżej:

Przykład 8.19:

```
//SredniaKlient.java:

import java.rmi.*;
import java.util.*;

public class SredniaKlient{

        public static void main(String args[]){
                        if (System.getSecurityManager()==null)
                        System.setSecurityManager(new RMISecurityManager());

                        try{
                        Random r = new Random();
                        int n = r.nextInt(50);
                        Srednia s = (Srednia) Naming.lookup("rmi://medvis.eti.pg.gda.pl/srednia");
                        Double d =  s.policzSrednia(n);
                        System.out.println("Srednia z kolejnych"+ n+" wartości wynosi: "+d);
                }catch (Exception e){
                        System.out.println("Wystąpił wyjątek: "+ e);
                }
        }
}// koniec public class SredniaKlient
```

Na początku metody głównej aplikacji klienta zawarto test wykrywający działanie systemu zarządzania bezpieczeństwem platformy Javy. Działanie systemu bezpieczeństwa jest tu konieczne ze względu na określenie możliwości działania ładowanego przez klienta kodu. Wszystkie programy używające RMI muszą używać systemu zarządzania bezpieczeństwem inaczej protokół RMI nie przekopiuje wymaganych klas z serwera. Dla RMI stworzono oddzielną klasę systemu zarządzania bezpieczeństwem, która prawie w całości jest identyczna z klasą SecurityManager zmieniając definicję tylko jednej z metod checkPackageAccess().

Tak przygotowany kod należy skompilować:

javac –g SredniaKlient.java

8.4.5 Uruchamianie systemu.

Zanim będzie można skorzystać z klienta należy uruchomić rejestr. W tym celu należy wywołać program rmiregistry bez numeru portu, ponieważ w rozpatrywanych przykładach przyjęty został port domyślny (1099). Dla właściwego działania systemu przed uruchomieniem rejestru należy wyeliminować z ustawień środowiska terminala, w którym będzie wywoływany program rmiregistry, ścieżkę dostępu (CLASSPATH) do stworzonych właśnie klas serwera. Dzięki temu możliwe będzie podanie przy wywołaniu programu serwera katalogu bazowego wykorzystywanych klas (CODEBASE). W przeciwnym przypadku „widziane" przez program rmiregistry klasy uniemożliwiłyby ładowanie klas z katalogu podanego w opcji CODEBASE wywołania programu serwera. Z tego powodu przed można przed wywołaniem programu rmiregistry podać polecenie w danym shellu terminala Unixa:

unsetenv CLASSPATH (lub dla Win32 unset CLASSPATH).

Następnie można wywołać program rmiregistry poprzez podane jego nazwy:

rmiregistry

Po uruchomieniu rejestru czas na rozpoczęcie pracy przez serwer. Uruchomienie aplikacji serwera powinno zawierać dodatkowe opcje oznaczające własności:
CODEBASE - podanie katalogu klas serwera
HOSTNAME – podanie nazwy serwera.
Wywołanie programu serwera ma więc postać:

java –Djava.rmi.server.codebase=http://medvis.eti.pg.gda.pl/classes/
-Djava.rmi.server.hostname=medvis.eti.pg.gda.pl
ObliczeniaSrednia

co oznacza, że wymagane klasy znajdują się na serwerze medvis.eti.pg.gda.pl w katalogu classes. W przypadku braku dostępu programisty do głównego katalogu serwisu WWW można klasy umieścić w swoim katalogu WWW, np. zależnie od ustawień serwera WWW: www lub public_html; i wówczas katalog będzie dostępny pod adresem: ~username/classes/, gdzie: username to nazwa użytkownika w danym systemie.

Po uruchomieniu serwera można oczywiście wywołać klienta, zakładając że dla maszyny wirtualnej klienta widziane są klasy interfejsu obiektu zdalnego oraz klasa klienta:

java SredniaKlient

Przy standardowych ustawieniach środowiska bezpieczeństwa dla maszyny wirtualnej klienta jako efekt działania uruchamianej aplikacji otrzymamy komunikat:

Wystąpił wyjątek: java.security.AccessControlException: access denied (java.net.SocketPermission medvis.eti.pg.gda.pl resolve)

Jest to związane z tym, że dla klienta uruchomiany jest system zarządzania bezpieczeństwem, który sprawdza pliki zasad bezpieczeństwa. Standardowo skonfigurowany plik java.policy nie zawiera właściwych uprawnień dla ustalania połączeń sieciowych (potrzebnych w RMI do wywołania metod i skopiowania stub'a). Informacje na temat ustawień plików zasad bezpieczeństwa podano w ostatnim rozdziale tego podręcznika. W celu przyzwolenia na wykonywanie połączeń na określonych portach konieczne jest nadanie następujących uprawnień:

```
//moje_policy:

grant {
  permission java.net.SocketPermission "*:1024-65535", "connect, accept";
  permission java.net.SocketPermission "*:80", "connect";
};
```

- ustawienia dla portów 1024-65535 są potrzebne ze względu na port rmi – 1099
- ustawienia dla portu 80 są potrzebne ze względu na połączenie z miejscem lokalizacji klas stub'a czyli z serwerem WWW działającym na porcie 80, w rozpatrywanym przypadku jest to medvis.eti.pg.gda.pl.

Uprawnienia te mogą być zapisane w pliku np. moje_policy i wówczas wywołanie programu klienta będzie następujące:

java -Djava.security.policy=moje_policy SredniaKlient

Efektem czego będzie komunikat o przykładowej następującej treści:

Srednia z kolejnych 34 wartości wynosi: 16.5