Wykład 8

URL

Interfejs gniazd

Transmisja SSL

protokół JNLP

URL

`URL` – *Unified Resource Locator* jest podstawową klasą identyfikującą zasoby w internecie.

> Klasa URL znajduje się w pakiecie java.net.

```
import java.net.*;
import java.io.*;
public class URLExample {
    public static void main(String[] args) throws Exception {
        URL url = new URL("http://www.google.pl/");

        BufferedReader in = new BufferedReader(new InputStreamReader(
                                                    url.openStream()));
        String s;
        while ((s = in.readLine()) != null)
            System.out.println(s);
        in.close();
    }
}
```

`url.openStream()` — zostaje zwrócony strumień typu *InputStream*, dzięki któremu możemy odczytać dane znajdujące się pod tym adresem (tutaj: http://www.google.pl/)

`in.readLine()` — odczytuje zawartość *"in"* linia po linii

URLConnection

URLConnection — zawiera metody umożliwiające nawiązanie połączenia
z zasobem reprezentowanym przez URL.

```
import java.net.*;
import java.io.*;
public class URLConnectionExample {
    public static void main(String[] args) throws Exception {
        URL url = new URL("http://www.google.pl/");
        URLConnection con = url.openConnection();
        BufferedReader in = new BufferedReader(new InputStreamReader(
                                                con.getInputStream()));
        String s;
        while ((s = in.readLine()) != null)
            System.out.println(s);
        in.close();
    }
}
```

Dwie różnice w strukturze:

1. Tworzymy obiekt URLConnection z ustalonego wcześniej adresu URL

```
URLConnection con = url.openConnection();
```

2. Korzystamy z `con.getInputStream()` zamiast `url.openStream()`

Jak widzimy programy są praktycznie identyczne, więc

Po co używać `URLConnection` zamiast `URL` ?

> **URLConnection oferuje także zapis do wskazanego zasobu** przez obiekt *URL* (serwer musi nam na to pozwolić).

```
import java.io.*;
import java.net.*;
public class URLConnectionWriter {
    public static void main(String[] args) throws Exception {
        URL url = new URL(args[0]);
        URLConnection con = url.openConnection();
        con.setDoOutput(true);
        OutputStreamWriter out = new OutputStreamWriter(
                                                con.getOutputStream());

        for (int i = 1; i < args.length; i++)
            out.write(args[i]);
        out.close();
    }
}
```

`setDoOutput(true)` – w klasie *URLConnection*, umożliwia również wysyłanie danych przez obiekt *con*

URL (Uniform Resource Locator)

1. **Zastosowanie**
   Klasa *URL* służy do reprezentowania adresu URL, czyli jednoznacznego identyfikatora zasobu dostępnego w internecie.
2. **Funkcje**:
   - Parsowanie adresów URL.
   - Umożliwianie dostępu do poszczególnych składowych adresu URL, takich jak protokół, host, port, ścieżka, itp.
   - Tworzenie obiektu *URL* na podstawie łańcucha znaków reprezentującego adres URL.

URLConnection

1. **Zastosowanie**
   Klasa *URLConnection* jest używana do nawiązywania połączeń z zasobami dostępnymi pod danym adresem URL.
2. **Funkcje**:
   - Umożliwianie otwierania połączenia do zasobu zdalnego.
   - Pobieranie strumieni wejściowych i wyjściowych, co umożliwia odczyt i zapis danych do zdalnego zasobu.
   - Manipulowanie nagłówkami żądania HTTP (jeśli połączenie opiera się na tym protokole).
   - Udostępnianie informacji o połączeniu, takich jak długość zawartości, itp.

Gdzie może być wykorzystana taka funkcja zapisu?

Protokół – sposób porozumiewania się komputera z różnymi serwerami – danych, www, itp.

Zarówno w przypadku *URL* jak i *URLConnection* komunikacja odbywa się z wykorzystaniem odpowiedniego protokołu. Dokumentacja Javy gwarantuje standardowo obsługę następujących protokołów:

- `http` — serwer www, zwykły
- `https` — serwer www, szyfrowany
- `ftp` — usługa do transferu plików
- `file` — komunikacja z dyskiem twardym
- `jar` — komunikacja z archiwami jar

> Obsługa innych protokołów wymaga implementacji klasy **URLStreamHandler**

Zapisywanie danych do plików odbywa się poprzez `ftp` , działa to jak **repozytorium**, użytkownik może wrzucać tam swoje rzeczy, ktoś innym może pobrać. Za pomocą ścieżki wysyłamy, np. ftp://login:haslo@serwer:port/katalog/podkatalog/plik

Interfejs Gniazd

W Javie, interfejs gniazd (ang. socket) jest mechanizmem, który umożliwia komunikację pomiędzy różnymi aplikacjami działającymi na różnych maszynach w sieci. Gniazda umożliwiają przesyłanie danych między aplikacjami przez sieć, bez względu na to, czy są one uruchomione na tym samym komputerze czy na różnych maszynach.

Podstawowym celem interfejsu gniazd w Javie jest umożliwienie niskopoziomowej komunikacji między aplikacjami przy użyciu protokołów.

Najważniejsze klasy, umożliwiające komunikację poprzez sieć to:

`Socket` — klasa reprezentująca gniazdo służące do **nawiązywania połączenia, wysyłania I odbierania danych**
`ServerSocket` — klasa reprezentująca gniazdo **oczekujące na przychodzące żądania połączeń**

> Ponadto istnieją także gniazda `SSLSocket` I `SSLServerSocket` obsługujące komunikację szyfrowaną protokołem *SSL/TLS*

W tym internecie którego używamy, większość połączeń jest realizowana w schemacie *klient-serwer* (jest możliwe utworzenie opcji, gdy oba serwery pełnią funkcję serwera i klienta na raz, ale nie wprowadza to jakiś rewolucyjnych zmian).

Przykład Programu Klienta

Wysyła i odbiera dane od serwera.

`args[0]` — Podajemy adres serwera

`args[1]` — Numer portu na którym program na nasłuchiwać

```java
import java.io.*;
import java.net.Socket;
public class ClientExample {
    public static Socket sock;
    public static void main(String[] args) throws IOException{
        // tworzymy gniazdo i nawiazujemy polaczenie z komputerem
        // identyfikowanym przez adres args[0] na porcie args[1]
        sock = new Socket(args[0], Integer.valueOf(args[1]));

        // pobieramy strumienie zwiazane z gniazdem
        OutputStream os = sock.getOutputStream();
        InputStream is = sock.getInputStream();

        // tworzymy Reader na standardowym wejsciu (klawiaturze)
        BufferedReader br = new BufferedReader(new InputStreamReader(
                                                            System.in));
        // zmienne pomocnicze
        String sLine;
        byte[] bRes = new byte[100];

        // glowna petla programu, pobieramy dane z klawiatury
        while((sLine=br.readLine())!=null){
            // wysylamy je przez gniazdo
            os.write(sLine.getBytes());
            System.out.println("wyslalem: " + sLine);
            // odbieramy odpowiedz z serwera – to jest zle rozwiazanie
            // dobre rozwiązanie – odbieranie danych w osobnym wątku
            is.read(bRes);
            System.out.println("odebralem" + new String(bRes));
        }
        // zamykamy strumien i gniazdo
        br.close();
        sock.close();
    }
} // koniec programu
```

Zamykanie gniazda zamyka także strumienie oraz kończy połączenie!

> metoda `read()` wywołana na strumienia **wstrzymuje działanie programu do momentu, aż coś nie zostanie odebrane**. W przypadku, gdy serwer będzie chciał odpowiedzieć dopiero po kilku

> linijkach kodu, to nie będzie mieć takiej możliwości. (instrukcja blokująca). Dlatego warto takie działania robić w osobnych wątkach!

Osobny wątek do odbioru danych:

```
Thread t = new Thread(new Runnable(){
    public void run(){
        byte[] bRes = new byte[100];
        InputStream is;
        int l;
        try {
            is = sock.getInputStream();
            while(true){
                l = is.read(bRes);
                System.out.println("odebralem: " + new String(bRes,0,l));
            }
        } catch (IOException e) {
        e.printStackTrace();
        }
    }
});
t.start();
```

tutaj metoda `read()` też wstrzyma działanie wątku, ale ten wątek służy tylko do odbierania danych więc nie przeszkadza to głównemu wątkowi wykonywać dalej instrukcje main'a.

Przykład Programu Serwera Echo

Program serwera z którym klient może sobie porozmawiać. Serwer zwraca wysłany komunikat.

Serwer echo w Javie to prosta aplikacja, która nasłuchuje na określonym porcie i odsyła z powrotem wszystko, co otrzyma od klienta. Innymi słowy, serwer echo replikuje (odbija) dane, które otrzymuje od klienta.

```
import java.io.*;
import java.net.*;
public class ServerExample {
    // gniazdo oczekujace na polaczenia
    private static ServerSocket ss;
    public static void main(String[] args) throws IOException{

        // tworzymy gniazdo, ktore oczekuje na przychodzace polaczenia
        // na porcie przekazanym jako argument wywolania programu
        ss = new ServerSocket(Integer.valueOf(args[0]));
        // nieskonczona petla
        while(true){
            // akceptujemy polaczenie, dostajemy gniazdo do komunikacji
            // z klientem
            Socket s = ss.accept();
            // strumienie
            InputStream is = s.getInputStream();
            OutputStream os = s.getOutputStream();
            int b;
            // czytamy, piszemy na konsoli i odsylamy
            while((b=is.read())!=-1){
                System.out.print((char)b);
                os.write(b);
            }
            s.close();
        }
    }
}
```

`ss` – gniazdo oczekiwania na połączenie i odbieranie tych połączeń

W przypadku metody `ServerSocket` nie podajemy dwóch argumentów, tylko jeden, port. W nim będziemy oczekiwać na połączenie.

> Numery portów do 1024 są tzw. portami **well-known**, mają zarejestrowane numery do danych usług (np. numer 80 dla serwera www)

Powyżej to porty typu **register**, których numery trzeba zarezerwować dla jakiś własnych potrzeb.

`accept()` – oczekiwanie na komunikat od klienta. Program się zatrzymuje i czeka na klienta.

```
Tutaj nie ma problemu związanego z czekaniem w read(),
bo i tak musimy czekać na dane od klienta.
```

SSL w Javie

Java ma wbudowane mechanizmy związane z szyfrowaniem.

`Protokół SSL` — **umożliwia bezpieczną (szyfrowaną) transmisję danych poprzez niezabezpieczoną sieć**. Dodatkowo SSL umożliwia autoryzację stron komunikacji. W tym celu wykorzystywany jest mechanizm certyfikatów. Za transmisję z użyciem protokołu SSL odpowiedzialne są klasy zgrupowane w pakiecie *javax.net.SSL*.

> Implementacja SSH jest dostępna poprzez zewnętrzne biblioteki. Jedną z nich jest *jsch*

Serwer SSL

```
import javax.net.ssl.*;
import java.io.*;
public class EchoServer {
    public static void main(String[] args) throws IOException {
        SSLServerSocketFactory factory = (SSLServerSocketFactory)
        SSLServerSocketFactory.getDefault();
        SSLServerSocket ss = (SSLServerSocket) factory
                                               .createServerSocket(9999);
        SSLSocket s = (SSLSocket) ss.accept();
        BufferedReader br = new BufferedReader(new InputStreamReader(
                                               s.getInputStream()));
        String sTmp = null;
        while ((sTmp = br.readLine()) != null) {
            System.out.println(sTmp);
            System.out.flush();
        }
    }
}
```

Gniazdo serwerowe nie tworzymy przez konstruktory jak wcześniej, tylko wykorzystujemy wzorzec projektowy **factory**. Najpierw trzeba utworzyć `fabrykę gniazd` .

Tworzymy domyślną fabrykę poprzez funkcję `getDefault()` i jej instancję.

Gdy już mamy utworzoną fabrykę, wywołujemy na niej metodę `factory.createServerSocket(9999)`, która tworzy gniazdo serwerowe na podanym numerze portu.

> Później program wygląda analogicznie, z tego względu że klasa `SSLSocket` rozszerza klasę *ServerSocket* i *Socket*.

Klient SSL

```
import javax.net.ssl.*;
import java.io.*;
public class EchoClient {
    public static void main(String[] args) throws Exception {
        SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory
                                                            .getDefault();
        SSLSocket s = (SSLSocket) factory.createSocket("localhost", 9999);
        BufferedReader br = new BufferedReader(
                                    new InputStreamReader(System.in));
        OutputStreamWriter osw = new OutputStreamWriter(
                                                s.getOutputStream());
        BufferedWriter bw = new BufferedWriter(osw);
        String sTmp = null;
        while ((sTmp = br.readLine()) != null) {
            bw.write(sTmp + '\n');
            bw.flush();
        }
    }
}
```

Tworząc SSL Socket dodatkowo podajemy adres serwera oprócz numer portu.

`flush()` — wymusza natychmiastowe wysłanie danych

Jak działa SSL?

serwer echo szyfrowany zawsze musi się autoryzować przed klientem jakimś certyfikatem, ale klient nie musi.

Przykład:

> Wchodzimy na stronę banku i chcemy wiedzieć czy weszliśmy na faktyczną stronę a nie jakąś scamerską próbującą nas oskubać. Serwer banku musi się przed nami zautoryzować swoim certyfikatem abyśmy byli pewni czy jesteśmy na właściwej stronie.

Certyfikat można kupić lub wygenerować samemu do własnych prywatnych celów:

1. Pierwsza czynność to **wygenerowanie klucza**:

```
keytool -genkey -keystore mySrvKeystore -keyalg RSA
```

2. **Uruchomienie serwera**:

```
java -Djavax.net.ssl.keyStore=mySrvKeystore
-Djavax.net.ssl.keyStorePassword=123456 EchoServer
```

3. **Uruchomienie klienta**:

```
java -Djavax.net.ssl.trustStore=mySrvKeystore
-Djavax.net.ssl.trustStorePassword=123456 EchoClient
```

dajemy klientowi znać, że certyfikat, który utworzyliśmy, jest dla niego znany i bezpieczny (zaufany magazyn kluczy i haslo do niego)

4. Dodatkowe parametry wywołania pozwolą zobaczyć informacje związane z połączeniem SSL:

```
-Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
-Djavax.net.debug=ssl
```

`-keystore mySrvKeystore` — archiwum na klucze
`-keyalg RSA` — algorytm do generowania kluczy

`-Djavax.net.ssl.keyStore=mySrvKeystore` — podajemy klucz i z jakiego magazynu pochodzi
`-Djavax.net.ssl.keyStorePassword=123456` — hasło do magazynu kluczy

Domyślnie tylko jedna strona komunikacji (serwer) musi potwierdzać swoją tożsamość. Aby **wymusić autoryzację klienta należy użyć metod:**

- `setNeedClientAuth(true)`
- `setWantClientAuth(true)`

  wywołanych na rzecz obiektu *SSLServerSocket*.

> Jeśli chcemy aby żadna ze stron nie musiała potwierdzać swojej tożsamości musimy zmienić domyślne algorytmy kodowania. Najłatwiej zrobić to tworząc własne rozszerzenie klasy *SSLSocketFactory*.

Listę obsługiwanych i domyślnych algorytmów uzyskamy za pomocą metod:
getSuppotredCipherSuites() oraz ***getDefaultCipherSuites()***

Metody te są często używane w kontekście programowania aplikacji zabezpieczonych protokołem *SSL/TLS*, a dokładniej w kontekście klas z pakietu `javax.net.ssl.SSLSocket` i `javax.net.ssl.SSLServerSocket` .

`getSupportedCipherSuites()` — zwraca listę nazw obsługiwanych algorytmów szyfrowania dla danego gniazda SSL/TLS.

`getDefaultCipherSuites()` — zwraca listę domyślnie zalecanych (rekomendowanych) nazw algorytmów szyfrowania. Takie które są domyślnie zalecane, uważane za bezpieczne i efektywne.

Java Web Start

Jest to sposób na dystrybucję oprogramowania (udostępnienie programu klientom). Technologia Java Web Start jest stosowana do lokalnego uruchamiania programów w Javie umieszczonych w sieci.

Umożliwia on pobieranie i uruchamianie aplikacji Java z Internetu za pomocą jednego kliknięcia, bez konieczności instalacji.

JWS:

- jest w pełni niezależna od używanych przeglądarek internetowych
- umożliwia automatyczne pobranie właściwej wersji środowiska JRE
- pobierane są tylko pliki, które zostały zmienione
- obsługuje prawa dostępu do zasobów lokalnego komputera (dysk, sieć, itp.)
- do opisu zadania do uruchomienia wykorzystuje pliki `.jnlp` (Java Network Launch Protocol)

Protokół JNLP (Java Network Launch Protocol)

Plik typu `xml` zawiera podstawowe dane o programie

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp
    spec="1.0+"
    codebase="http://www.serwer.w.sieci.pl/katalog"
    href="plik_jws.jnlp">
    <information>
        <title>Nazwa programu</title>
        <vendor>Producent programu</vendor>
        <homepage href="http://www.strona.programu.pl"/>
        <description kind="short">Krotki opis programu</description>
        <icon kind="splash" href="kat/splashscreen.gif"/>
        <icon href="kat/ikona.gif"/>
        <offline-allowed/>
    </information>
    <security>
        <all-permissions/>
    </security>
    <resources>
        <j2se version="1.4+"/>
        <jar href="kat/archiwum1.jar"/>
        <jar href=" kat2/lib/biblioteka.jar"/>
    </resources>
    <application-desc main-class="pl.edu.uj.if.ExampleClass"/>
</jnlp>
```

`<all-permissions>` - dostęp wszędzie, dysk, dostęp do Internetu itp.

Plik jnlp umieszczamy na serwerze www w którym będzie nasz program

> Często należy skonfigurować odpowiadający mu typ mime:
> `application/x-java-jnlp-file JNLP`

Typ mime służy do określania z jakim typem dokumentu mamy do czynienia (np. podczas pobierania pdfa). Jeśli podczas pobierania widzimy binarny zapis danego pliku to oznacza to, że serwer www został źle skonfigurowany i nie poinformował przeglądarki o typie dokumentu, a ta potraktowała dokument typem domyślnym.

Uwaga!

Jeśli program wymaga wszystkich uprawnień od klienta (`<all-permissions/>`), to **musi on mieć podpisane wszystkie archiwa *jar* certyfikatem**. Robimy to za pomocą ***jar signer***. Musimy podać

klucz, którym są podpisane *jary* (klucze generujemy tak jak wyżej lub dostajemy od zaufanej instytucji).