semestr zimowy 2012/2013

Michał Cieśla (michal.ciesla@uj.edu.pl)

http://users.uj.edu.pl/~ciesla/

Konsultacje: środa 10-12, pokój 440a

JAVA

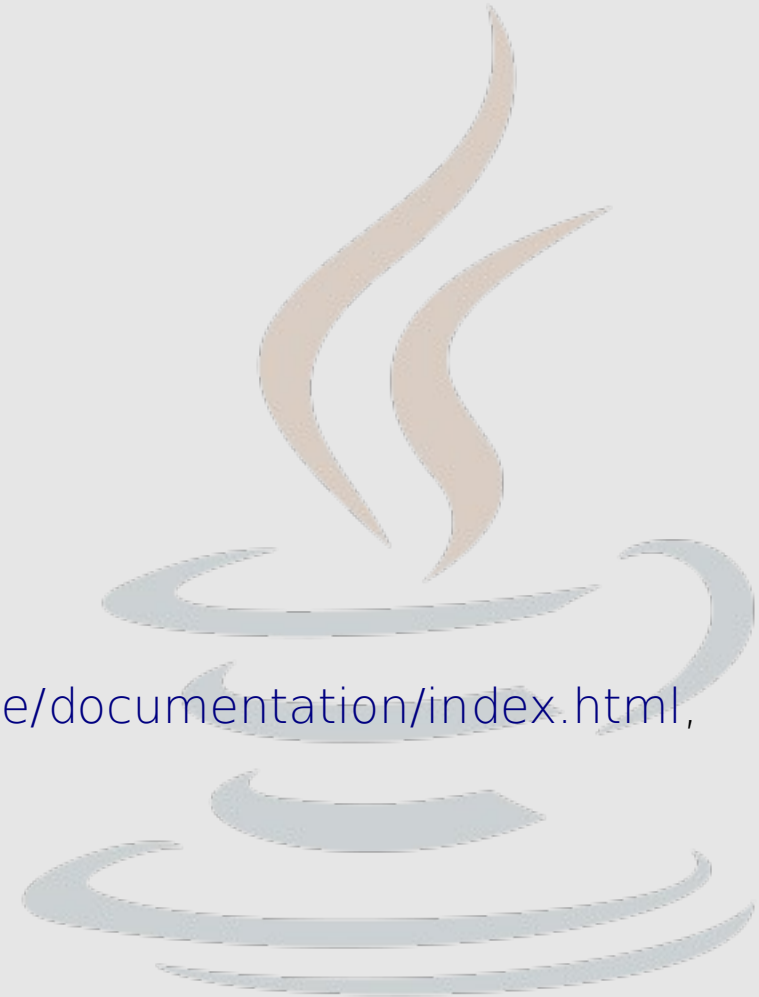
Java is one of several Indonesian islands that grow coffee. Originally, the term "Java coffee" or "Kopi jaw" identified the dark Arabica coffee specific to Java. In American slang, "Java" came to mean coffee in general .
By Tamasin Wedgwood, eHow Contributor

Read more: Why Is Coffee Called Java? | eHow.com

JĘZYK JAVA

ZAGADNIENIA:

- podstawy;
- przegląd biblioteki standardowej;
- Java w zastosowaniach.



LITERATURA:

http://www.oracle.com/technetwork/java/javase/documentation/index.html,

iTunesU,

http://www.google.com/,

Bruce Eckel, *Thinking in Java.*

PREREKWIZYTY

JDK – Java Development Kit

http://www.oracle.com/technetwork/java/javase/downloads/index.html

JRE – Java Runtime Environment

http://www.java.com

lub

http://www.oracle.com/technetwork/java/javase/downloads/index.html

HELLO WORLD

HelloWorldConsole.java

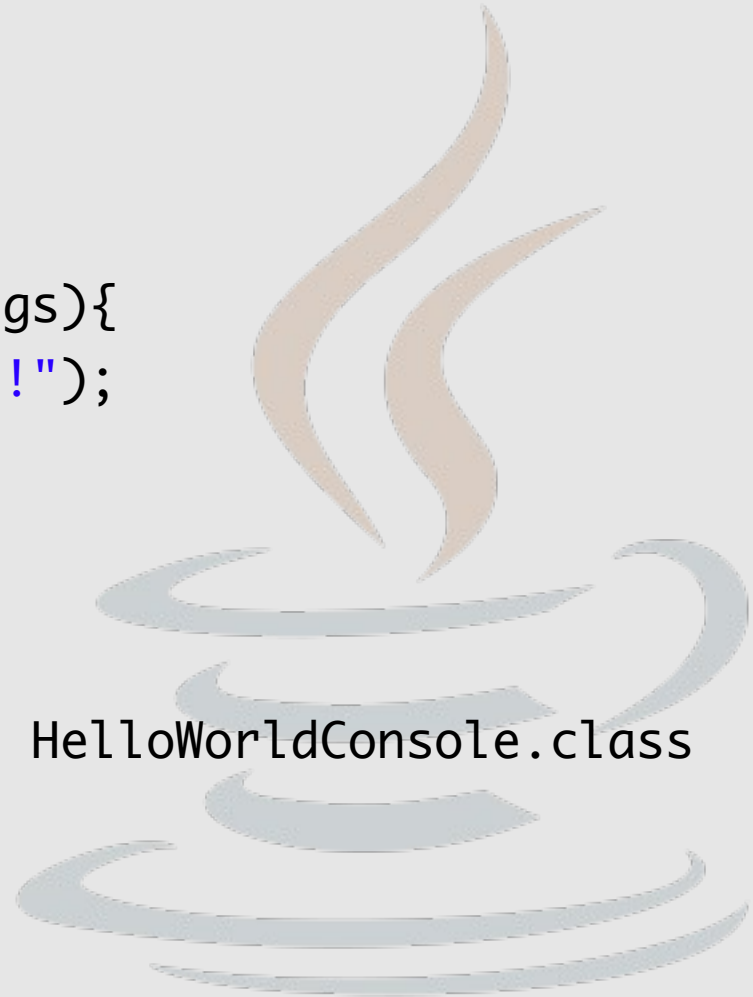
```
public class HelloWorldConsole {
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

KOMPILACJA:

```
javac HelloWorldConsole.java  ⟶  HelloWorldConsole.class
```

URUCHOMIENIE:

```
java HelloWorldConsole
```



HELLO WORLD: APPLET

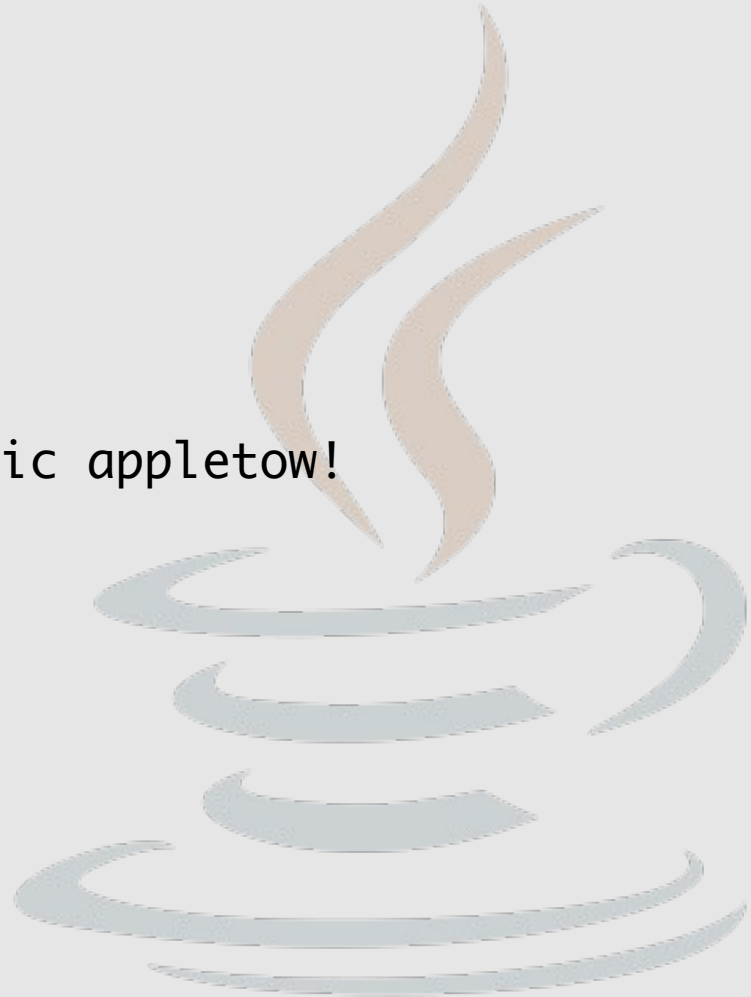
HelloWorldApplet.java

```
import javax.swing.JApplet;
import javax.swing.JLabel;

public class HelloWorldApplet extends JApplet {
    public void init(){
        this.setContentPane(new JLabel("Hello World"));
    }
}
```

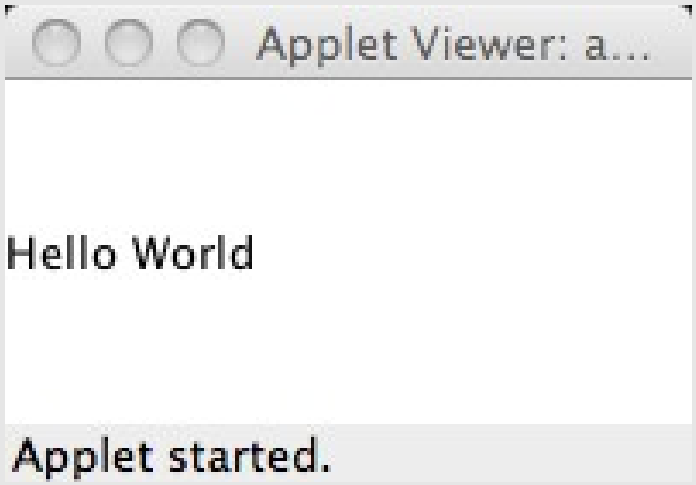
HELLO WORLD: APPLET

Applet.html

```
<html>
<body>
<applet code="HelloWorldApplet.class"
        width="200"
        height="100">
Twoja przegladarka nie potrafi wyswietlic appletow!
</applet>
</body>
</html>
```



HELLO WORLD: APPLET



TESTOWANIE:

```
appletviewer Applet.html
```

lub równoważnie:

```
java sun.applet.AppletViewer Applet.html
```

URUCHOMIENIE:

za pośrednictwem przeglądarki www.

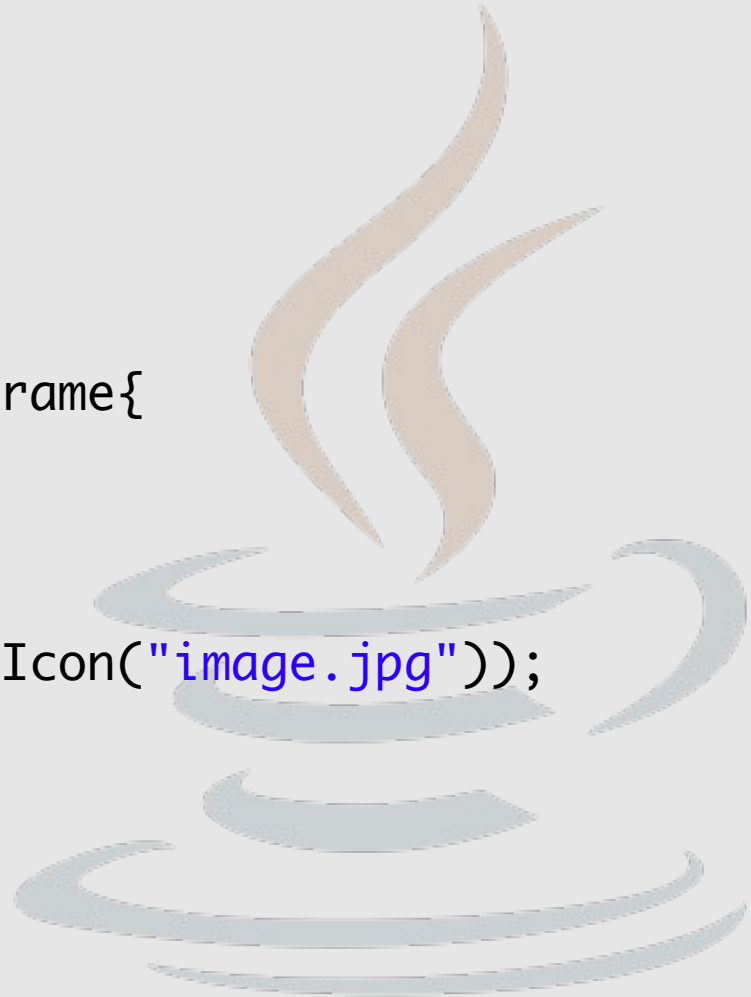
WARTO POCZYTAĆ (Java Web Start):

http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/developersguide/overview.html#jws

HELLO WORLD: OKIENKO

HelloWorldFrame.java

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorldFrame extends JFrame{

    public HelloWorldFrame(){
        super();
        JLabel l = new JLabel(new ImageIcon("image.jpg"));
        this.add(l);
    }
```



HELLO WORLD: OKIENKO

HelloWorldFrame.java (c.d.)

```
    public static void main(String[] args){
        HelloWorldFrame frame = new HelloWorldFrame();
        frame.setTitle("Pierwsze Okno");
        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

PODSTAWY: TYPY DANYCH

PODSTAWY JĘZYKA JAVA:

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html

PRYMITYWNE TYPY DANYCH:

- **byte** (8-bit), **short** (16-bit), **int** (32-bit), **long** (64-bit)
- **float** (32-bit), **double** (64-bit),
- **boolean** (1-bit) - flaga
- **char** (16-bit) – znak w unikodzie, np. **\u015b**

OBIEKTOWE TYPY DANYCH:

- **String**, **JLabel**,... (wszystko poza typami prymitywnymi).

PRZEPŁYW STEROWANIA

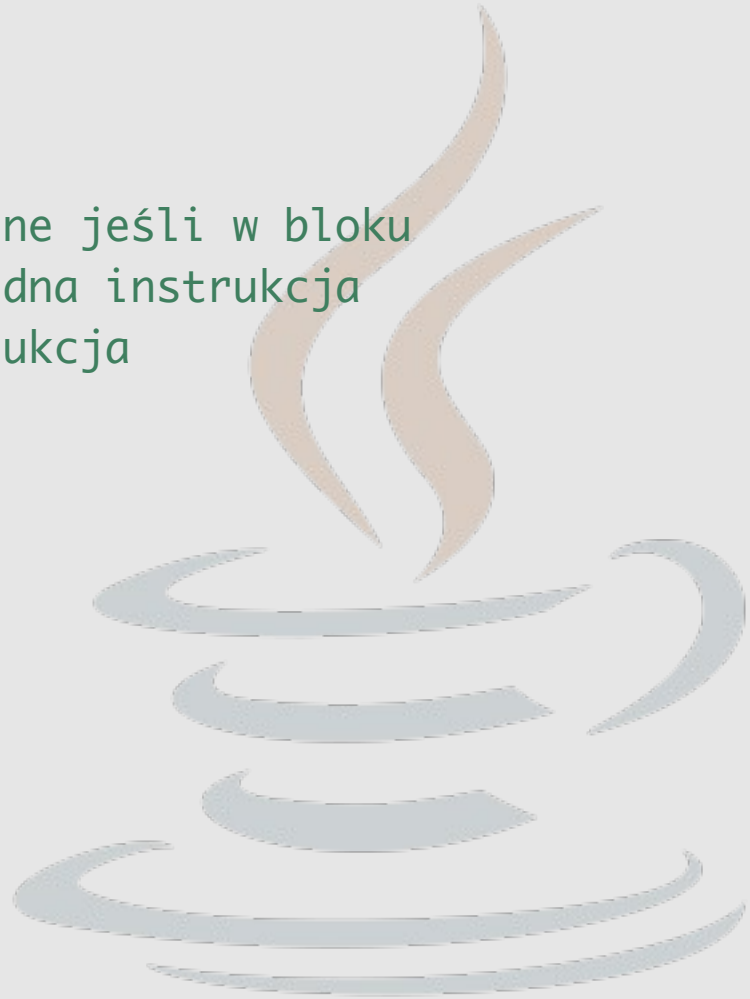
INSTRUKCJE WARUNKOWE

- if...then...else...

```
if(a>0){        // nawiasy klamrowe są wymagane jeśli w bloku
    return 1;   // znajduje się więcej niż jedna instrukcja
}else{          // tak samo jak w C/C++ instrukcja
    return -1;
}
```

- switch

```
switch (a){
    case 1: makeSomething(a);
            break;
    case 2: makeSomethingElse(a);
    default: a++;
}
```



PRZEPŁYW STEROWANIA

PĘTLE:

- for.

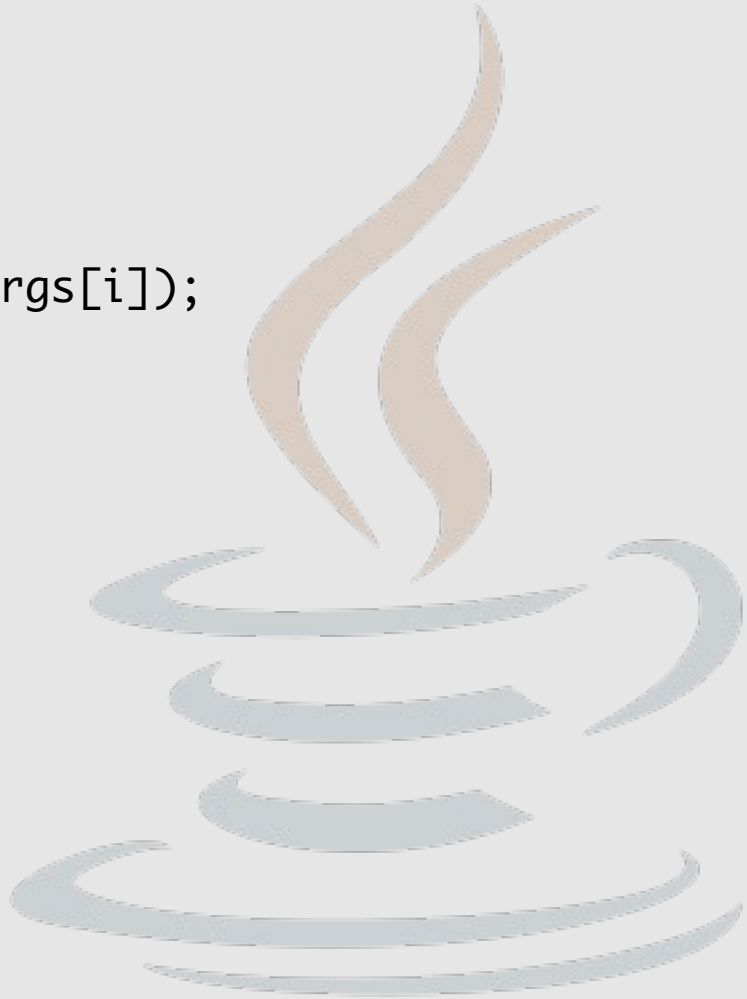
```
for(i=0; i<args.length; i++)
    System.out.printf(Locale.US, "%.2f\n", args[i]);
```

- while

```
String s="Ala";
while(s.length()<20)
    s = " " + s;
```

- do...while

```
do{
    String s = getValue();
}while(s!=null);
```



PRZEPŁYW STEROWANIA

ZABURZENIA PRZEPŁYWU:

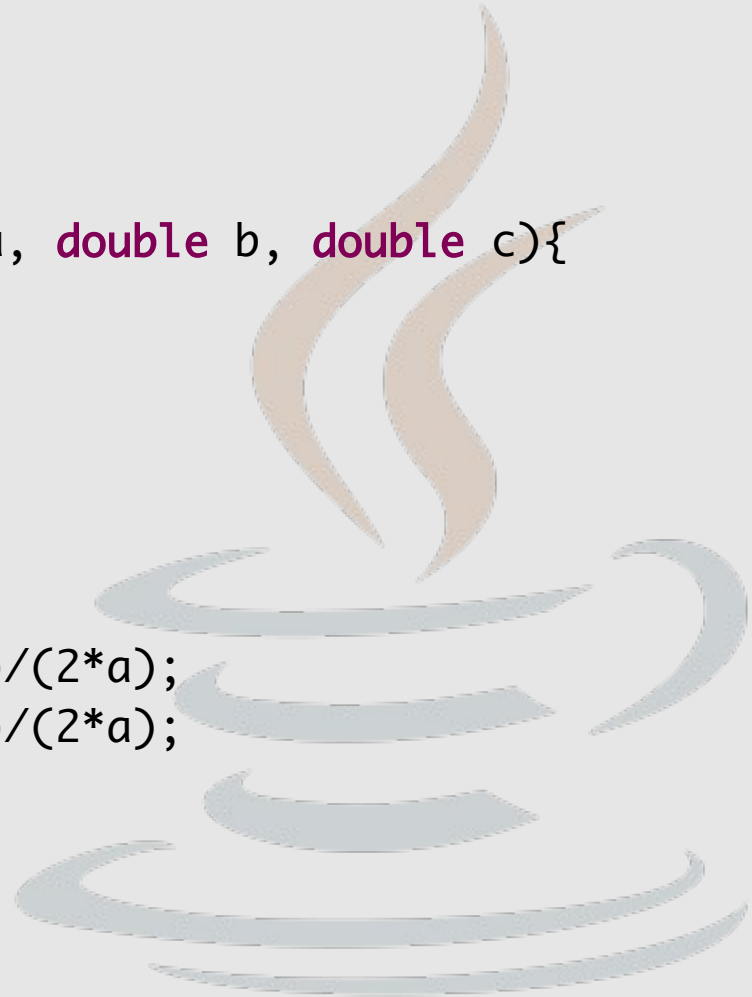
- break, continue, return

```
String[] names = getNames();
for(int i=0; i<s.length; i++){
    if (names[i].equals("JAVA")){
        found = true;              // znalezlismy i nie musimy dalej szukać
        break;
    }
}

File[] f = dir.listFiles();
for(int i=0; i<f.length; i++){
    if (f[i].isDirectory())        // chemy wypisac tylko nazwy plikow
        continue;
    System.out.println(f[i].getName());
}
```

ZASTOSOWANIA

SquareRoots.java

```
public class SquareRoots {

    public static double[] getRoots(double a, double b, double c){
        double[] roots = new double[3];
        double delta = b*b-4*a*c;
        if (delta<0){
            roots[0] = 0;
            return roots;
        }else{
            roots[1] = (-b+Math.sqrt(delta))/(2*a);
            roots[2] = (-b-Math.sqrt(delta))/(2*a);
            roots[0] = (delta==0)?1:2;
        }
        return roots;
    }
```



ZASTOSOWANIA

SquareRoots.java (c.d.)

```
    public static void main(String[] args){
        double a=Integer.parseInt(args[0]);
        if(a==0) System.out.println("Nieprawid\u0142owe dane");
        double b=Integer.parseInt(args[1]);
        double c=Integer.parseInt(args[2]);
        double[] results = getRoots(a, b, c);
        String[] sa = {"Liczba rzeczywistych pierwiastk\u00f3w: ",
                       "x1 = ", "x2 = "};
        for(int i=0; i<results[0]+1; i++)
            System.out.println(sa[i] + results[i]);
    }
}
```

URUCHOMIENIE (po skompilowaniu):

```
java SquareRoots 1 2 -2
```

ŚRODOWISKA DEWELOPERSKIE

- NETBEANS

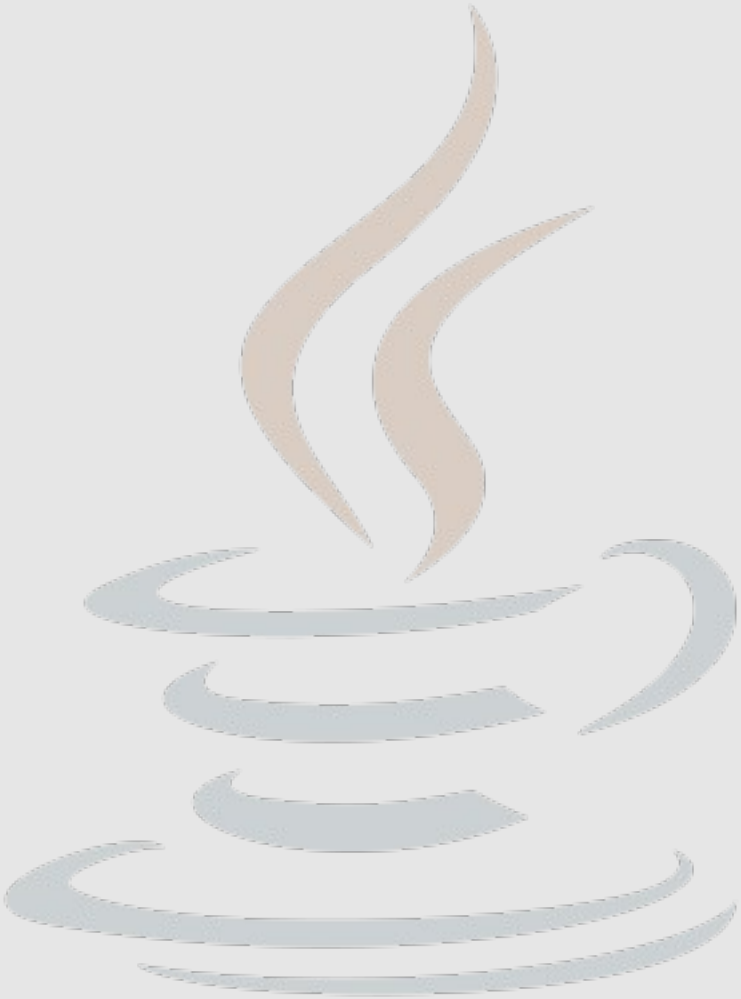
  http://netbeans.org/

- ECLIPSE

  http://www.eclipse.org/

- INTELLIJ IDEA

  http://www.jetbrains.com/idea/

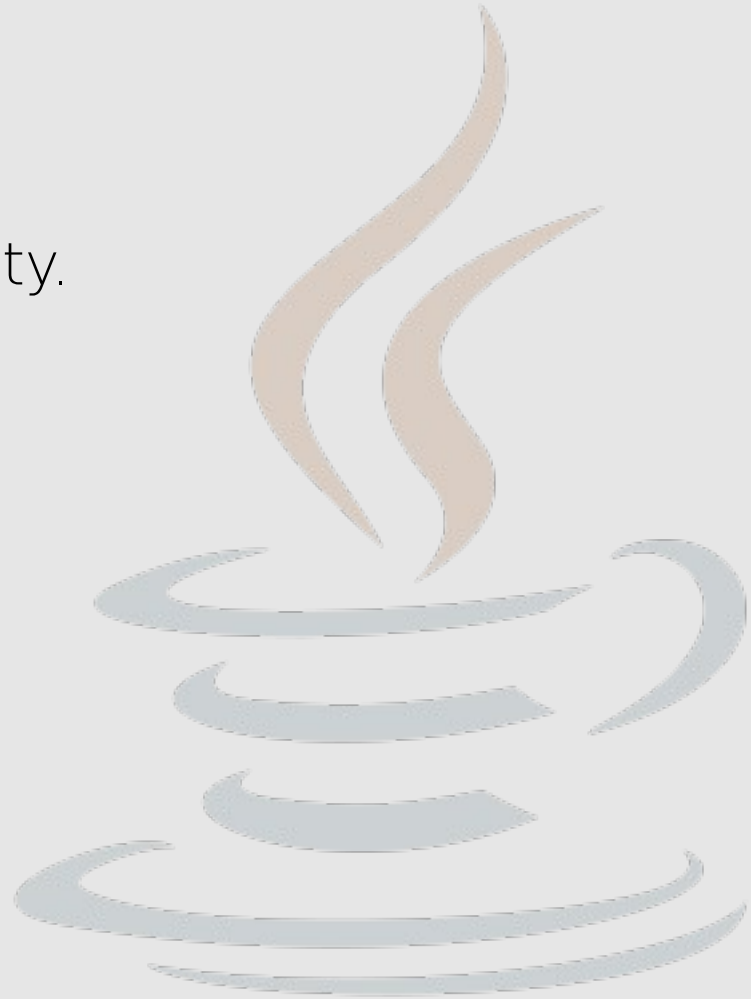


DZIĘKUJĘ ZA UWAGĘ

KLASY, INTERFEJSY, ITP

ZAGADNIENIA:

- Komentarze i javadoc,
- Klasy, modyfikatory dostępu, pakiety.
- Zmienne i metody statyczne.
- Klasy abstrakcyjne, dziedziczenie.
- Interfejsy.
- Wyjątki.



ZASTOSOWANIA

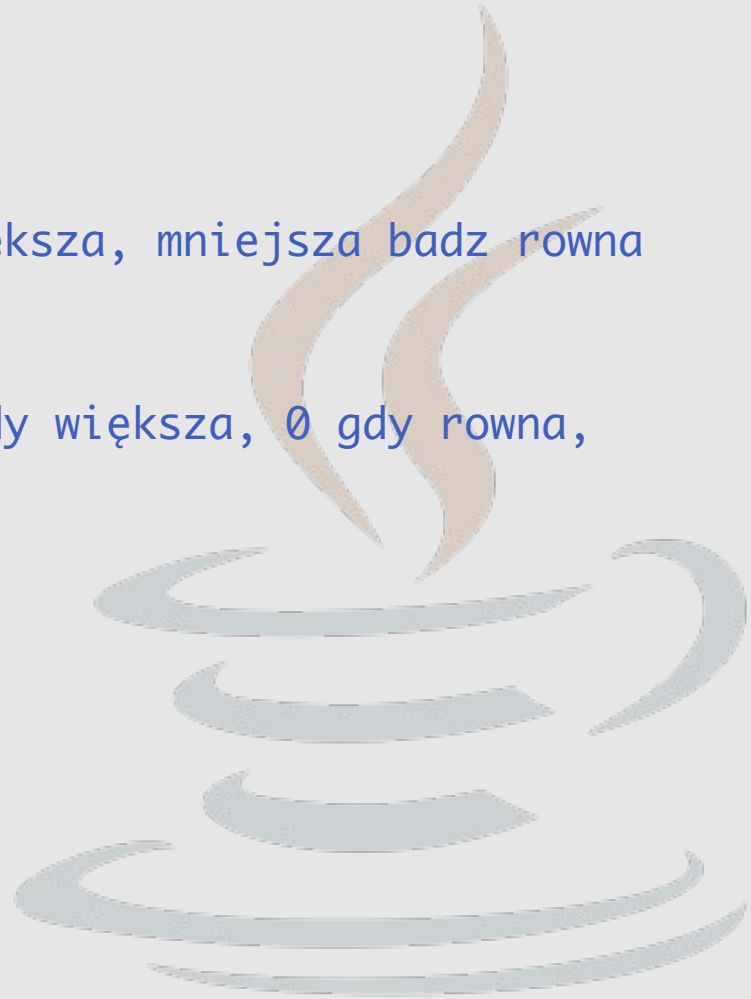
TryAndCheck.java

```
import java.io.IOException;

/**
 * Klasa umożliwiająca zgadywanie liczby, ktora wylosowal komputer
 * @author Kubus Puchatek
 */
public class TryAndCheck {
    private int number;

    /**
     * konstruktor, w nim odbywa sie losowanie liczby
     */
    public TryAndCheck(){
        this.number = (int)(Math.random()*10);
    }
```

ZASTOSOWANIA

TryAndCheck.java (c.d.)

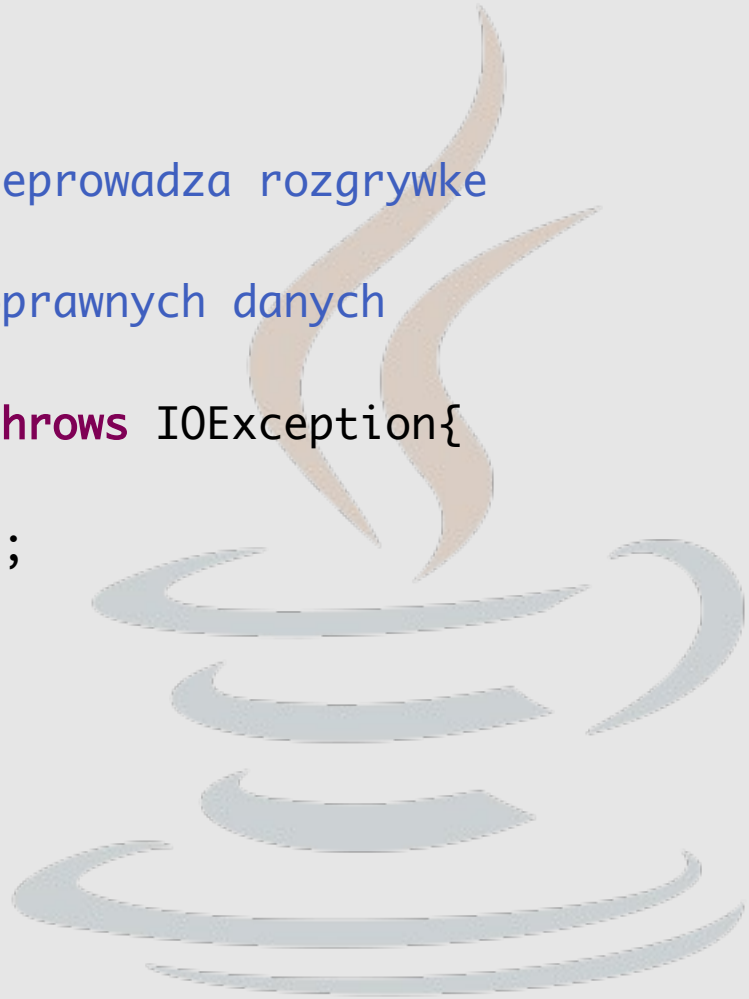
```
/**
 * sprawdza, czy podana wartosc jest wieksza, mniejsza badz rowna
 * wylosowanej liczbie
 * @param iv
 * @return -1 gdy iv jest mniejsza, 1 gdy większa, 0 gdy rowna,
 */
public byte check(int iv){
    if (iv<this.number) return -1;
    if (iv>this.number) return +1;
    return 0;
}
```



ZASTOSOWANIA

TryAndCheck.java (c.d.)

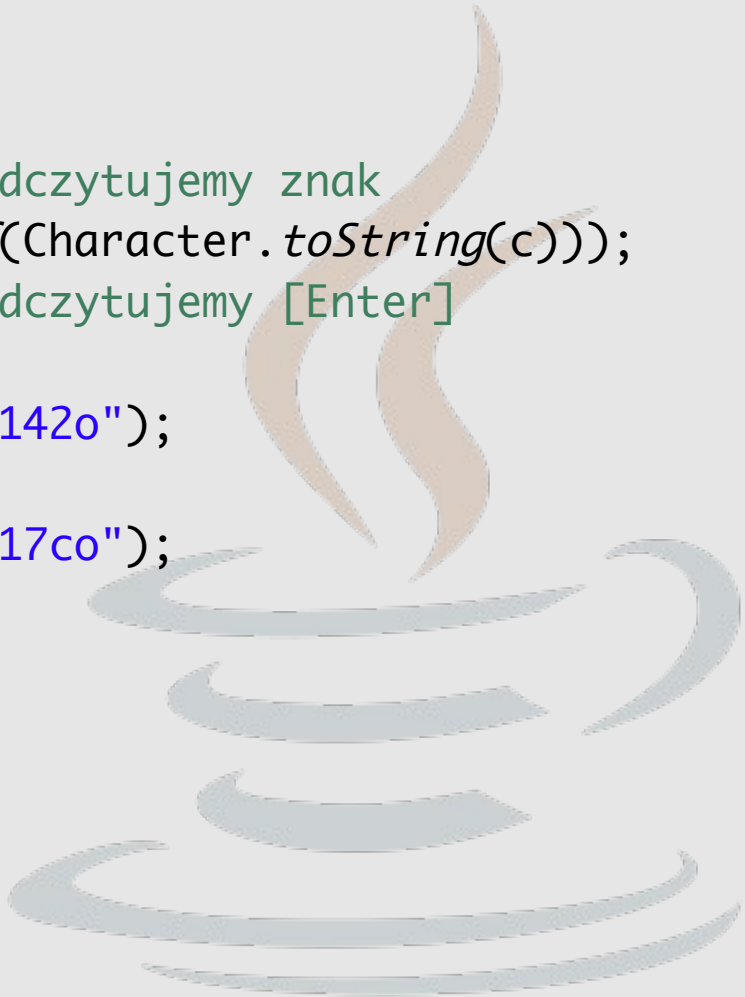
```
/**
 * metoda uruchamiana automatycznie. Przeprowadza rozgrywke
 * @param args nieobslugiwane
 * @throws IOException w przypadku niepoprawnych danych
 */
public static void main(String[] args) throws IOException{

    TryAndCheck play = new TryAndCheck();
    int res;
    char c;
```



ZASTOSOWANIA

TryAndCheck.java (c.d.)

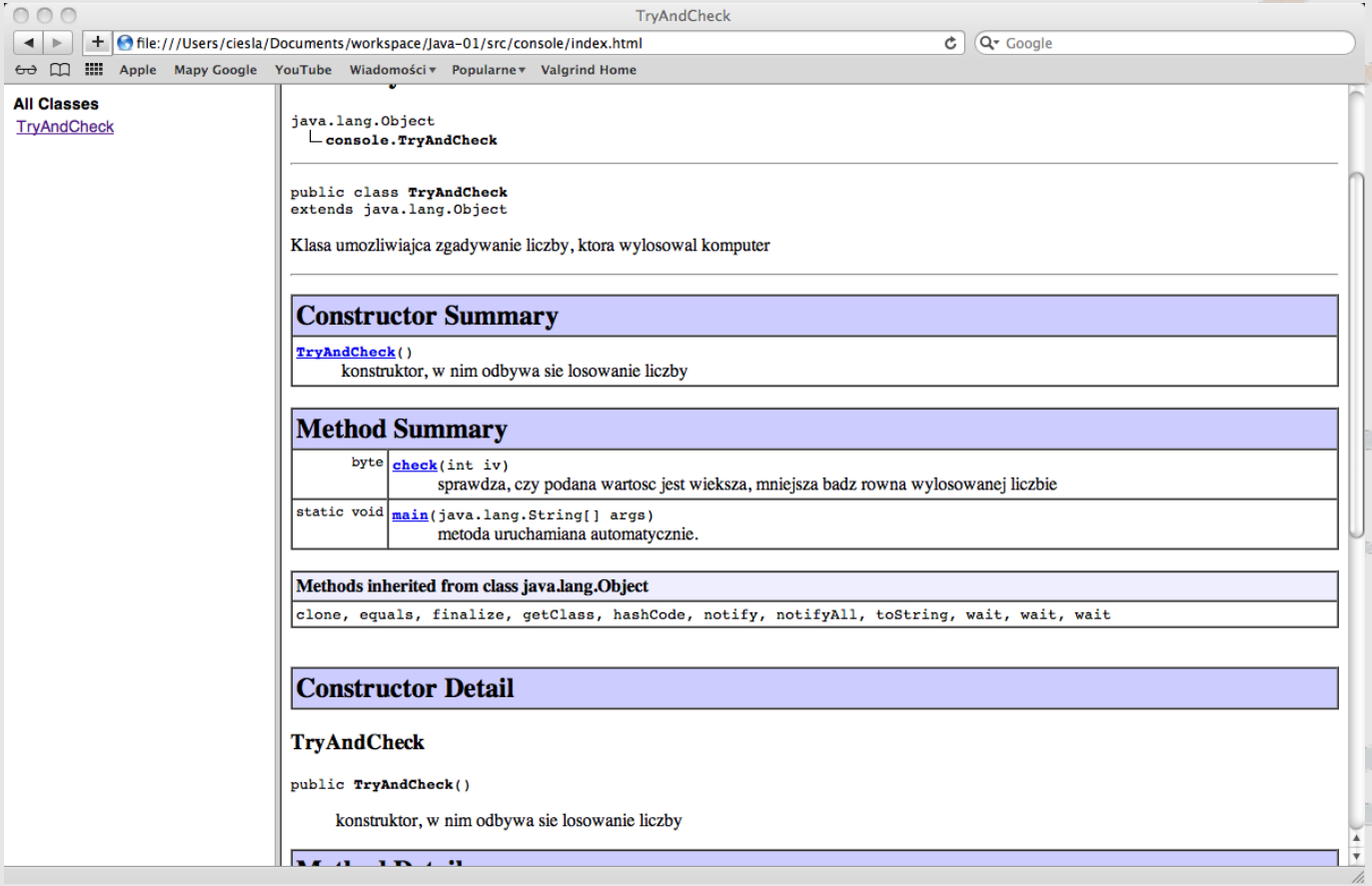
```
        do{
            c = (char)System.in.read(); // odczytujemy znak
            res = play.check(Integer.valueOf(Character.toString(c)));
            c = (char)System.in.read(); // odczytujemy [Enter]
            if(res<0)
                System.out.println("Za ma\u0142o");
            if(res>0)
                System.out.println("Za du\u017co");
        }while(res!=0);
        System.out.println("Gratulacje");
    }
}
```



JAVADOC

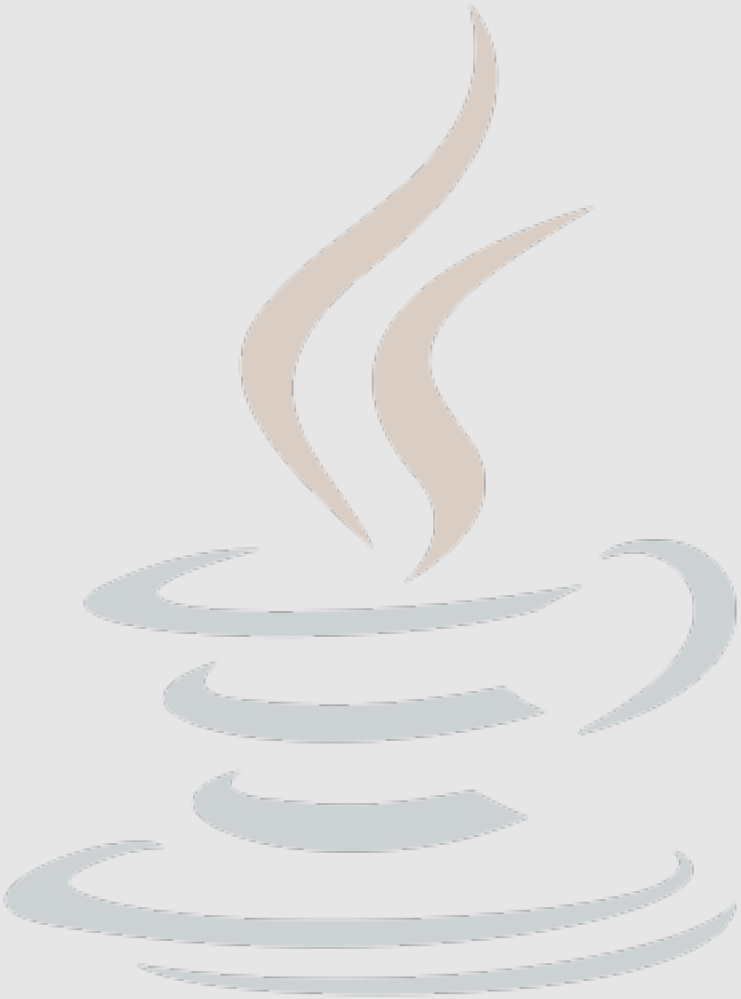
AUTOMATYCZNE GENEROWANIE DOKUMENTACJI

`javadoc TryAndCheck.java`



DWIE KLASY

```
public class Klasa1{
    public void metoda1(){
        ...
    }
...
}

public class Klasa2{
...
    public void metoda2(){
        Klasa1 k1;
        k1 = new Klasa1(...);
        k1.metoda1();
    }
}
```

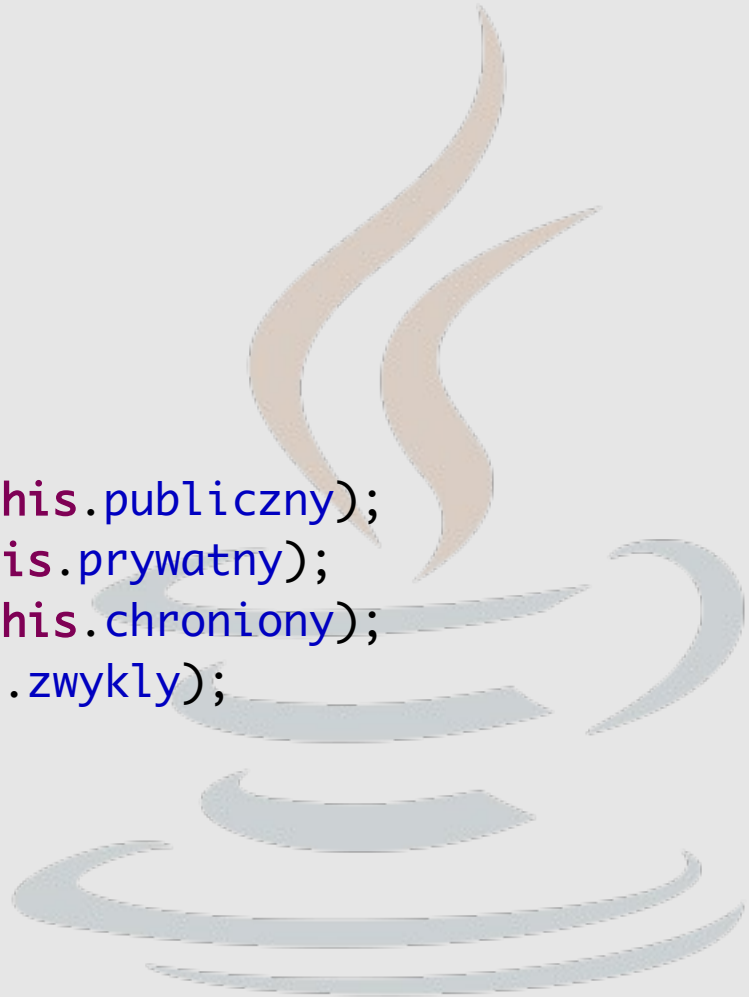


KLASY

```
package pakiet.podpakiet;
public class Klasa {
    public int publiczny; // public - dostępny wszędzie
    private int prywatny; // private – dostępny tylko dla metod tej klasy
    protected int chroniony; // protected – dostępny tylko w danej klasie
                             // i klasach potomnych
    int zwykly; // protected – dostępny tylko w danej klasie i klasach
                //z tego samego pakietu
    protected Klasa(){
        // konstruktor moze nic nie robic, moze go nie byc,
        // nie musi byc publiczny
    }
    public Klasa(int a, int b, int c, int d){
        this.publiczny = a;
        this.prywatny = b;
        this.chroniony = c;
        this.zwykly = d;
    }
```

KLASY

```
public void set(){
    this.publiczny = 7;
    this.prywatny = 13;
    this.chroniony = 27;
    this.zwykly = 11;
}

public void print(){
    System.out.println("publiczny: " + this.publiczny);
    System.out.println("prywatny: " + this.prywatny);
    System.out.println("chroniony: " + this.chroniony);
    System.out.println("zwykly: " + this.zwykly);
    System.out.println();
}
```



KLASY

```
    public static void main(String args[]){
        Klasa k1 = new Klasa();
        k1.print();
        k1.set();
        k1.print();
        Klasa k2 = new Klasa(1,2,3,4);
        k2.print();
    }
}
```

URUCHOMIENIE:

`java pakiet.podpakiet.Klasa`

plik **`Klasa.class`** musi sie znajdowac w podkatalogu **`./pakiet/podpakiet/`**

PAKIETY

Klasy można grupować w pakiety. Nazwa pakietu, do którego należy klasa jest podana w pliku definiującym klasę:

```
package pakiet.podpakiet;
```

Jeśli chcemy użyć klasy z innego pakietu niż nasz, musimy ją uprzednio zaimportować:

```
import pakiet.podpakiet.Klasa;
```

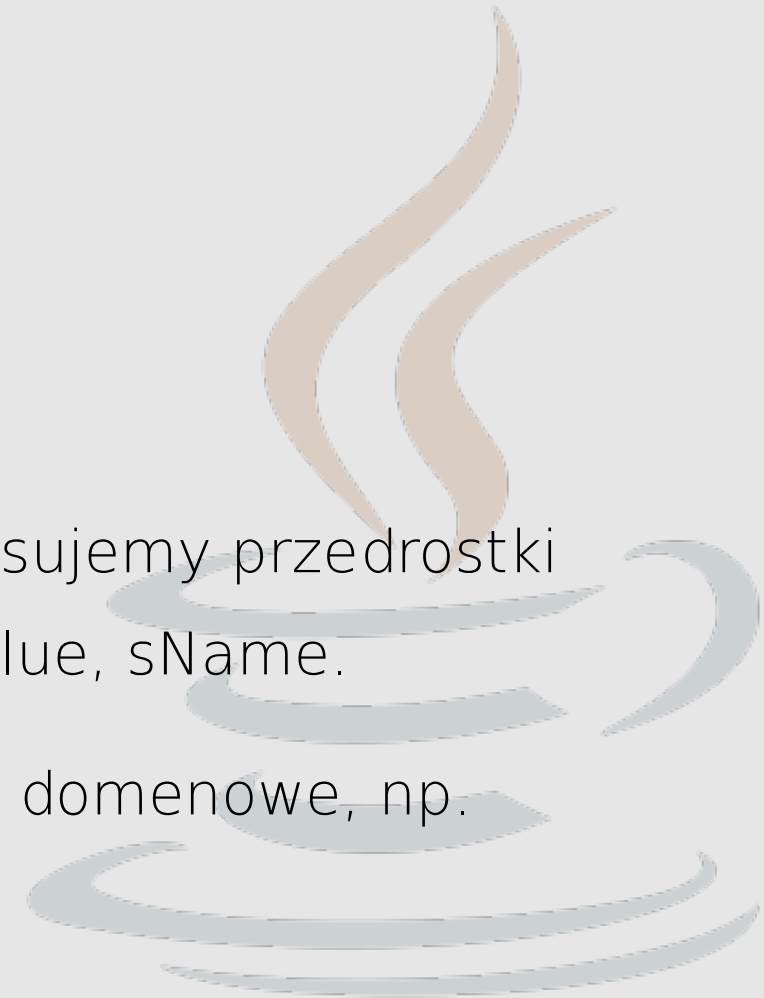
lub

```
import pakiet.podpakiet.*;
```

hierarchia pakietów jest odwzorowana w systemie plików w hierarchie katalogów.

KLASY - KONWENCJE

- nazwa pakietu: z małej litery,
- nazwa klasy: z DUŻEJ litery,
- nazwa atrybutu: z małej litery,
- nazwa metody: z małej litery,
- nazwa zmiennej: z małej litery, stosujemy przedrostki określające typ zmiennych, np. iValue, sName.
- nazwy pakietów – odwrotne nazwy domenowe, np. `pl.uj.edu.fais.java.wyklad2`



ZMIENNE I METODY STATYCZNE

```
public class Klasa1{
    public static void metoda1(){
        ...
    }
}
public class Klasa2{
...
    public void metoda2(){
        Klasa1.metoda1();

    }
}
```

Atrybuty i metody statyczne są związane z klasą a nie z jej instancjami (obiektami). Metody statyczne nie mogą więc bezpośrednio wywoływać zwykłych metod lub korzystać ze zwykłych atrybutów, gdyż one są określone wyłącznie w kontekscie obiektów.

ZMIENNE I METODY STATYCZNE

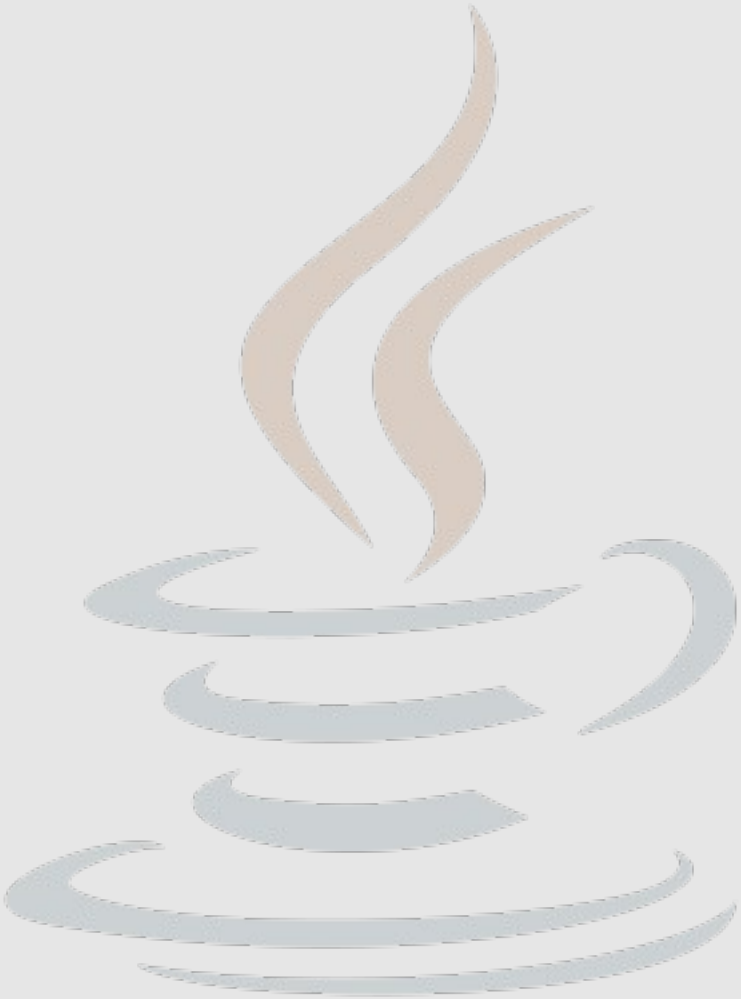
```
public class Klasa1{
...
    static{ // jawna inicjalizacja zmiennych statycznych
        ...
    }
...
```

Atrybuty statyczne są inicjalizowane bezpośrednio po załadowaniu klasy przez JVM.

Atrybuty zwykłe (niestatyczne) są inicjowane w momencie utworzenia obiektu (wywołania konstruktora). Zadeklarowane a niezainicjowane atrybuty są ustawiane na **0** lub **null**.

ZMIENNE I METODY STATYCZNE

```
public class OrderTest {

    static{
        System.out.println("static");
    }

    public OrderTest(){
        System.out.println("constructor");
    }

    public static void main(String[] args){
        System.out.println("main: begin");
        OrderTest o;
        System.out.println("main: middle");
        o = new OrderTest();
        System.out.println("main: end");
    }
}
```



KLASY ABSTRAKCYJNE

```
public abstract class AbstractClass {
    ...
    public abstract int doSomething();
    public int doSomethingElse(){
        ...
    }
    ...
}

public class SpecificClass extends AbstractClass{
    public int doSomething(){
        ...
    }
}
```

Klasa abstrakcyjna to klasa, której jedna z metod jest abstrakcyjna.

Nie można bezpośrednio tworzyć instancji klasy abstrakcyjnej.

DZIEDZICZENIE

```
public class AnyClass extends AnotherClass{
    ...
}
```

Klasa może mieć tylko jednego, bezpośredniego rodzica (inaczej niż w C++). Jeśli klasa nie posiada rodzica, dziedziczy automatycznie po klasie Object (java.lang.Obiect). W związku z tym instancja dowolnej klasy jest obiektem (instancją klasy Object).

INTERFEJSY

```
public interface InterfaceExample {
    public void method1();
    public int method2(double i);
    public AnotherInterface method3(AnyClass ac, AnyInterface ai);
}

public class ImplementationClass implements InterfaceExample, OtherIf {
    public void method1(){
        ...
    }
    ...
}
```

Interfejsy posiadają hierarchię dziedziczenia, jednak jest ona zupełnie niezależna od hierarchii klas.

DZIĘKUJĘ ZA UWAGĘ

WYJĄTKI, KOLEKCJE

ZAGADNIENIA:

1. Wyjątki,
2. Kolekcje,
   - vector,
   - hashtable,
   - properties,
   - Klasy **`Arrays`** i **`Collections`**.

WYJĄTKI

Błędy wykonania programu są sygnalizowane z wykorzystaniem obiektów (`Throwable`). Klasa `Throwable` posiada dwie klasy potomne:

- wyjątki (`Exception`),
- błędy (`Error`).

Wśród wyjątków znajduje się jedna szczególna klasa: `RuntimeException`, określająca błędy pojawiające się w trakcie działania programu, których nie można było łatwo przewidzieć na etapie tworzenia oprogramowania np. `NullPointerException` lub `IndexOutOfBoundsException`.

WYJĄTKI

Obsługa pozostałych wyjątków jest obowiązkowa, tzn. jeżeli korzystamy z metody mogącej zwrócić wyjątek musimy wykonać jedną z dwóch czynności:

- obsłużyć wyjątek za pomocą **try...catch...(finally...)**,

```
try{
    ...
}catch(FileNotFoundException ex){
    ex.printStackTrace();
    ...
}finally{
    ...
}
```

- zadeklarować, że nasza metoda może zwrócić ten wyjątek:

```
public void aMethod() throws FileNotFoundException{...}
```

WYJĄTKI

Obsługa wielu wyjątków:

```
try{
    ...
}catch(FileNotFoundException ex){
    ...
}catch(NullPointerException ex){
    ...
}catch(IOException ex){

}finally{
    ...
}
```

Od Javy 7 możliwe łączenie obsługi:

```
catch(FileNotFoundException | NullPointerException ex){
```

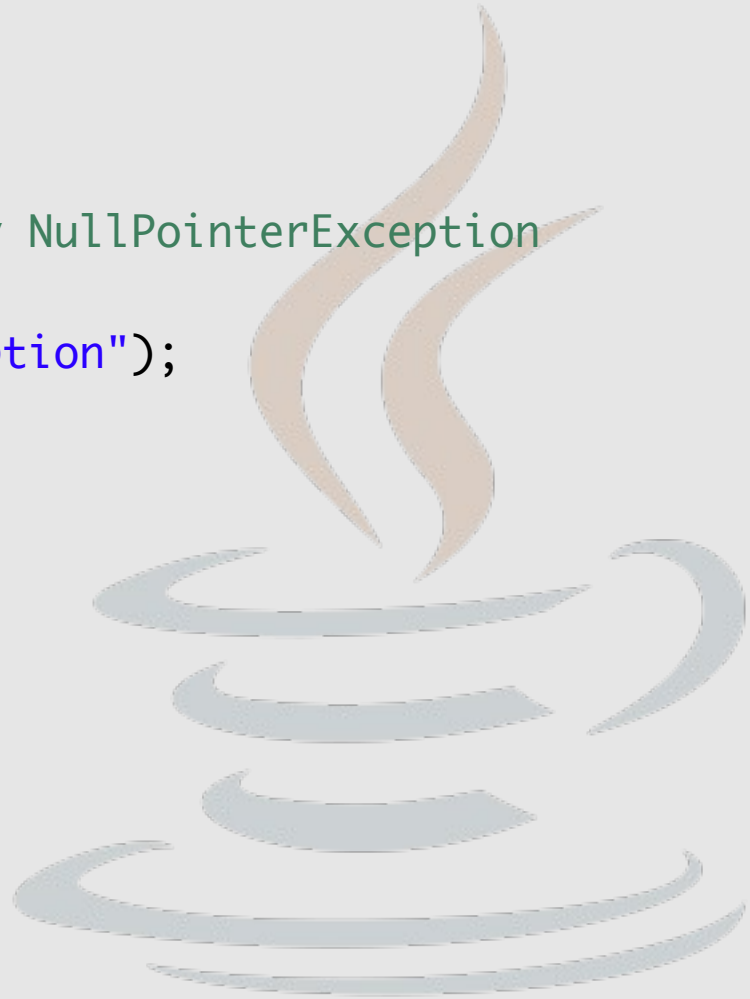
WYJĄTKI

Kolejność obsługi:

```
String s=null;
try{
    s.split(" ");  // tutaj jest rzucany NullPointerException
}catch(NullPointerException ex){
    System.out.println("NullPointerException");
}catch(Exception ex){
    System.out.println("Exception");
}finally{
    System.out.println("Finally");
}
```

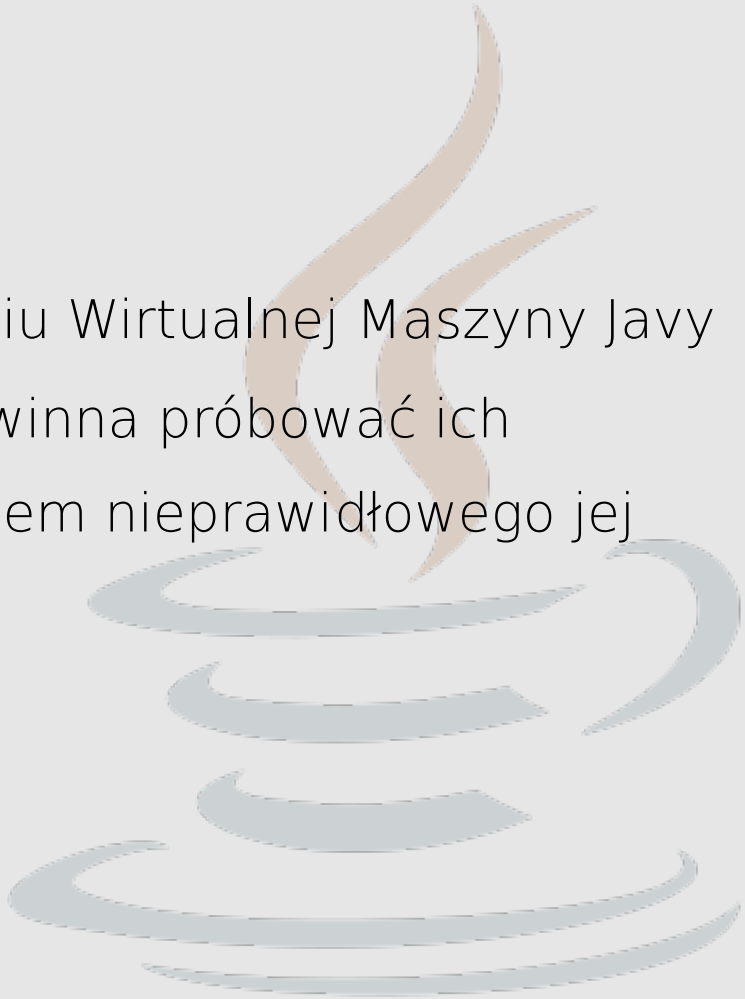
Co wypisze ten fragment kodu?

```
NullPointerException
Finally
```



BŁĘDY

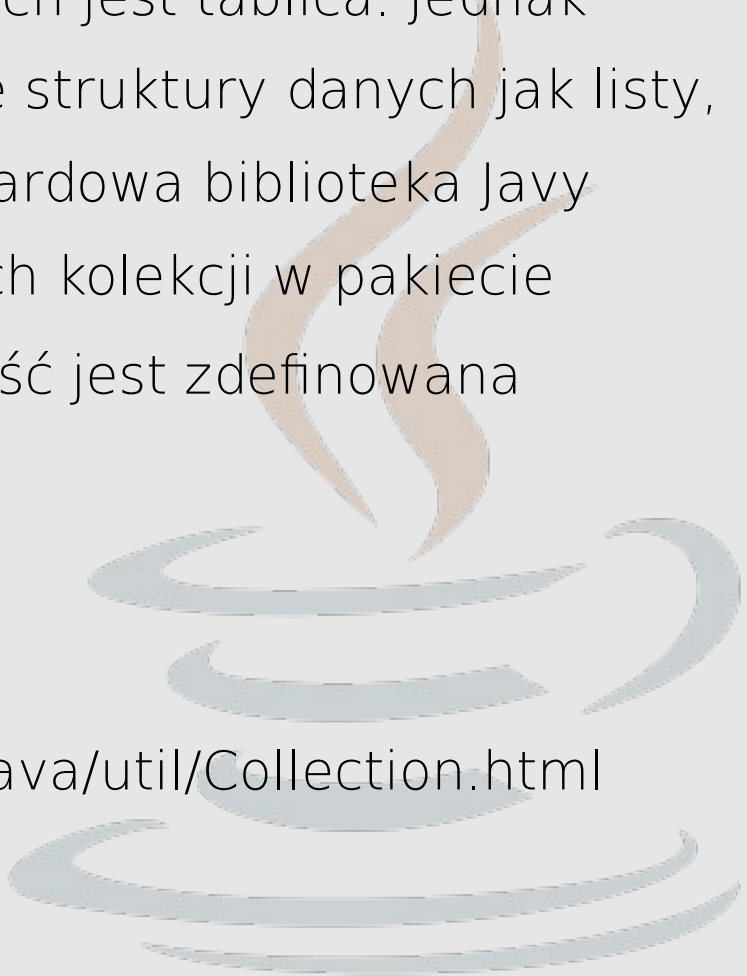
Błędy informują o nieprawidłowym działaniu Wirtualnej Maszyny Javy (np. `OutOfMemoryError`). Aplikacja nie powinna próbować ich obsługiwać, gdyż zwykle nie są one wynikiem nieprawidłowego jej działania.



KOLEKCJE

Najpopularniejszą kolekcją (zbiorem) danych jest tablica. Jednak w wielu zastosowaniach przydatne są inne struktury danych jak listy, zbiory, mapy, tablice haszujące itp. Standardowa biblioteka Javy zawiera implementacje najpopularniejszych kolekcji w pakiecie `java.util.` Ich podstawowa funkcjonalność jest zdefinowana w interfejsie `java.util.Collection`.

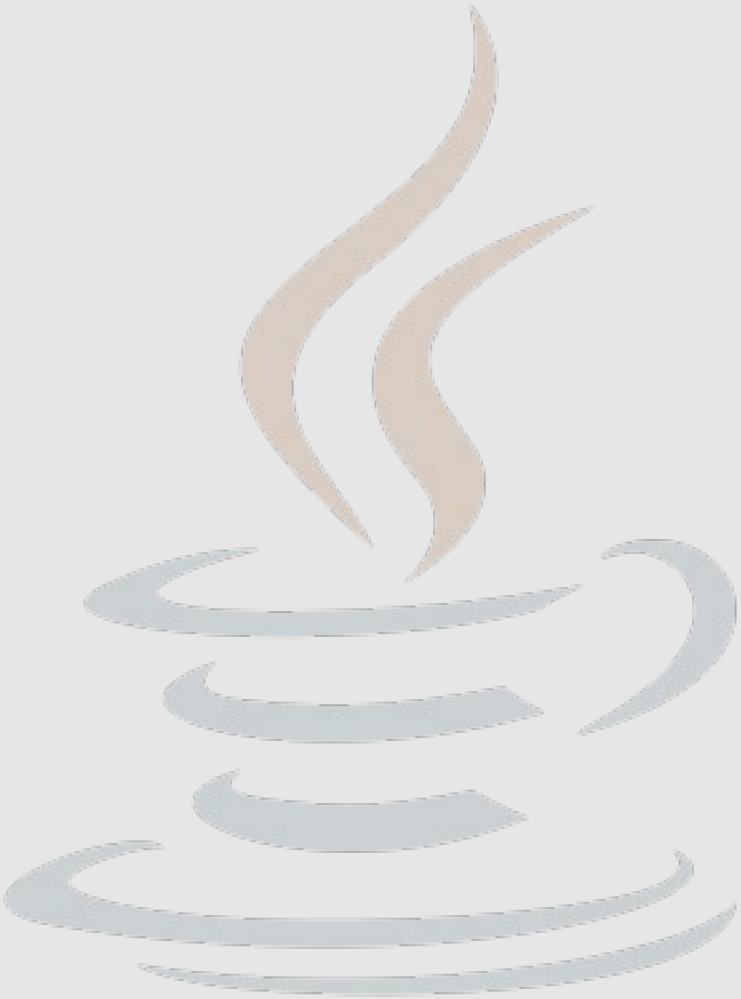
Dokumentacja:
http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html



java.util.Collection

Przykładowe metody:

```
boolean  add(Object o)
void     clear()
boolean  contains(Object o)
boolean  isEmpty()
Iterator iterator()
boolean  remove(Object o)
int      size()
Object[] toArray()
```



java.util.Collection

Przykładowe interfejsy rozszerzające:

`BlockingDeque<E>`

`BlockingQueue<E>`

`Deque<E>`

`List<E>`

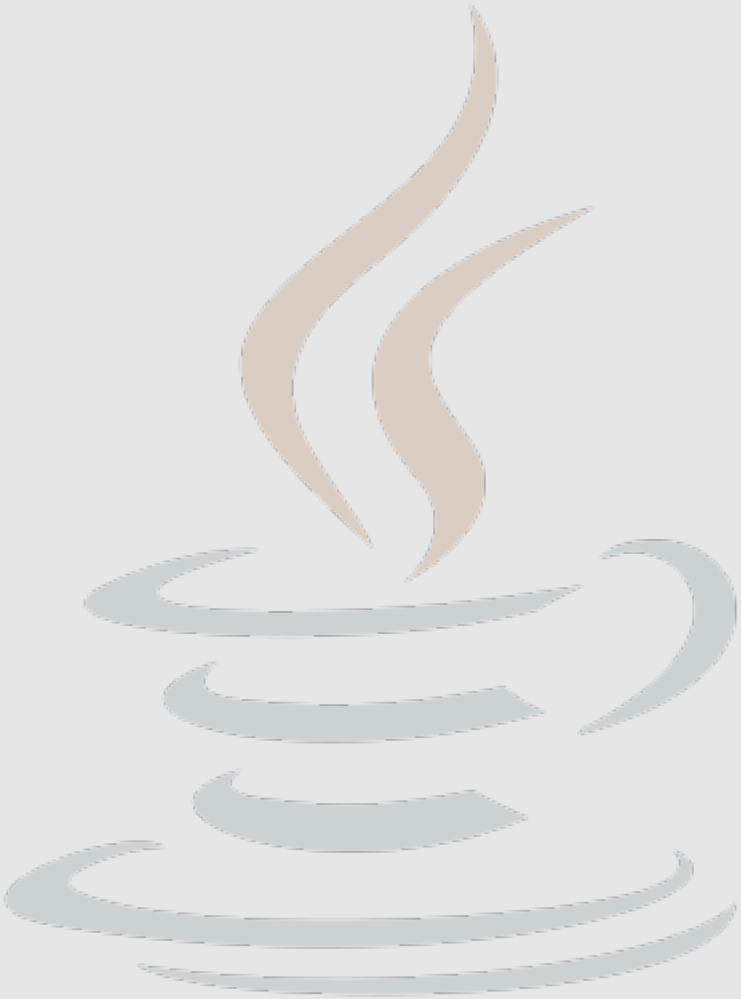
`NavigableSet<E>`

`Queue<E>`

`Set<E>`

`SortedSet<E>`

`TransferQueue<E>`



PRZYKŁADOWE KOLEKCJE

Vector – w rzeczywistości to dynamiczna tablica, której rozmiar jest automatycznie dostosowywany do ilości danych.

```
public class Vector<E> extends AbstractList<E> implements List<E>,
                                        RandomAccess, Cloneable, Serializable
```

```
Vector v = new Vector();
v.add("Ala");
v.add(true);  // dawniej v.add(new Boolean(true));
v.add(128.5);
v.add("Ola");

for(int i=0; i<v.size(); i++){
    Object o = v.get(i);
    System.out.println(o.getClass().getCanonicalName() + "\t" + o);
}
```

VECTOR

```
for(Enumeration e = v.elements(); e.hasMoreElements(); ){
    Object o = e.nextElement();
    System.out.println(o.getClass().getCanonicalName() + "\t" + o);
}

for(Iterator it=v.iterator(); it.hasNext(); ){
    Object o = it.next();
    System.out.println(o.getClass().getCanonicalName() + "\t" + o);
}

for(Object o: v){
    System.out.println(o.getClass().getCanonicalName() + "\t" + o);
}
```

VECTOR<E>

```
v = new Vector();
v.add("Ala");
v.add("Ela");
v.add("Ola");
String s = (String) v.get(0);
```

zaleca się określenie typu elementów w wektorze

```
Vector<String> v1=new Vector<String>();// od v.7 nie trzeba po prawej
                                       stronie pisac typu: new Vector<>()
...
String s = v1.get(0); // nie trzeba rzutować
```

Vector może zawierać też elementy bardziej skomplikowane:

```
Vector<Vector<String>> v1;
```

DYGRESJA: CLONEABLE

```
Vector v1, v2;
v1 = new Vector();
v2 = v1;                        // v2 i v1 referencje do tego samego obiektu

v2 = (Vector) v1.clone();  // tworzona jest kopia obiektu, v1 i v2
                                wskazuja na rozne, blizniacze obiekty

public interface Cloneable{
}
```

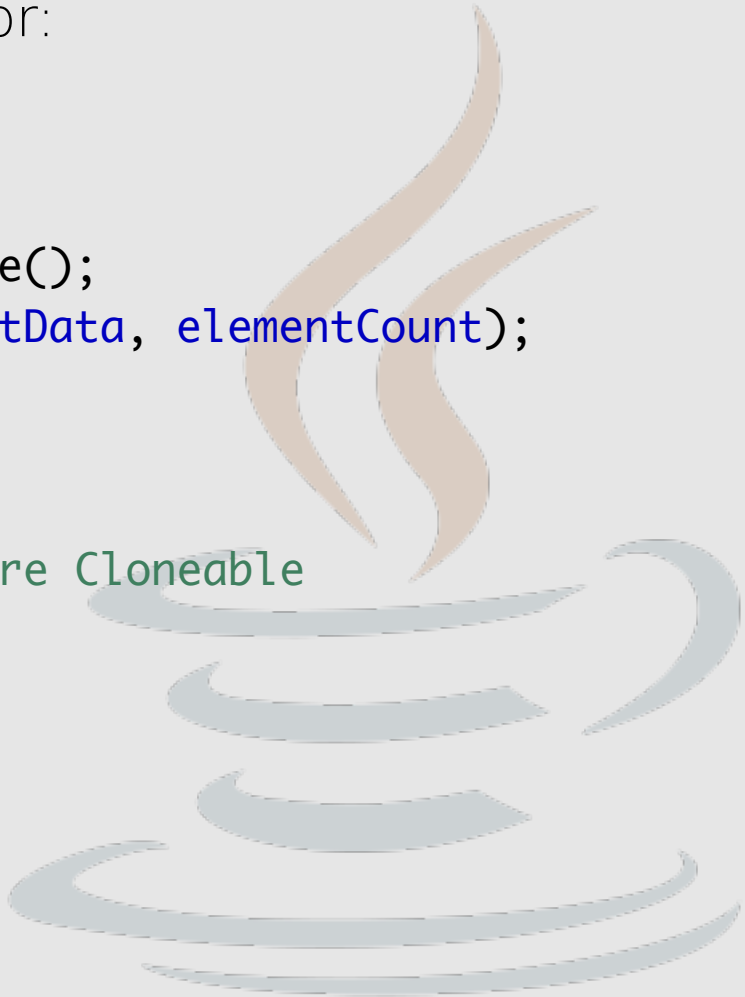
Implementacja interfejsu **Cloneable** informuje, że nasz obiekt wspiera klonowanie. Bazowa metoda clone jest zaimplementowana w klasie **Object**.

```
protected Object clone() throws CloneNotSupportedException
```

DYGRESJA: CLONEABLE

Implementacja kolonowania w klasie Vector:

```
public synchronized Object clone() {
    try {
        Vector<E> v = (Vector<E>) super.clone();
        v.elementData = Arrays.copyOf(elementData, elementCount);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError();
    }
}
```



HASHTABLE

Tablica haszująca to kolekcja (mapa) zawierająca pary (klucz, wartość). Zarówno klucz jak i wartość mogą być dowolnymi obiektami.

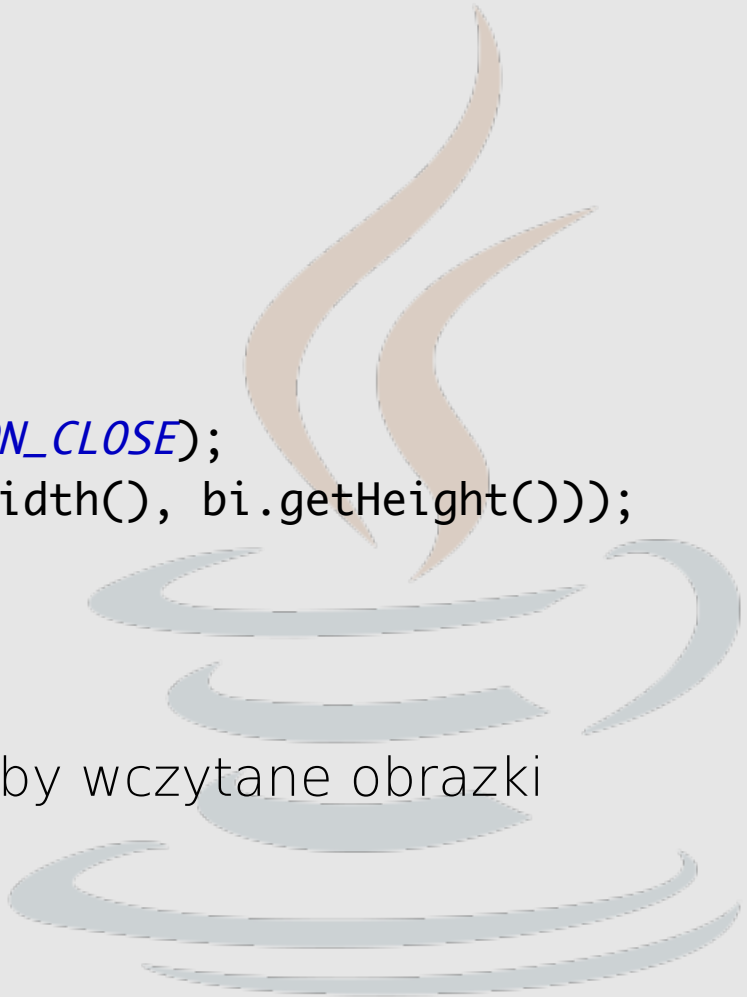
```
public class Hashtable<K,V> extends Dictionary<K,V>
             implements Map<K,V>, Cloneable, Serializable {
```

```
Hashtable<String,BufferedImage> ht=new Hashtable<String,BufferedImage>();
File dir = new File(System.getProperty("user.dir"));
File[] files = dir.listFiles();
for(File f: files){
    if (f.getName().endsWith(".jpg")){
        ht.put(f.getName(), ImageIO.read(f));
    }
}
```

HASHTABLE

```
final BufferedImage bi = ht.get("logo.jpg");
JFrame frame = new JFrame(){
    public void paint(Graphics g){
        super.paint(g);
        g.drawImage(bi, 0, 0, null);
    }
};
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setPreferredSize(new Dimension(bi.getWidth(), bi.getHeight()));
frame.pack();
frame.setVisible(true);
```

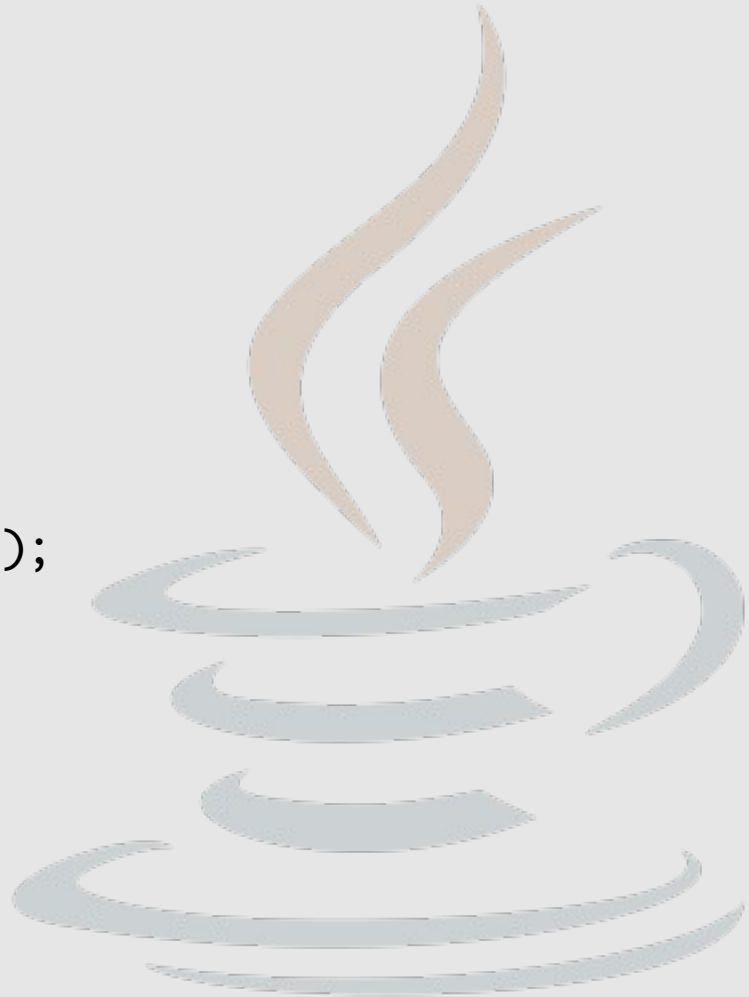
ĆWICZENIE: zmodyfikować program tak, aby wczytane obrazki zmieniały się.



HASHTABLE

Inne przydatne metody:

```
public Set<K> keySet();
public synchronized Enumeration<K> keys();

public Collection<V> values();
public synchronized Enumeration<V> elements();
public Set<Map.Entry<K,V>> entrySet();

public synchronized V remove(Object key);
```



PROPERTIES

Properties to rozszerzenie tablicy haszującej

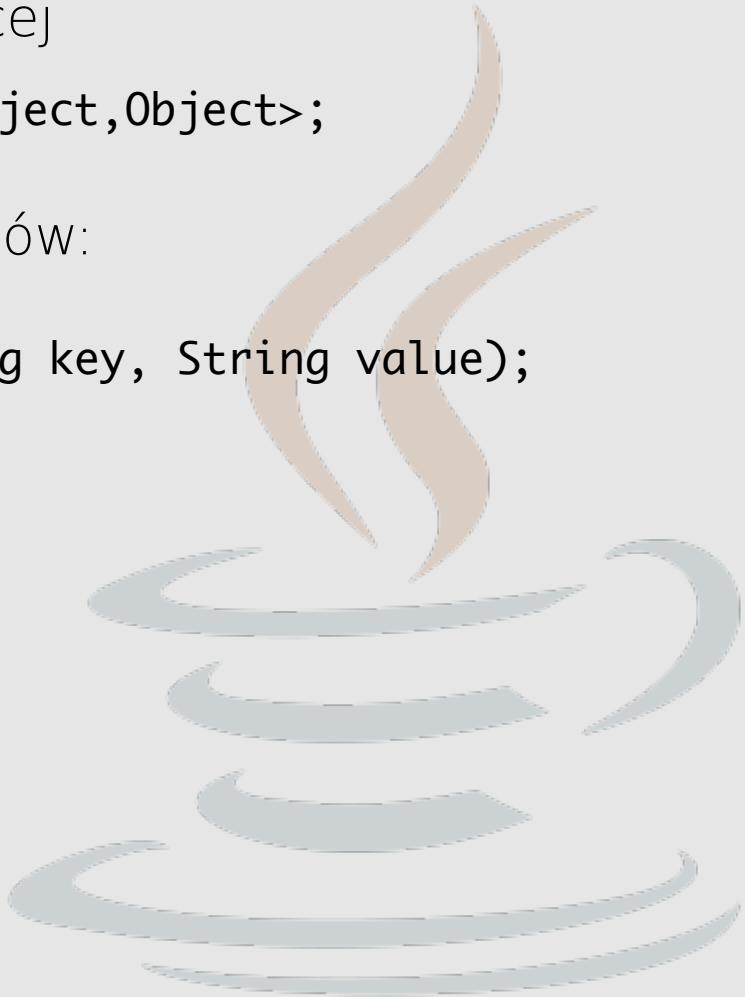
```
public class Properties extends Hashtable<Object,Object>;
```

nastawione na przechowywanie par Stringów:

```
public synchronized Object setProperty(String key, String value);

public String getProperty(String key);
```

ZASTOSOWANIA:

```
Properties p;
p = System.getProperties();
p.list(System.out);
```



PROPERTIES

Implementacja metody `list()`:

```
    public void list(PrintStream out) {
        out.println("-- listing properties --");
        Hashtable h = new Hashtable();
        enumerate(h);
        for (Enumeration e = h.keys() ; e.hasMoreElements() ;) {
            String key = (String)e.nextElement();
            String val = (String)h.get(key);
            if (val.length() > 40) {
                val = val.substring(0, 37) + "...";
            }
            out.println(key + "=" + val);
        }
    }
```

DO ZASTANOWIENIA: dlaczego metoda najpierw przepisuje dane do nowej tablicy i dopiero z niej je wypisuje?

PROPERTIES

Klasa Properties "współpracuje" z plkiami tekstowymi zapisanymi w określonym formacie:

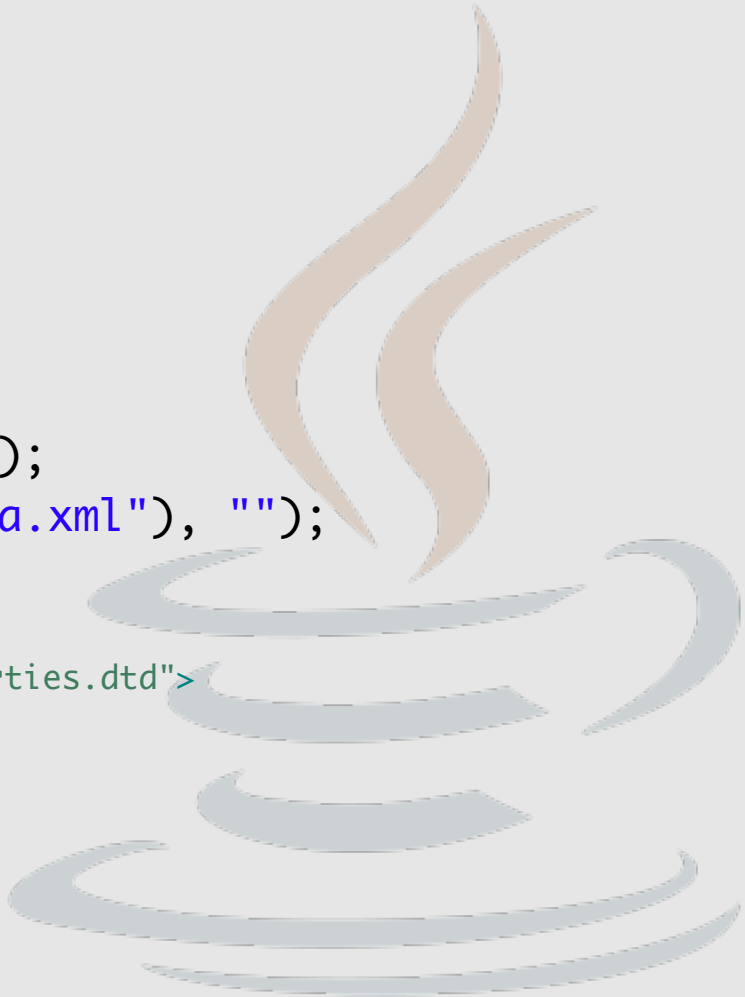
```
public synchronized void load(InputStream inStream) throws IOException

public void store(OutputStream out, String comments)
    throws IOException


public synchronized void loadFromXML(InputStream in)
    throws IOException, InvalidPropertiesFormatException

public synchronized void storeToXML(OutputStream os, String comment)
    throws IOException
```

PROPERTIES

```
# analizowany plik w formacie gif, png, jpg
image=obrazek.png
output=res.png
# inne ustawienia
moversCount=500
stepSize=1
ballRadius=1
```

```
Properties p = new Properties();
p.load(new FileInputStream("ustawienia.txt"));
p.storeToXML(new FileOutputStream("ustawienia.xml"), "");
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment/>
    <entry key="stepSize">1</entry>
    <entry key="ballRadius">1</entry>
    <entry key="output">res.png</entry>
    <entry key="moversCount">500</entry>
    <entry key="image">obrazek.png</entry>
</properties>
```



DWIE UŻYTECZNE KLASY

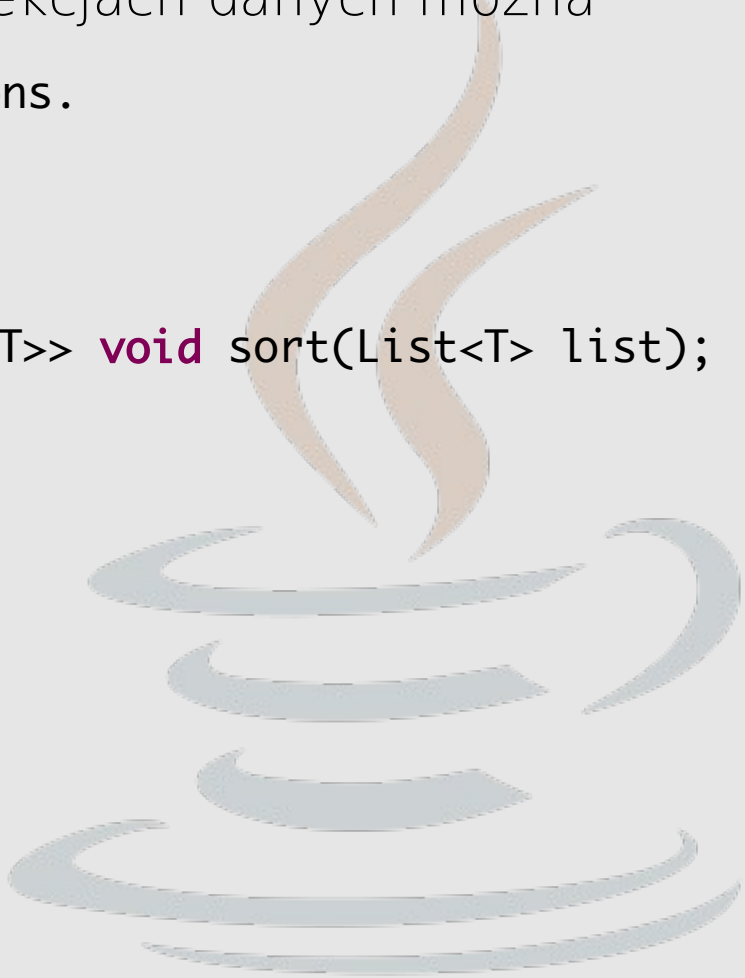
Podstawowe operacje na tablicach lub kolekcjach danych można wykonać za pomocą klas `Arrays` i `Collections`.

np sortowanie.

```
public static <T extends Comparable<? super T>> void sort(List<T> list);
```

przy czym:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```



DZIĘKUJĘ ZA UWAGĘ

TYPY GENERYCZNE (GENERICS)

ZAGADNIENIA:

- wprowadzenie,
- konwencje, metody, typy surowe
- parametry ograniczone
- podtypy, dziedziczenie,
- symbole wieloznaczne,
- ograniczenia.

MATERIAŁY:

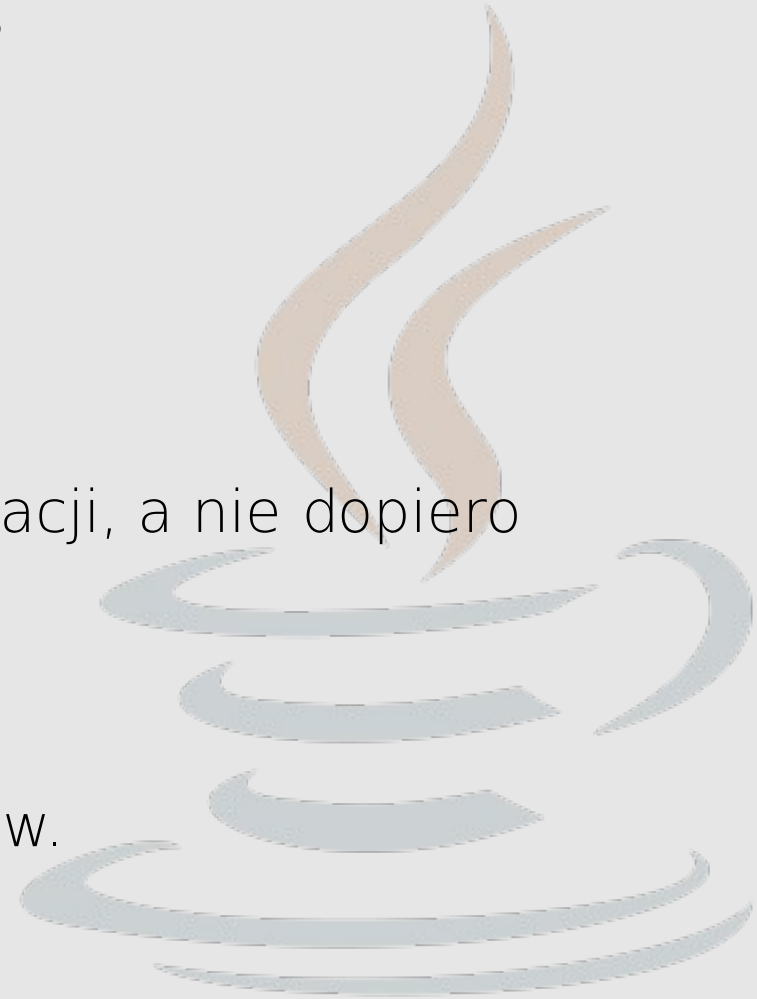
http://docs.oracle.com/javase/tutorial/java/generics/

TYPY GENERYCZNE

```
List<String> list = new ArrayList<String>();
list.add("Ala");
String s = list.get(0);
```

ZALETY:

- kontrola typów na poziomie kompilacji, a nie dopiero w trakcie działania programu
- brak potrzeby rzutowania,
- konstruowanie ogólnych algorytmów.



PRZYKŁAD

```
public class Box<T> { // T oznacza Typ
    private T t;

    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

ogólnie:

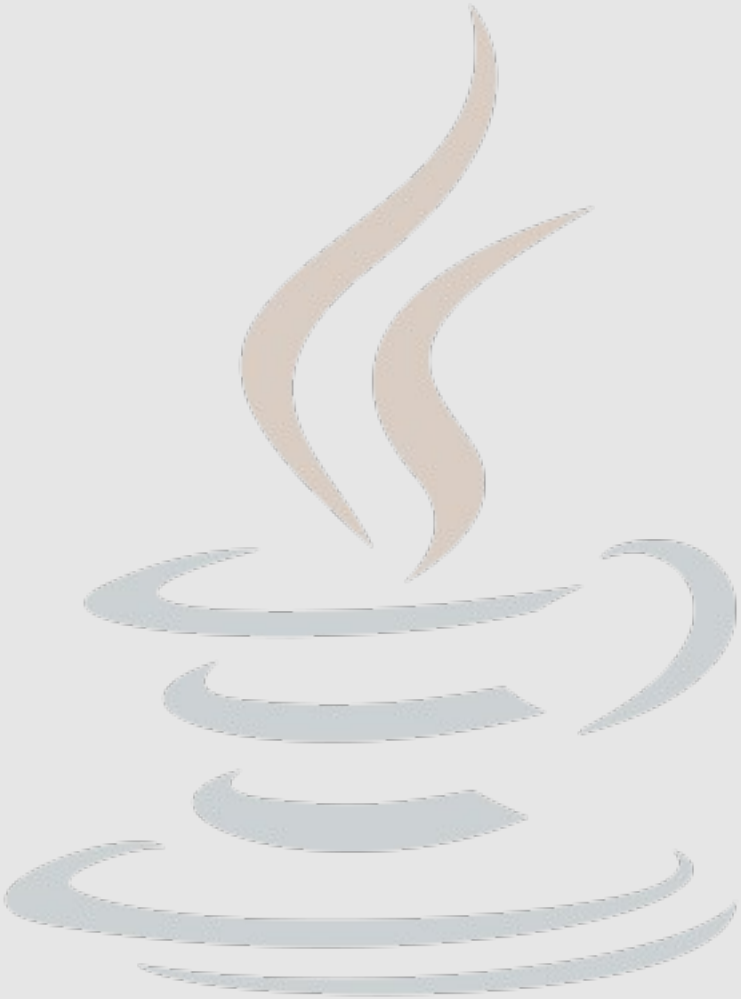
```
class NazwaKlasy<T1, T2, ..., Tn> {
...
}
```

KONWENCJE

- T – typ,
- K – klucz,
- V – wartość,
- E – element (np. kolekcji),
- N – liczba

PRZYKŁAD:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```



PRZYKŁADY

```
public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}

Pair<String, Integer> p = new OrderedPair<String, Integer>("Even", 8);
Pair<String, Integer> p = new OrderedPair<>("Even", 8);  // Java 7
```

TYPY SUROWE

```
public class Box<T> {
    public void set(T t){ ... }
    ...
}

Box<String> typedBox = new Box<String>();
Box rawBox = new Box(); // typ surowy <=> Box<Object>


Box rawBox = typedBox; // warning: przypisanie możliwe
                       // (Objct o – new String()), ale brak jawnej
                       // konwersji typów

rawBox.set(8);         // warning: w trakcie działania programu może
                       // spowodować RuntimeException
```

Ze względu na możliwość popełnienia błędów, nie zaleca się stosowania typów surowych.

METODY GENERYCZNE

```
public class Util {
    // Genericzna metoda statyczna
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

wywołanie metody generycznej:

```
Pair<Integer, String> p1 = new Pair<>(1, "jabłko");
Pair<Integer, String> p2 = new Pair<>(2, "gruszka");
boolean same = Util.<Integer, String>compare(p1, p2);
```

możliwe również prostsze wywołanie dzięki mechanizmowi wnioskowania typów (type inference).

```
boolean same = Util.compare(p1, p2);
```

PARAMETRY OGRANICZONE

```
public class Box<T> {
...
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: String to nie Number
    }
}
```

inny przykład

```
public class NaturalNumber<T extends Integer>
```

PARAMETRY OGRANICZONE

wielokrotne ograniczenia:

```
<T extends B1 & B2 & B3>
```

przykład:

```
class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

typ **T** musi rozszerzać klasę **A** i implementować interfejsy **B** i **C**.

UWAGA: **A** musi być zadeklarowana jako pierwsza (błąd w trakcie kompilacji)

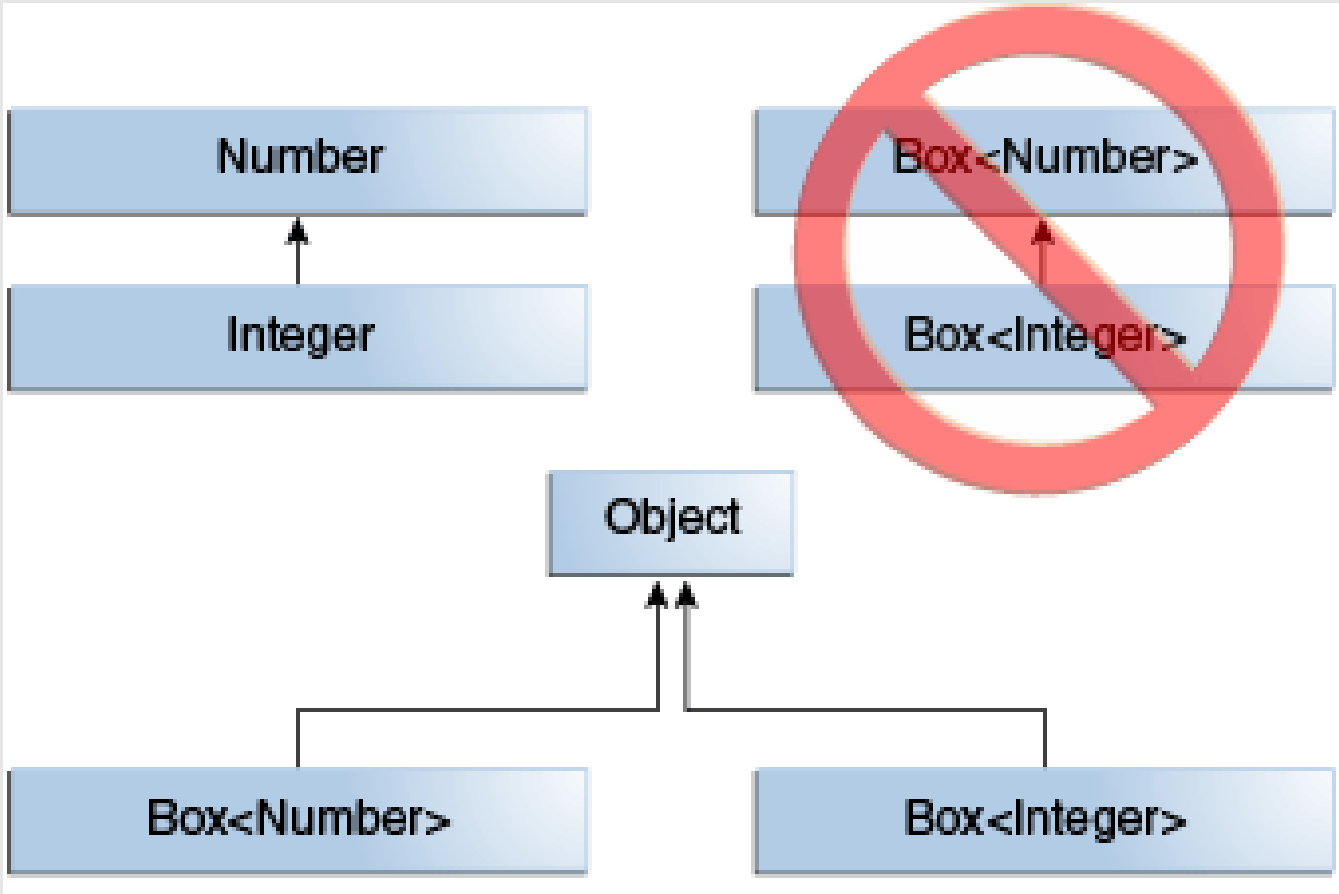
PRZYKŁAD

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // błąd kompilacji. Co oznacza '>'?
            ++count;
    return count;
}
```

poprawnie:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray,
                                                             T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

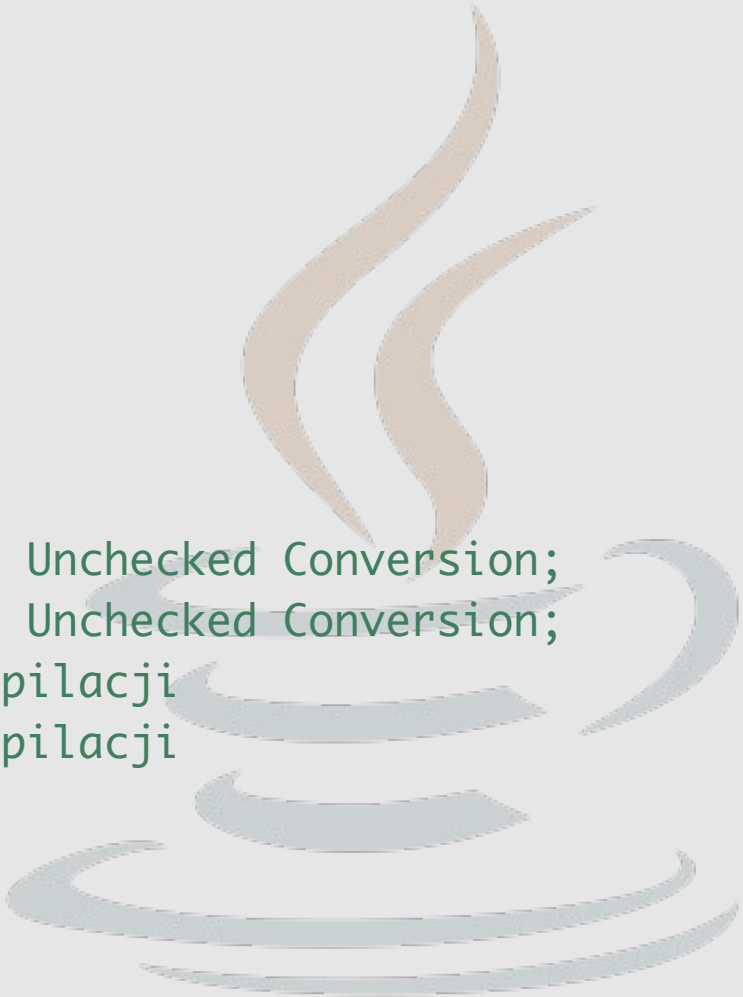
PODTYPY, DZIEDZICZENIE



http://docs.oracle.com/javase/tutorial/figures/java/generics-subtypeRelationship.gif

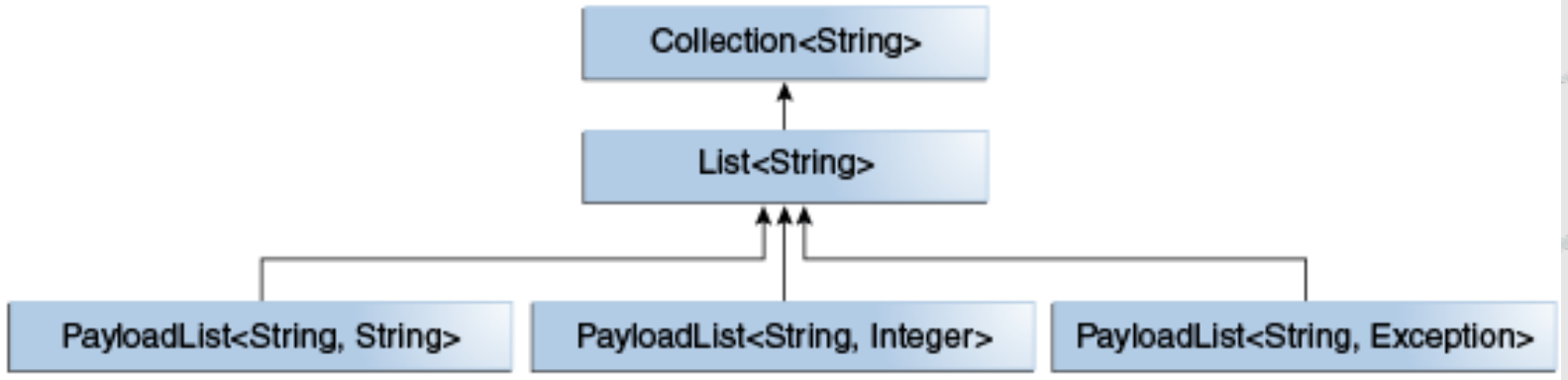
DZIEDZICZENIE PRZYKŁAD

```
Box<Integer> boxInteger;
Box<Number> boxNumber;
Box box;

box        = boxInteger;  // ok
box        = boxNumber;   // ok
boxNumber  = box;         // warning: Unchecked Conversion;
boxInteger = box;         // warning: Unchecked Conversion;
boxInteger = boxNumber;   // błąd kompilacji
boxNumber  = boxInteger;  // błąd kompilacji
```



PODTYPY, DZIEDZICZENIE

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```



http://docs.oracle.com/javase/tutorial/figures/java/generics-payloadListHierarchy.gif

SYMBOLE WIELOZANCZNE

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

Znak '?' ogranicza 'od góry' rodzaj parametru.

UWAGA: deklaracja / definicja metody

```
public static double sumOfList(List<T extends Number> list)
```

jest niepoprawna (składnia). Taka składnia jest douszczalna dla deklaracji klas lub wartości zwracanych (por. parametry ograniczone).

SYMBOLE WIELOZNACZNE

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Znak '?' dopuszcza dowolny typ jako zawartość listy, np.

`List<String>`, `List<Integer>`, itd.

UWAGA: Metoda

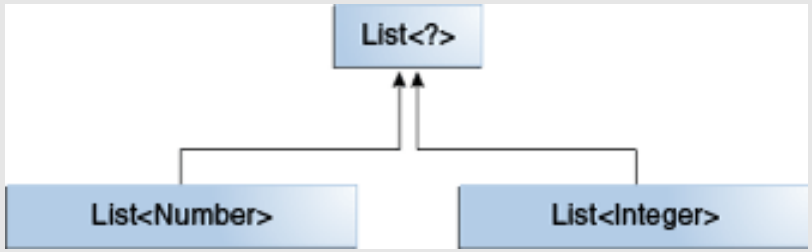
```
public static void printList(List<Object> list)
```

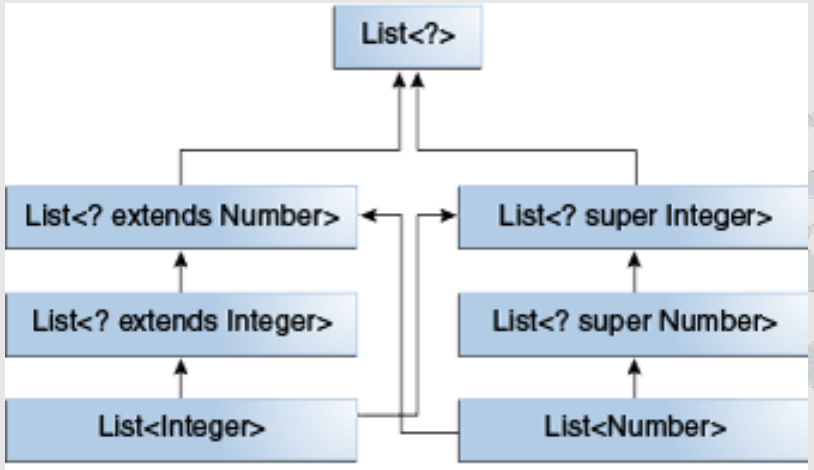
nie obsłuży `List<String>` bo `List<String>` nie jest podklasą

`List<Object>`.

SYMBOLE WIELOZNACZNE

Dziedziczenie a symbole wieloznaczne





http://docs.oracle.com/javase/tutorial/figures/java/generics-wildcardSubtyping.gif

SYMBOLE WIELOZNACZNE

Przechwytywanie symboli wieloznacznych (wildcard capture):

```
public class WildcardError {

    void doSomething(List<?> l) {
        l.set(0, l.get(0)); // blad kompilacji
    }
}
```

Kompilator traktuje `l` jako listę dowolnych obiektów, podczas gdy wywołanie metody `l.set()` wstawia do listy konkretny obiekt, który niekoniecznie musi do niej pasować.

SYMBOLE WIELOZNACZNE

```
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }

    // metoda pomocnicza, która rozwiązuje problem
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }

}
```

Metoda wykorzystuje mechanizm wnioskowania (inference). Tutaj lista jest procesowana jak lista zawierająca konkretne elementy (typu **T**) więc mogą to być obiekty **Integer**.

SYMBOLE WIELOZNACZNE

Wskazówki:

- zmienne IN – górne ograniczenie **extends**,
- zmienne OUT – dolne ograniczenie **super**,
- zmienne IN, mogące być używane jako obiekty – brak ograniczeń,
- zmienne IN/OUT – nie używamy symboli wieloznacznych,
- nie używamy symboli wieloznacznych jako wartości zwracanych.

Listy zadeklarowane z użyciem symboli wieloznacznych można traktować jako obiekty read-only

SYMBOLE WIELOZNACZNE

```
List<Integer> li = new ArrayList<Integer>();
List<? extends Number> ln = li; // OK. ( Object obj = new String() )
ln.add(new Integer(35));  // Blad kompilacji, ( obj.charAt(7) )
```

`List<Integer>` jest podtypem `List<? extends Number>`. A klasy nie można traktować tak jak podklasy. Jedyne operacje które można wykonać na takiej liście to:

- dodanie null
- wyczyszczenie clear()
- pobranie iteratora i wywołanie remove()
- zapisać elementy odczytane z listy z wykorzystaniem 'wildcard capture'.

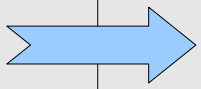
DZIAŁANIE

Typy generyczne zostały wprowadzone aby lepiej kontrolować typy na etapie kompilacji i aby umożliwić programowanie generyczne. W generowanym bytecodzie otrzymujemy zwykłe klasy i interfersy. Aby to zrealizować kompilator:

- zastępuje typy przez ich ograniczenia a nieograniczone typy przez **Object**,
- dodaje niezbędne operacje rzutowania,
- tworzy metody pomostowe implementujące polimorfizm w rozszerzonych typach generycznych.

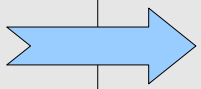
DZIAŁANIE

```
public class Box<T> {

    private T t;
    public void set(T t) {
        this.t = t;
    }
}
```



```
public class Box {

    private Object t;
    public void set(Object t) {
        this.t = t;
    }
}
```

```
public class Box<T extends Number> {

    private T t;
    public void set(T t) {
        this.t = t;
    }
}
```



```
public class Box {

    private Number t;
    public void set(Number t) {
        this.t = t;
    }
}
```

METODY POMOSTOWE

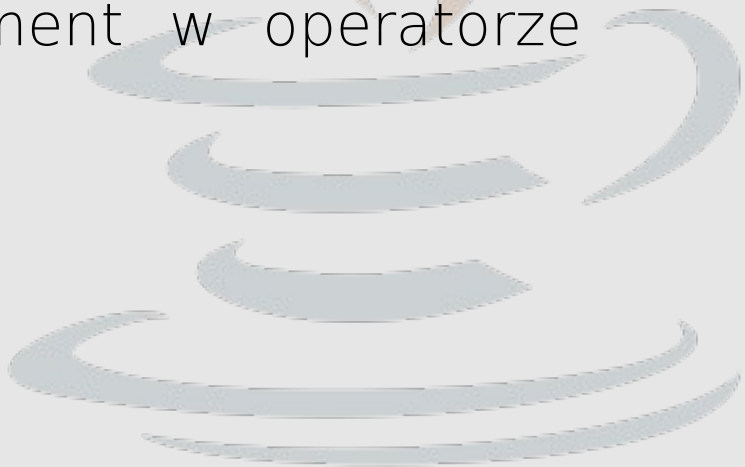
```
public class MyBox extends Box<String>{
    public void set(String s){
        super.set(s);
    }
}
```

metoda **set()** w klasie **Box** przyjmuje argument Object podczas gdy metoda w klasie potomnej przyjmuje **String**. To powodowałoby brak polimorfizmu. Dlatego kompilator automatycznie dodaje metodę pomostową (do **MyBox**):

```
public void setData(Object data) {
    setData((Integer) data);
}
```

OGRANICZENIA

- nie można używać typów prymitywnych (**Box<int>**),
- nie można używać operatora **new** (**E e = new E();**),
- nie można deklarować typów statycznych (**private static T v;**)
- typów parametryzowanych (**List<Integer>**) nie można rzutować ani używać jako argument w operatorze `instanceof`,



OGRANICZENIA

- nie można używać tablic typów parametryzowanych (**List<Integer>[] arrayOfLists = new List<Integer>[2]**),
- typ generyczny nie może rozszerzać (bezpośrednio lub pośrednio) **Throwable**,
- nie można przeciążać metod, których argumenty sprowadzają się do tego samego typu.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

DZIĘKUJĘ ZA UWAGĘ

I/O (STRUMIENIE, PLIKI, ...)

ZAGADNIENIA:

- pakiet java.io,
- strumienie bajtowe,
- strumienie znakowe,
- strumienie binarne I serializacja,
- operacje na plikach.

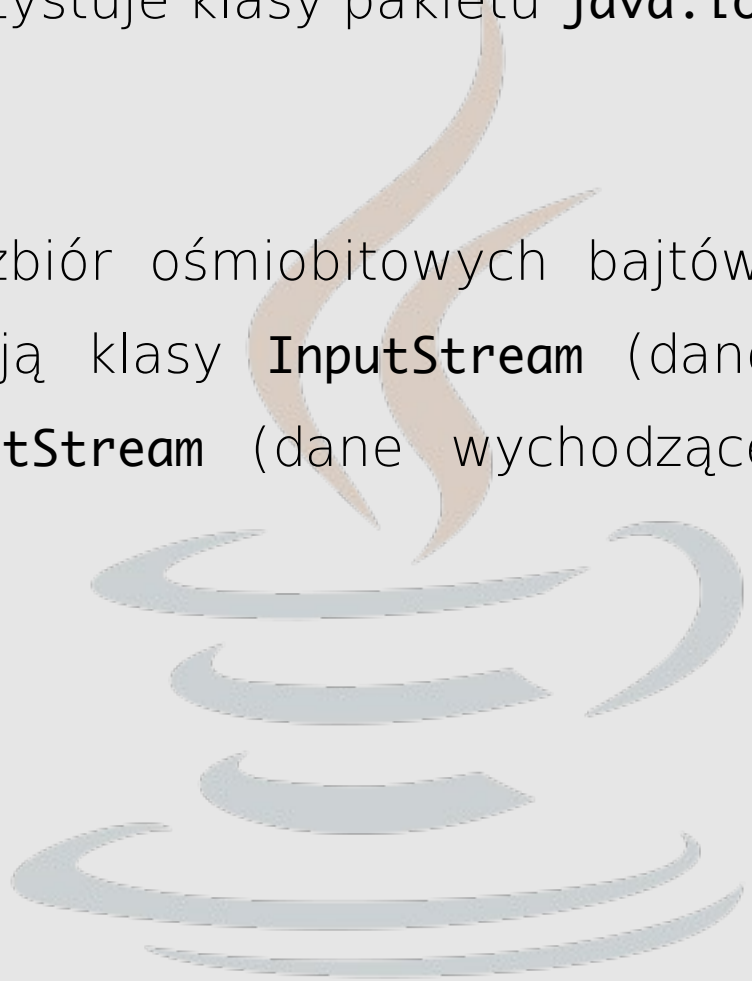
MATERIAŁY:

http://docs.oracle.com/javase/tutorial/essential/io/

STRUMIENIE BAJTOWE

Większość operacji wejścia/wyjścia wykorzystuje klasy pakietu **java.io**.

Strumienie bajtowe traktują dane jak zbiór ośmiobitowych bajtów. Wszystke strumienie bajtowe rozszerzają klasy **InputStream** (dane przychodzące do programu) lub **OutputStream** (dane wychodzące z programu).



STRUMIENIE BAJTOWE

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
```

STRUMIENIE BAJTOWE

```
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

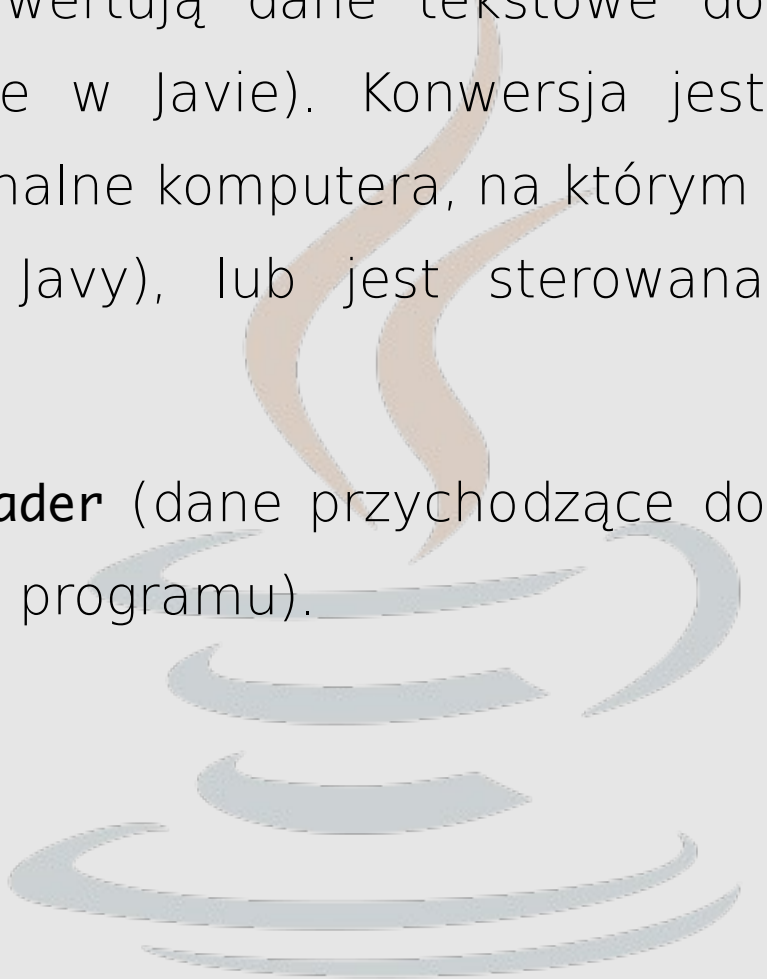
Strumienie zawsze należy zamykać!

Strumienie bajtowe reprezentują "niskopoziomowy" dostęp do danych. Dlatego w konkretnych sytuacjach warto je zastąpić przez bardziej specjalistyczne rodzaje strumieni.

STRUMIENIE ZNAKOWE

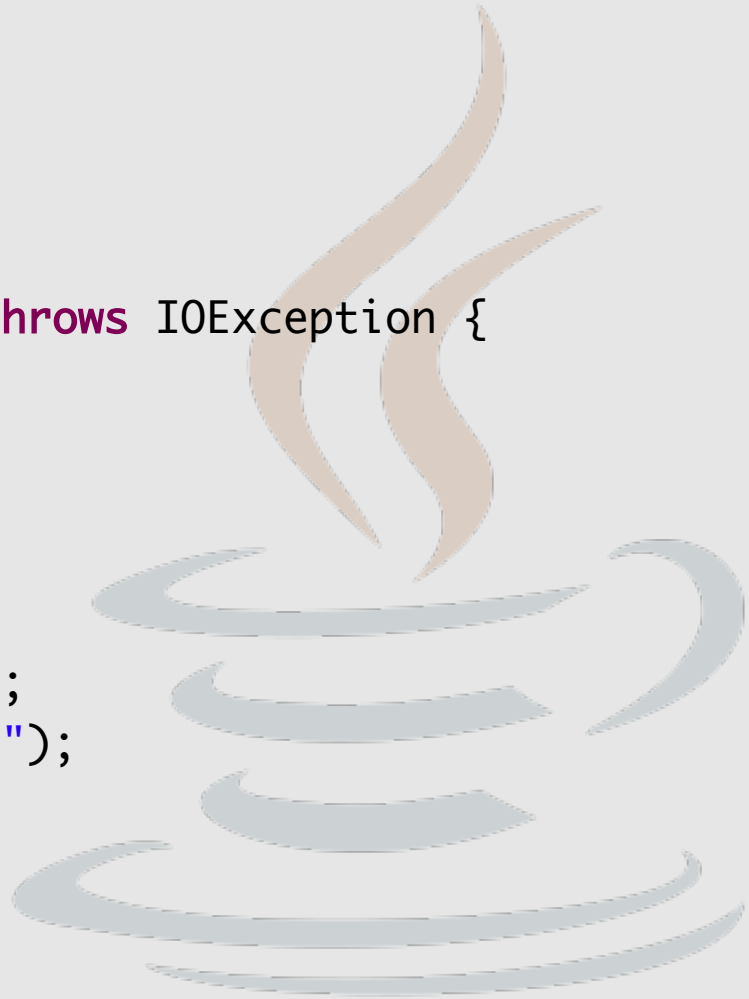
Strumienie znakowe automatycznie konwertują dane tekstowe do formatu Unicode (stosowanego natywnie w Javie). Konwersja jest dokonywana w oparciu o ustawienia regionalne komputera, na którym uruchomiono JVM (Wirtualną Maszynę Javy), lub jest sterowana "ręcznie" przez programistę.

Strumienie znakowe rozszerzają klasy **`Reader`** (dane przychodzące do programu) lub **`Writer`** (dane wychodzące z programu).



STRUMIENIE ZNAKOWE

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
```



STRUMIENIE ZNAKOWE

```
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Strumienie znakowe wykorzystuję do komunikacj strumienie bajtowe, a same zajmują się konwersją danych.

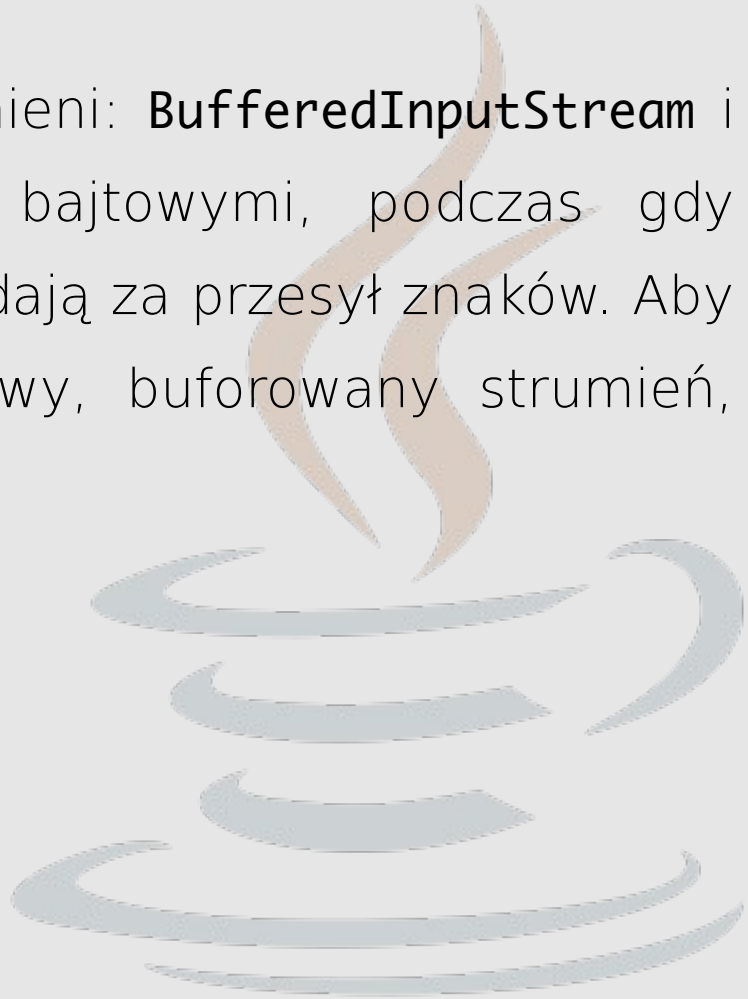
STRUMIENIE BUFOROWANE

Strumienie znakowe buforowane umożliwiają odczytywanie tekstu linia po linii:

```
BufferedReader in = null;
PrintWriter out = null;

try {
    in = new BufferedReader(new FileReader("input.txt"));
    out = new PrintWriter(new FileWriter("output.txt"));

    String l;
    while ((l = in.readLine()) != null) {
        out.println(l);
    }
} catch {... }
```

STRUMIENIE BUFOROWANE

Istnieją cztery klasy buforowanych strumieni: **BufferedInputStream** i **BufferedOutputStream** są strumieniami bajtowymi, podczas gdy **BufferedReader** i **BufferedWriter** odpowiadają za przesył znaków. Aby wymusić zapis danych poprzez wyjściowy, buforowany strumień, można użyć metody **flush()**.



SKANOWANIE

`Scanner` pozwala na przetwarzanie tokenów (domyślnie rozdzielonych przez `Character.isWhitespace(char c)`):

```
import java.io.*;
import java.util.Scanner;

public class ScanXan {
    public static void main(String[] args) throws IOException {

        Scanner s = null;
        try {
            s = new Scanner(new BufferedReader(
                                        new FileReader("input.txt")));
            while (s.hasNext()) {
                System.out.println(s.next());
            }
```

SKANOWANIE

```
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Obiekt należy zamknąć ze względu na strumień, z którym jest związany. Aby zmienić zachowanie obiektu Scanner, można skorzystać z metody: **useDelimiter()**. Przykładowo **s.useDelimiter(",\\s*");** zmienia znak rozdzielający na przecinek po którym następuje dowolna liczba "białych spacji".

FORMATOWANIE

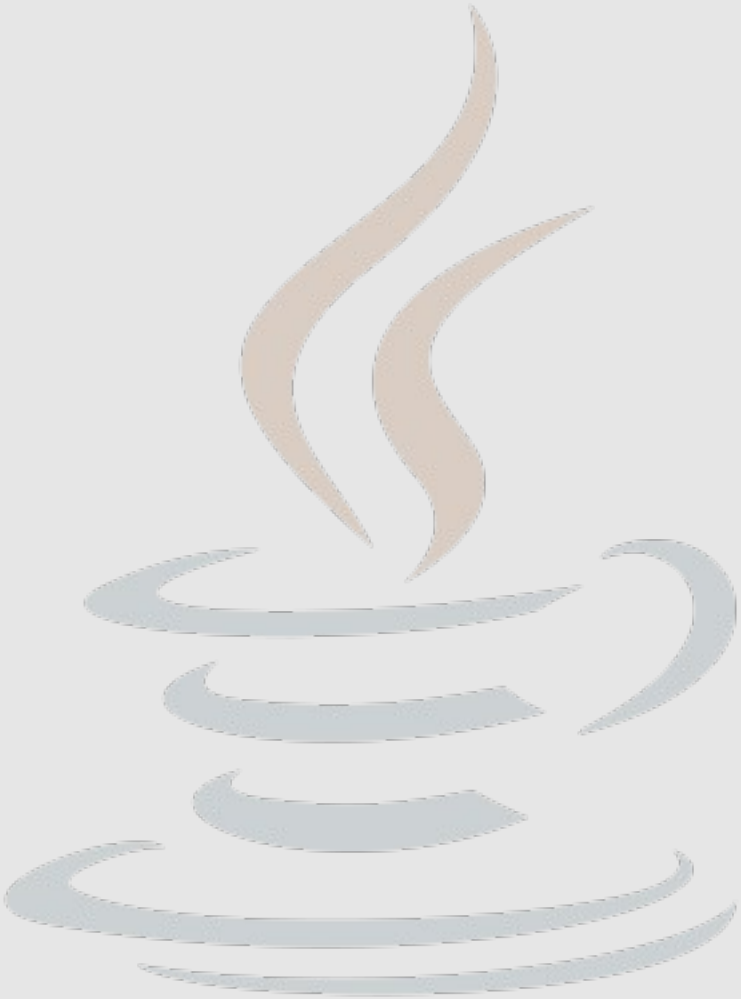
Wyjściowe strumienie znakowe umożliwiają podstawowe formatowanie danych za pomocą kilku odmian metody **print()** i **format()**.

```
double d = 2.0;
double s = Math.sqrt(2.0);
System.out.println("Pierwiastek z  " + d + " to " + s + ".");
Pierwiastek z  2.0 to 1.4142135623730951

System.out.format("Pierwiastek z  %f to %.4f\n", d, s);
Pierwiastek z  2,000000 to 1,4142

System.out.format(Locale.US, "Pierwiastek z  %.1f to %.4f\n", d, s);
System.out.printf(Locale.US, "Pierwiastek z  %.1f to %.4f\n", d, s);
Pierwiastek z  2.0 to 1.4142
```

Opis wszystkich możliwości formatowania jest opisany w dokumentacji klasy **java.util.Formatter**.

METODY WIELOARGUMENTOWE

```
public static void multiint(int... ints){
    for (int i=0; i<ints.length; i++)
        System.out.println(ints[i]);
    System.out.println();
    for(int i: ints)
        System.out.println(i);
}


public static void main(String[] args){
    multiint(123,34,65,76,44,11,0);
    multiint();
    multiint(12, 28);
}
```

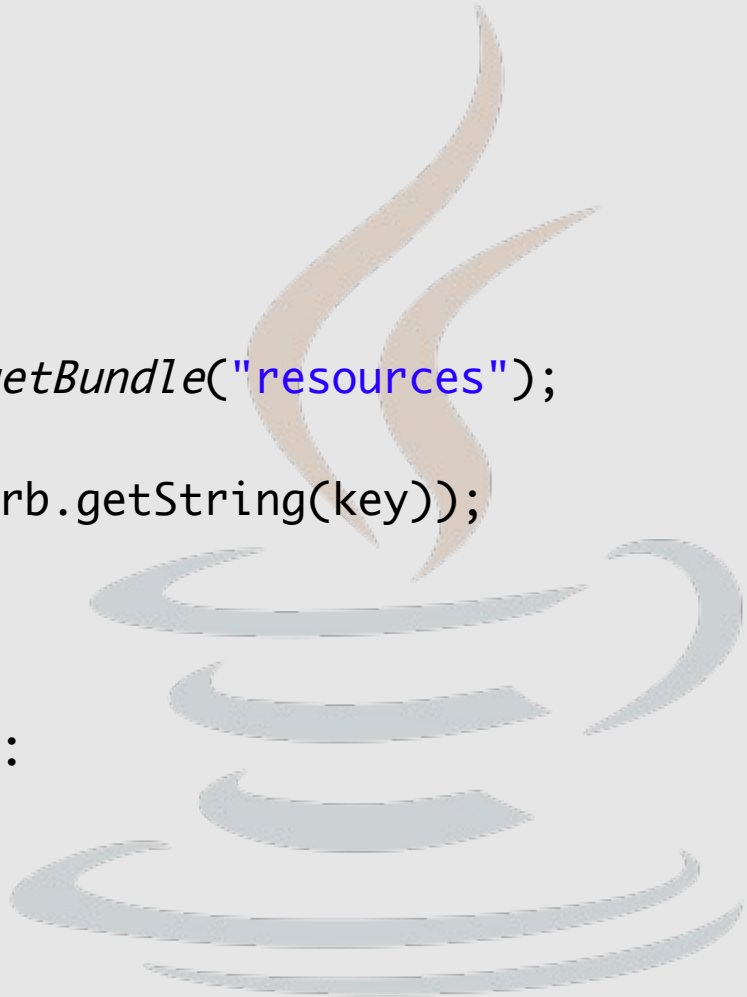


ZASOBY I LOKALIZACJA

```
import java.util.Locale;
import java.util.ResourceBundle;

public class LocalizationExample {

    public static void main(String[] args){
        ResourceBundle rb = ResourceBundle.getBundle("resources");
        for(String key: rb.keySet())
            System.out.println(key + ": " + rb.getString(key));
    }
}
```

Przykładowy plik `resources_pl.properties`:

```
KeyHello=witaj
KeyWorld=\u015bwiat
KeyKey=klucz
```



ZASOBY I LOKALIZACJA

Statuczna metoda *getBundle*("resources") jest równoważna wywołaniu

getBundle("resources", Locale.getDefault(), this.getClass().getClassLoader()).

i za pomocą bieżącego **ClassLoader**a poszukuje pliku o nazwie:

```
baseName +"_"+ language +"_"+ script +"_"+ country +"_"+ variant +".properties"
```

Konkretna nazwa pliku jest ustalana na podstawie ustawien regionalnych systemu operacyjnego (**Locale.getDefault()**), np. **resources_en_US_WINDOWS_VISTA.properties**. Metoda ta wczytuje pary (klucz,wartość). Dzięki temu można łatwo dostosować komunikaty, używane przez program do użytkownika.

CLASS LOADER

```
class NetworkClassLoader extends ClassLoader {
    String host;
    int port;

    public Class findClass(String name) {
        byte[] b = loadClassData(name);
        return defineClass(name, b, 0, b.length);
        }

        private byte[] loadClassData(String name) {
            // wczytywanie bytecode'u klasy z określonej
            // lokalizacji sieciowej
            . . .
        }
    }
}
```

STRUMIENIE BINARNE

Strumienie binarne pozwalają efektywniej zarządzać zasobami. Istnieją dwa podstawowe rodzaje strumieni:

- strumienie danych: **DataInputStream** i **DataOutputStream**:

```
DataOutputStream das = new DataOutputStream(System.out);
das.writeDouble(123.12);
das.writeUTF("Grzegrz\u00f3\u0142ka");
das.writeInt(12345);
das.close();
```

- strumienie obiektowe: **ObjectInputStream** i **ObjectOutputStream**:

```
ObjectOutputStream oos = new ObjectOutputStream(System.out);
oos.writeObject("Grzegrz\u00f3\u0142ka");
oos.close();
```

SERIALIZACJA

Podstawowym zastosowaniem strumieni obiektowych jest serializacja. Klasa wspierająca serializację musi implementować interfejs **Serializable**. Jeśli obiekty tej klasy wymagają specjalnego traktowania podczas serializacji należy zaimplementować metody:

```
private void writeObject(java.io.ObjectOutputStream out)
                                                    throws IOException;

private void readObject(java.io.ObjectInputStream in)
                                        throws IOException, ClassNotFoundException;
```

SERIALIZACJA

```
public class SerialisationTest implements Serializable{
    public int id;
    public String name;

    public SerialisationTest(int i, String s){
        this.id = i;
        this.name = s;
    }

    public static void main(String[] args) throws FileNotFoundException,
                                                   IOException, ClassNotFoundException{

        SerialisationTest st1 = new SerialisationTest(7, "Ala");

        ObjectOutputStream oos = new ObjectOutputStream(
                                     new FileOutputStream("output_object"));
        oos.writeObject(st1);
        oos.close();
```

SERIALIZACJA

```
        SerialisationTest st2;

        ObjectInputStream ois = new ObjectInputStream(new
                                        FileInputStream("output_object"));
        st2 = (SerialisationTest)ois.readObject();
        ois.close();
        System.out.println(st2.id + "\t" + st2.name);
    }
}
```

Dla obiektów typu JavaBeans istnieje także możliwość serializacji tekstowej (do plików w formacie XML) z wykorzystaniem klas `XMLEncoder` i `XMLDecoder`.

DZIĘKUJĘ ZA UWAGĘ

I/O (STRUMIENIE, PLIKI, ...)

ZAGADNIENIA:

- pakiet java.nio.file,
- operacje na plikach,
- swobodny dostęp do plików,
- katalogi,
- archiwa.
- jar i manifest.

MATERIAŁY:

http://docs.oracle.com/javase/tutorial/essential/io/

OPERACJE NA PLIKACH

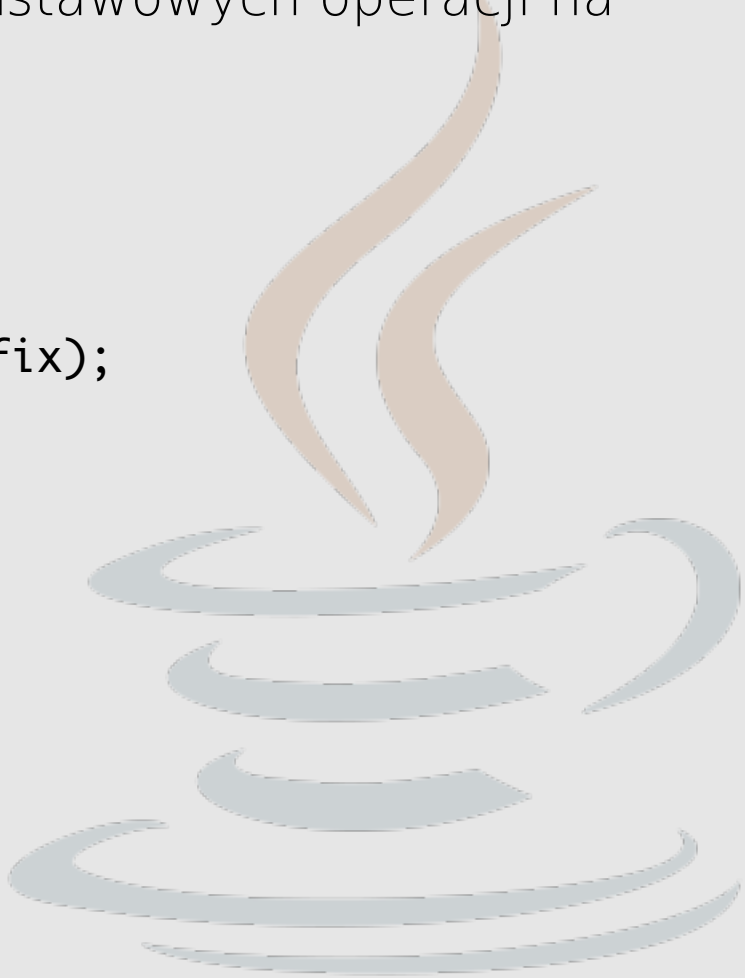
Podstawowa klasa reprezentująca plik to **File** (**java.io.File**).

```
import java.io.File;
public class FileTest {
    public static void listDirectory(File fdir){
        File[] fa = fdir.listFiles();
        if (fa==null) return;
        for(File f: fdir.listFiles()){
            if (f.isDirectory())
                listDirectory(f);
            System.out.println(f.getAbsolutePath());
        }
    }
    public static void main(String[] args){
        File f = new File("/");
        listDirectory(f);
    }
}
```

OPERACJE NA PLIKACH

Klasa **File** umożliwia wykonanie wielu podstawowych operacji na plikach np.:

- `exists();`
- `createNewFile();`
- `createTempFile(String prefix, String suffix);`
- `delete();`
- `deleteOnExit();`
- `renameTo(File dest);`
- `mkdirs();`
- `getParentFile();`
- `toPath();`



OPERACJE NA PLIKACH

Począwszy od Javy 1.7, bardziej zaawansowane operacje plikowe są dostępne poprzez klasy pakietu **java.nio.file**. Plik jest tutaj reprezentowany przez interfejs **Path**.

```
Path p1 = Paths.get("/tmp/file");

Path p2 = FileSystems.getDefault().getPath("/tmp/file");

Path p3 = Paths.get(URI.create("file:///tmp/file"));
```

lub

```
Path p = (new File("/tmp/file")).toPath();
```

OPERACJE NA PLIKACH

Podstawowe operacje na plikach wykonujemy używając metod statycznych klasy **Files**. Przykładowy dostęp atrybutów pliku:

```
Path file = ...;
BasicFileAttributes attr =
                    Files.readAttributes(file, BasicFileAttributes.class);
System.out.println("creationTime: " + attr.creationTime());
System.out.println("lastAccessTime: " + attr.lastAccessTime());
System.out.println("lastModifiedTime: " + attr.lastModifiedTime());
System.out.println("isDirectory: " + attr.isDirectory());
System.out.println("isOther: " + attr.isOther());
System.out.println("isRegularFile: " + attr.isRegularFile());
System.out.println("isSymbolicLink: " + attr.isSymbolicLink());
System.out.println("size: " + attr.size());
```

Można także używać **DosFileAttributes** lub **PosixFileAttributes**.

DOSTĘP DO PLIKÓW

Klasa posiada również metody odpowiadające za podstawowe operacje na plikach: **exists**, **move**, **delete**, **isReadable**, ...

Można ją wykorzystać także do dostępu do plików:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
BufferedWriter writer = null;
try {
    writer = Files.newBufferedWriter(file, charset);
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
} finally {
    if (writer != null) writer.close();
}
```

DOSTĘP DO PLIKÓW

Dostęp do niewielkich plików:

- odczyt

```
Path file = ...;
byte[] fileArray;
fileArray = Files.readAllBytes(file);
```

- zapis:

```
Path file = ...;
byte[] buf = ...;
Files.write(file, buf);
```

sposób działania metody write można ustawiś za pomocą dodatkowych flag. Domyślnie plik jest tworzony lub nadpisywany.

DOSTĘP DO PLIKÓW

Pliki i strumienie bajtowe:

```
Path fin=..., fout=...;
String line = null;
byte data[] = "Ala ma kota".getBytes();
try{
    InputStream in = Files.newInputStream(fin);
    BufferedReader reader =
                            new BufferedReader(new InputStreamReader(in));
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
    OutputStream out = new BufferedOutputStream(
                fout.newOutputStream(CREATE, APPEND));
    out.write(data, 0, data.length);
} catch (IOException x) {
    System.err.println(x);
}
```

DOSTĘP DO PLIKÓW - KANAŁY

Strumienie odczytują/zapisują "na raz" bajt lub znak. Kanały (Channels) odczytują/zapisują "na raz" bufor, co umożliwia swobodny dostęp do pliku.

```
Path file;
// domyślny kanał ma flagę READ
try (SeekableByteChannel sbc = Files.newByteChannel(file)) {
        ByteBuffer buf = ByteBuffer.allocate(10);
    // odczyt zgodny z domyślnym kodowaniem
    String encoding = System.getProperty("file.encoding");
    while (sbc.read(buf) > 0) {
        buf.rewind();
        System.out.print(Charset.forName(encoding).decode(buf));
        buf.flip();
    }
} catch (IOException x) { System.out.println("caught exception: " + x); }
```

SWOBODNY DOSTĘP DO PLIKÓW

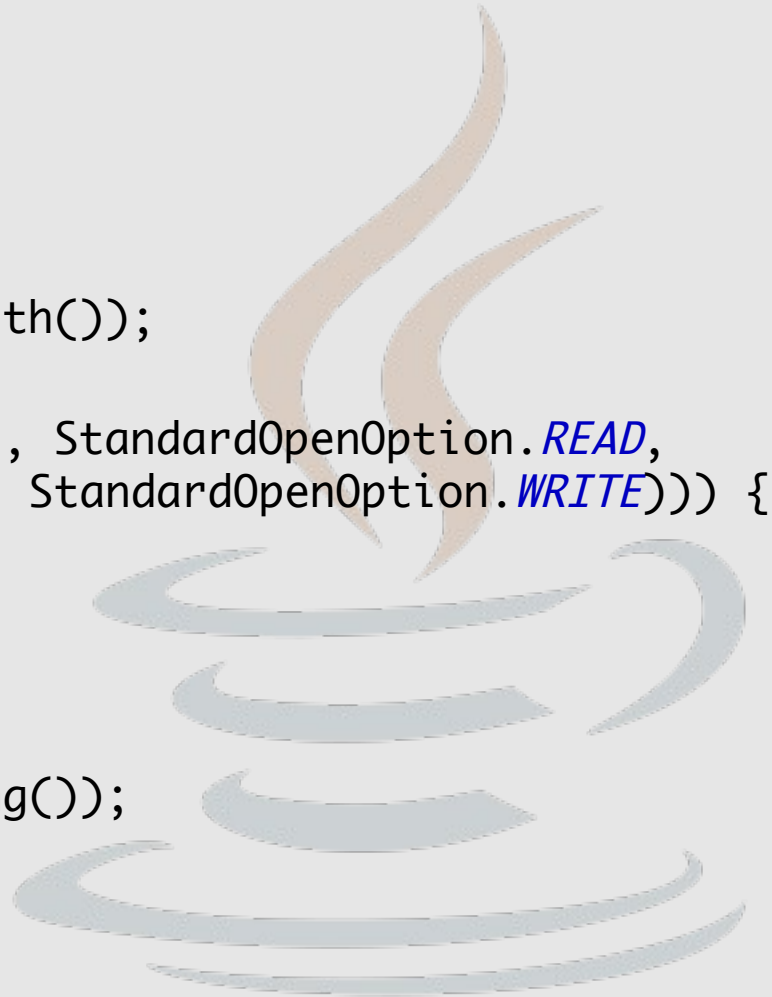
Interfejs **SeekableByteChannel** udostępnia metody pozwalające na swobodny dostęp do plików:

- **position()** - zwraca bieżącą pozycję,
- **position(long)** - ustawia pozycję,
- **read(ByteBuffer)** - odczytuje dane z kanału,
- **write(ByteBuffer)** - zapisuje dane do kanału,
- **truncate(long)** - nadpisuje plik połączony do kanału.

Kanał utworzony metodą **Path.newByteChannel()**. można zrzutować na **FileChannel** dający więcej możliwości (np. odwzorowania pliku w pamięci w celu np. przyspieszenia dostępu)

SWOBODNY DOSTĘP DO PLIKÓW

```
Path file = Paths.get("plik.txt");
String s = "tu bylem!";
byte data[] = s.getBytes();

ByteBuffer out = ByteBuffer.wrap(data);
ByteBuffer copy = ByteBuffer.allocate(s.length());

try (FileChannel fc = (FileChannel.open(file, StandardOpenOption.READ,
                                        StandardOpenOption.WRITE))) {
    // Odczyt poczatkowych 10 bajtów z pliku
    int nread;
    do {
        nread = fc.read(copy);
    } while (nread != -1 && copy.hasRemaining());
```



SWOBODNY DOSTĘP DO PLIKÓW

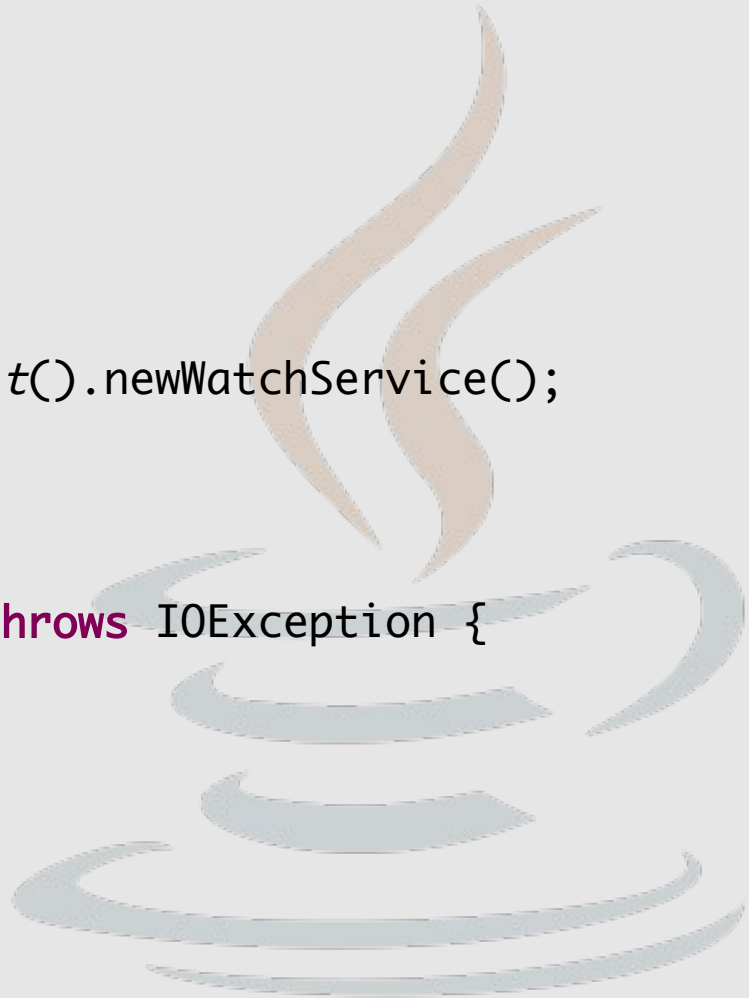
```
    // zapis "tu bylem!" na początku pliku (nadpisuje pierwsze 10 bajtów)
    fc.position(0);
    while (out.hasRemaining())
        fc.write(out);
    out.rewind();
    // przesuniecie na koniec pliku.
    fc.position(fc.size() - 1);
    // zapisanie kopii na końcu pliku
    copy.flip();
    while (copy.hasRemaining())
        fc.write(copy);
    // zapisanie "tu bylem!" na końcu pliku
    while (out.hasRemaining())
        fc.write(out);
    } catch (IOException x) {
        System.out.println("I/O Exception: " + x);
    }
}
```

KATALOGI

```
Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
for (Path name: dirs)
    System.err.println(name);

Path dir = ...; // listujemy tylko pliki *.java, *.class i *.jar
try (DirectoryStream<Path> stream =
     Files.newDirectoryStream(dir, "*.{java,class,jar}")) {
    for (Path entry: stream)
        System.out.println(entry.getFileName());
} catch (IOException x) { System.err.println(x); }

Path dir = ...;
Files.createDirectory(dir); // tworzymy katalog
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rwxr-x---");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.createDirectory(file, attr); // tworzymy katalog i nadajemy mu prawa
```

MONITOROWANIE KATALOGÓW

```
public class WatchDir {

private final WatchService watcher;
private final Path dir = Paths.get(".");

public WatchDir() throws IOException {
    this.watcher = FileSystems.getDefault().newWatchService();
    dir.register(watcher, ENTRY_CREATE);
}

public static void main(String[] args) throws IOException {
    // oczekujemy na zdarzenia
    new WatchDir().processEvents();
}
```



MONITOROWANIE KATALOGÓW

```
    void processEvents() {
        while(true){
            // wait for key to be signalled
            WatchKey key;
            try {
                key = watcher.take();
            } catch (InterruptedException x) { return; }
            for (WatchEvent<?> event: key.pollEvents()) {
                WatchEvent<Path> ev = (WatchEvent<Path>) event;
                Path name = ev.context();
                Path child = this.dir.resolve(name);

                System.out.format("%s: %s\n", event.kind().name(), child);
            }
            key.reset();
        }
    }
}
```

ARCHIWA

Java implementuje strumienie realizujące kompresję danych:

```
GZIPOutputStream gos = new GZIPOutputStream(
                                              new FileOutputStream("plik.gz"));
for (int i = 0; i < 10000; i++)
    gos.write("Ala ma kota".getBytes());
gos.close();

ZipOutputStream zos = new ZipOutputStream(
                                            new FileOutputStream("plik.zip"));
for(int i=0; i<5; i++){
    ZipEntry ze = new ZipEntry("plik" + i);
    zos.putNextEntry(ze);
    for(int j=0; j< 1000; j++)
        zos.write("Ala ma kota".getBytes());
    zos.closeEntry();
}
zos.close();
```

ARCHIWA JAR

Java wyróżnia także szczególny rodzaj archiwym ZIP: JAR (JarOutputStream, JarInputStream). Archiwa JAR zawierają pliki klas wraz z dodatkowymi zasobami potrzebnymi do działania aplikacji. Podstawowe zalety dystrybucji programów w postaci plików jar to:

- bezpieczeństwo: archiwa mogą być cyfrowo podpisywane,
- kompresja: skrócenie czasu ładowania apletu lub aplikacji,
- zarządzanie zawartością archiwów z poziomu języka Java,
- zarządzanie wersjami na poziomie pakietów oraz archiwów (Package Sealing, Package Versioning),
- przenośność.

JAR

Archiwum jar tworzy sie używając komendy jar, np:

```
jar cf archiwum.jar klasa1.class klasa2.class ...
```

Użyte opcje:

- c – tworzenie pliku (create),
- `f` – zawartość archiwum zostanie zapisana do pliku archiwum.jar zamiast do standardowego wyjscia (stdout);

Inne najczęściej używane opcje:

- `m` – do archiwum zostanie dołączony plik manifest z określonej lokalizacji, np: jar cmf plik_manifest archiwum.jar *,
- `C` – zmiana katologu w trakcie działania archiwizatora, np: jar cf ImageAudio.jar -C images * -C audio *.

MANIFEST

W archiwum jar znajduje się katalog **META-INF** a w nim plik **MANIFEST.MF** zawierający dodatkowe informacje o archiwum. Przykładowa zawartość:

```
Manifest-Version: 1.0
Created-By: 1.5.0-b64 (Sun Microsystems Inc.)
Ant-Version: Apache Ant 1.6.5
Main-Class: pl.edu.uj.if.wyklady.java.Wyklad06
```

mówi, że po uruchomieniu archiwum wykonana zostanie metoda **main(String[] args)** zawarta w klasie **Wyklad06** znajdującej się w pakiecie **pl.edu.uj.if.wyklady.java**.

Uruchomienie pliku jar:

```
java -jar archiwum.jar
```

DZIĘKUJĘ ZA UWAGĘ

WSPÓŁBIEŻNOŚĆ

ZAGADNIENIA:

- procesy,
- wątki,
- synchronizacja,
- synchronizacja w Java 5.0
- blokady, egzekutory, zbiory wątków

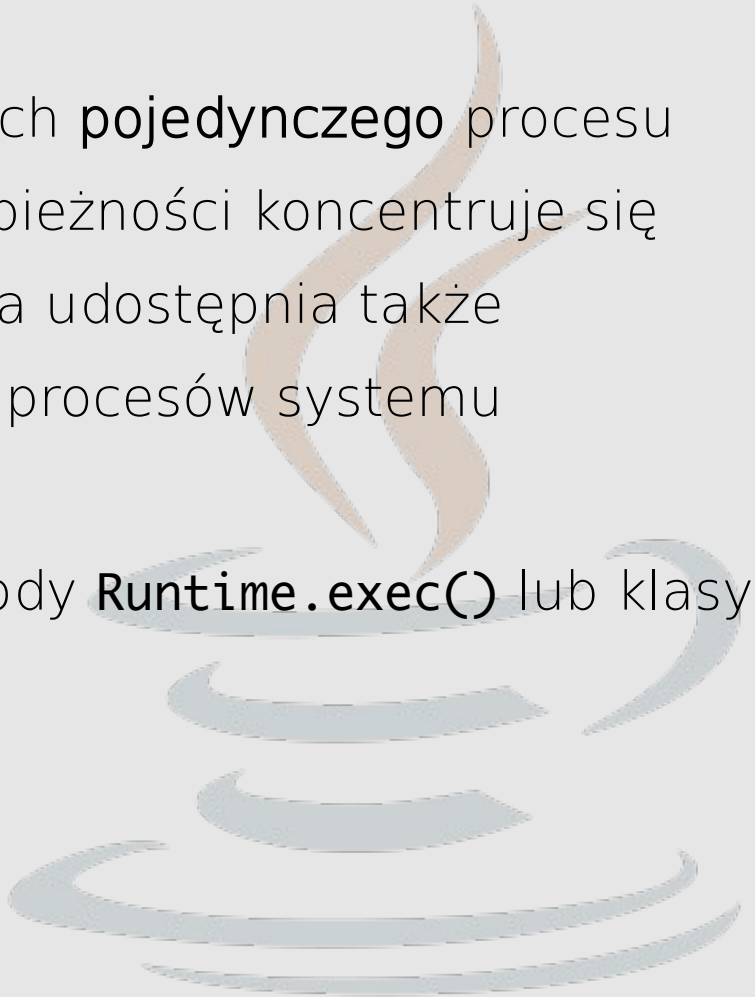
MATERIAŁY:

http://docs.oracle.com/javase/tutorial/essential/concurrency/

PROCESY I WĄTKI

Program w Javie uruchamiany jest w ramach **pojedynczego** procesu JVM. Z tego powodu implementacja współbieżności koncentruje się głównie na obsłudze wątków. Niemniej Java udostępnia także mechanizmy umożliwiające uruchamianie procesów systemu operacyjnego.

Proces można stworzyć korzystając z metody `Runtime.exec()` lub klasy `ProcessBuilder` z pakietu `java.lang`.

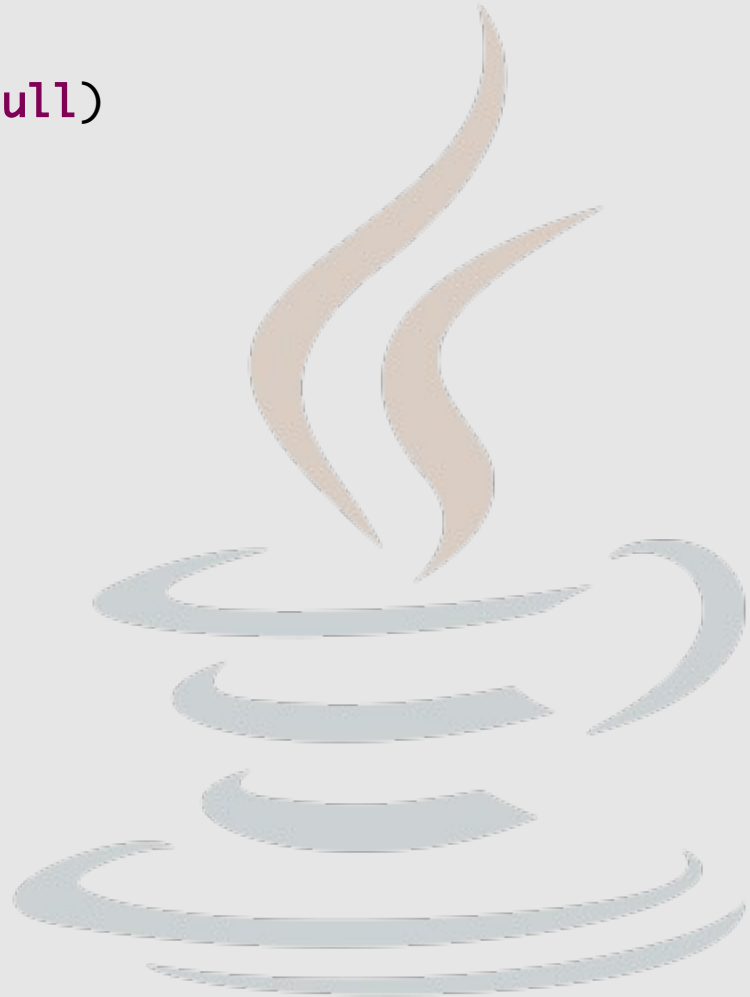


PROCESY

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ProcessExample {

    public static void main(String[] args) {
        try {
            String s;
            Process ps = Runtime.getRuntime().exec("ls -l");
            BufferedReader bri = new BufferedReader(new InputStreamReader(
                                                    ps.getInputStream()));
            BufferedReader bre = new BufferedReader(new InputStreamReader(
                                                    ps.getErrorStream()));
            while ((s = bri.readLine()) != null)
                System.out.println(s);
            bri.close();
```

PROCESY

```
            while ((s = bre.readLine()) != null)
                System.out.println(s);
            bre.close();
            ps.waitFor();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Gotowe.");
    }
}
```



PROCESY

Klasa **ProcessBuilder** pozwala precyzyjniej określić środowisko, w którym działa proces:

```
ProcessBuilder builder = new ProcessBuilder("ls", "-l");
builder.directory(new File("."));
builder.redirectErrorStream(true);
builder.redirectOutput(Redirect.INHERIT);
Process ps;
try {
    ps = builder.start();
    ps.waitFor();
} catch (IOException | InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Gotowe.");
```

Warto zajrzeć na: http://www.rgagnon.com/javadetails/java-0014.html

WĄTKI

Istnieją dwa podstawowe sposoby tworzenia wątków. Pierwszy polega na rozszerzeniu klasy **java.lang.Thread**:

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Witam z wątku");
    }

    public static void main(String args[]) {
        Thread t = new HelloThread();
        t.start();
        System.out.println("Witam z programu");
    }
}
```

WĄTKI

Drugi opiera się na skonstruowaniu wątku w oparciu o klasę implementującą interfejs **java.lang.Runnable**:

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Witam z watku");
    }

    public static void main(String args[]) {
        Thread t = new Thread(new HelloRunnable());
        t.start();
        System.out.println("Witam z programu");
    }
}
```

Ten przypadek jest ogólniejszy (i zalecany), gdyż klasa implementująca wątek może rozszeżać inną klasę.

KLASA THREAD

Wątki mogą być wstrzymywanie oraz wzbudzane/przerywane:

```
public class InterruptExample implements Runnable{
    public void run() {
        try {
            Thread.sleep(10000); // wstrzymanie na 10 sek.
        } catch (InterruptedException e) {
            System.out.println("interupted");
        }
    }
    public static void main(String[] args) throws InterruptedException{
        Thread t = new Thread(new InterruptExample());
        t.start();
        Thread.sleep(5000);
        System.out.println("budzenie");
        t.interrupt();
    }
}
```

KLASA THREAD

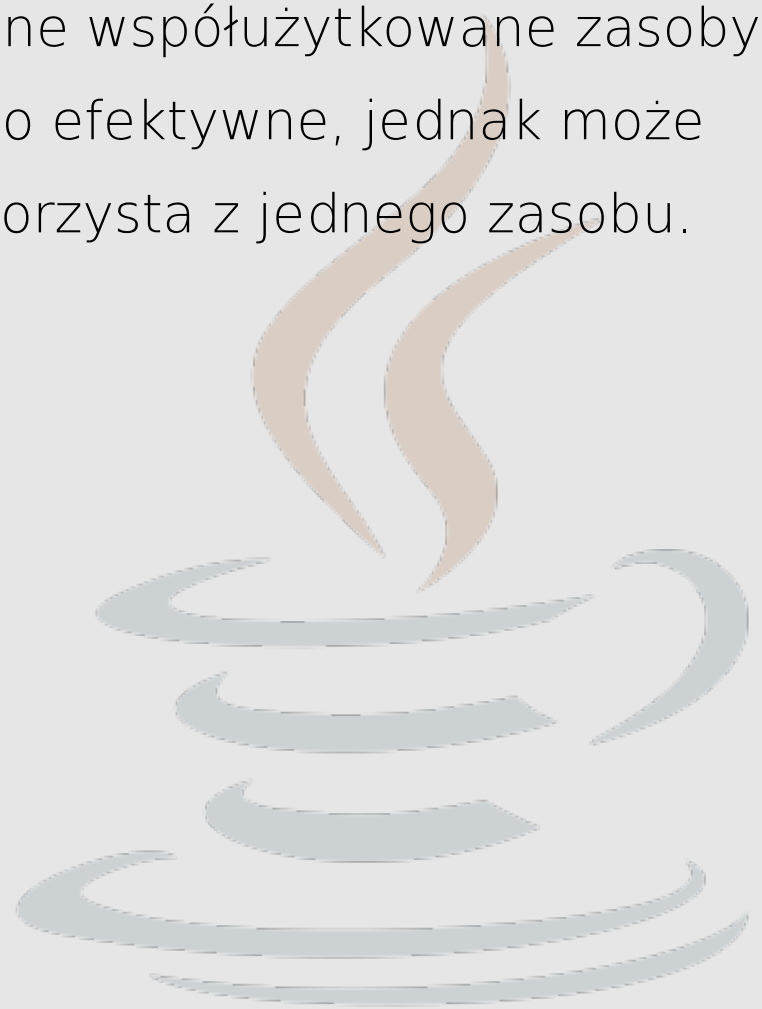
Można również poczekać na zakończenie wskazanego wątku.

```
public class JoinExample implements Runnable{
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            return;
        }
        System.out.println("watek");
    }
    public static void main(String[] args) throws InterruptedException{
        Thread t = new Thread(new JoinExample());
        t.start();
        t.join(); // czekamy na zakonczenie t
        System.out.println("teraz ja");
    }
}
```

SYNCHRONIZACJA

Wątki mogą się komunikować przez dowolne współużytkowane zasoby (np. referencje do obiektów). Jest to bardzo efektywne, jednak może powodować problemy, gdy kilka wątków korzysta z jednego zasobu.

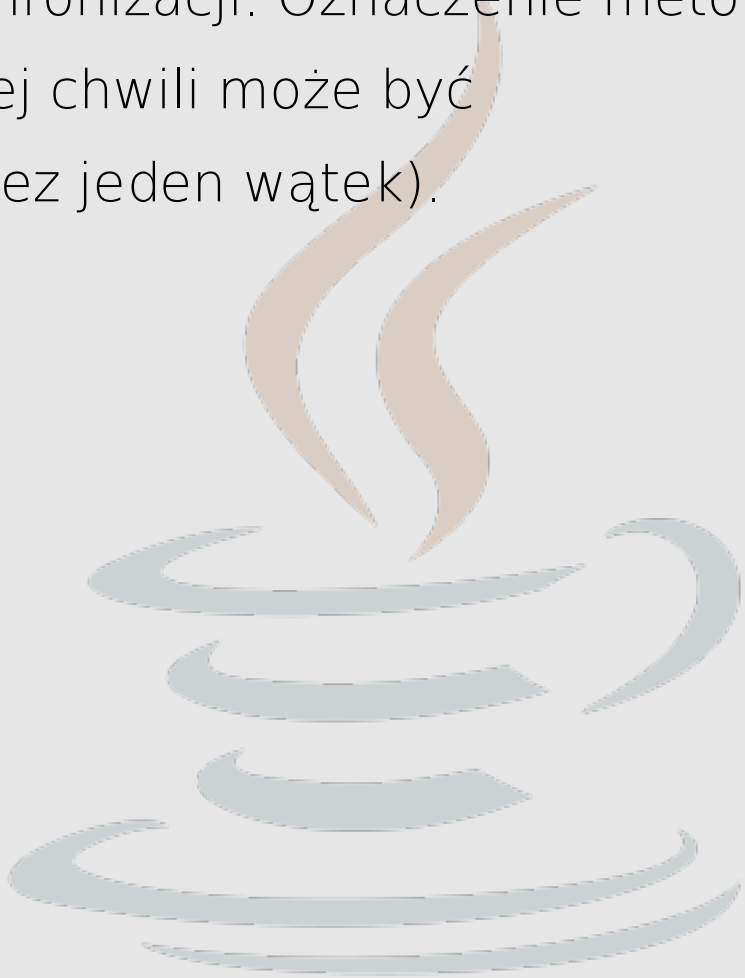
```
class Counter {
    private int c = 0;
    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```



SYNCHRONIZACJA METOD

Problem ten rozwiązuje się używając synchronizacji. Oznaczenie metod słowem **synchronized** powoduje, że w danej chwili może być wykonywana tylko jedna z nich (i tylko przez jeden wątek).

```
class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```



BLOKADA WEWNĘTZNA

Każdy obiekt posiada powiązaną z nim blokadę wewnętrzna (intrinsic lock). Jeśli wątek chce uzyskać wyłączny dostęp do obiektu (lub jego atrybutów) może skorzystać z tej blokady.

```
public void addName(String s) {
    synchronized(this) {
        name = s;
        counter++;
    }
    nameList.add(name);
}
```

Wątek musi jednoznacznie wskazywać obiekt, z którym jest związana używana przez niego blokada. Taki tyb klokad nazywamy synchronizacją na poziomie instrukcji (synchronized statements)

BLOKADA DROBNOZIARNISTA

Dostęp do c1 i c2 musi być synchronizowany niezależnie - nie chcemy blokować na raz obu liczników.

```
public class FineGrainedLockEx {
    private long c1 = 0, c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

OPERACJE ATOMOWE

Typowo dostęp do zmienne/referencji nie jest realizowany jako pojedyncza operacja. Aby takie operacje (odczytu/zapisu zmiennej) były atomowe należy oznaczyć ją słowem **volatile**.

UWAGA:

```
    private volatile long c1 = 0;

    public void inc1() {
            c1 += 5; // nie jest atomowa bo atomowy jest odczyt i zapis
                     // wartosci a nie jej zwiekszanie
        }
    }
```

więcej:

http://www.javamex.com/tutorials/synchronization_volatile.shtml

TYPOWE PROBLEMY

W programach wielowątkowych występuje kilka rodzajów problemów, które mogą powodować niewłaściwe działanie programów:

- zakleszczenia (deadlock) – wątek blokują wzajemnie zasoby potrzebne im do dalszego działania (pięciu filozofów).
- zagłodzenia (starvation) - jeden wątek przez cały blokuje zasób potrzebny innym wątkom.
- livelock – "odwrotność" deadlocka – wątek reaguje na zachowanie drugiego wątku, które jest reakcją na zachowanie pierwszego wątku.

ZAKLESZCZENIE

```
public class Deadlock {

    static class Worker {
        public String name;
        public Worker(String name) {
            this.name = name;
        }
        public synchronized void doWork(Worker w){
            System.out.println(this.name + " pracuje z " + w.name);
            try {
                Thread.sleep(1000); // pracujemy
            } catch (InterruptedException e) { }
            w.release();
        }
        public synchronized void release() {
            System.out.println(this.name + " jest znowu gotowy");
        }
    }
```

ZAKLESZCZENIE

```
    public static void main(String[] args) {
        final Worker w1 = new Worker("w1");
        final Worker w2 = new Worker("w2");

        new Thread(new Runnable() {
            public void run() { w1.doWork(w2); }
        }).start();

        new Thread(new Runnable() {
            public void run() { w2.doWork(w1); }
        }).start();
    }
}
```

Obaj workerzy zaczną ze sobą pracować. Blokada nastąpi na metodzie `release()` w obu obiektach.

WAIT / NOTIFY

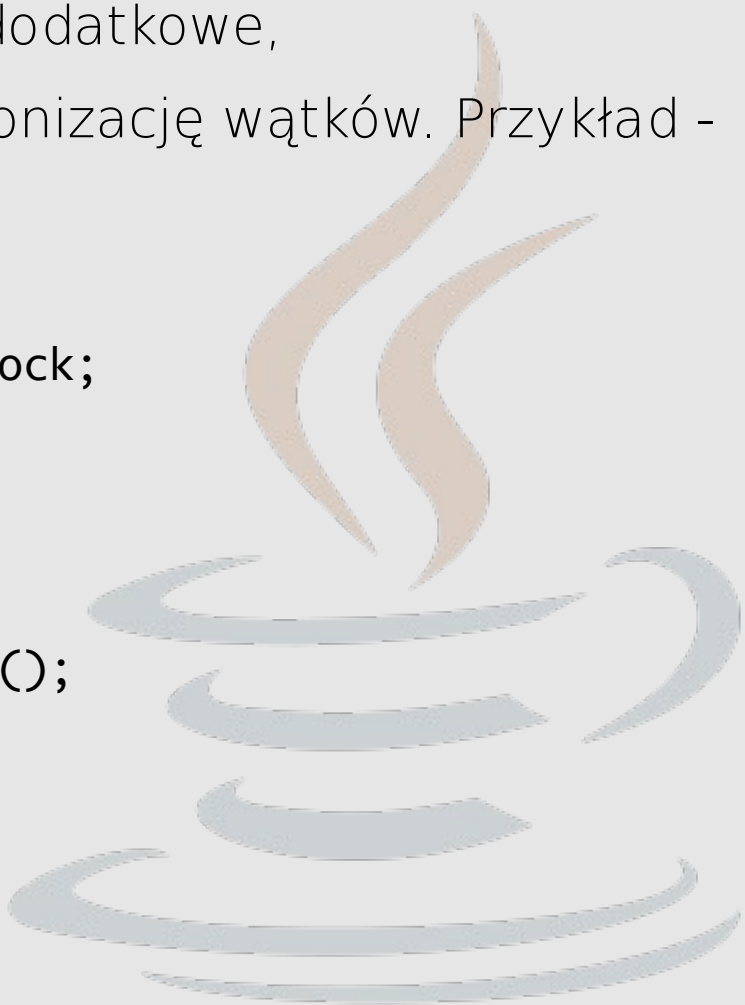
Często wątek musi poczekać, aż inny wąteg wykona określoną część swoich zadań, np. jeden wątek oblicza wyniki, a drugi je sukcesywnie wypisuje na ekranie.

```
public synchronized consume() {
    while(!available) {
        try {
            wait();  // wstrzymuje dzialanie watku i zwalnia blokadę
        } catch (InterruptedException e) {}
    }
    System.out.println("Skonsumowane");
    available = false;
}
public synchronized produce() {
    doProduce();
    available = true;
    notifyAll(); // powiadamia (budzi) wszystkie czekajace watki
}
```

SYNCHRONIZACJA W JRE 5.0

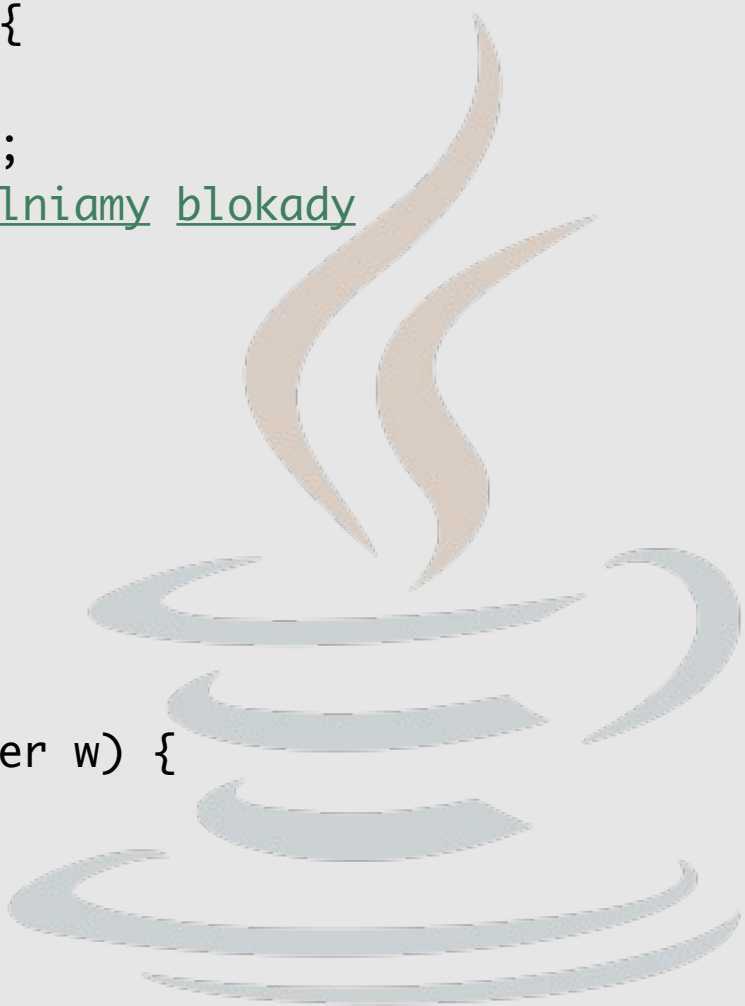
Począwszy od wersji 5.0 Java udostępnia dodatkowe, wysokopoziomowe API ułatwiające synchronizację wątków. Przykład - blokady:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockObjects {

    static class Worker {
        public Lock lock = new ReentrantLock();
        public String name;

        public Worker(String name) {
            this.name = name;
        }
```



BLOKADY

```
public boolean tryWorking(Worker w) {
    boolean myLock = lock.tryLock();
    boolean wLock = w.lock.tryLock();
    if (!(myLock && wLock)) { // zwalniamy blokady
        if (myLock)
            lock.unlock();
        if (wLock)
            w.lock.unlock();
    }
    return myLock && wLock;
}

public synchronized void doWork(Worker w) {
    boolean done = false;
```



BLOKADY

```
        while (!done) {
            if (tryWorking(w)) {
                System.out.println(name + ": pracuje z " + w.name);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) { }
                w.release();
                this.lock.unlock();
                done = true;
            } else {
               System.out.println(name+": jestem zajety wiec czekam");
                try {
                    wait();
                } catch (InterruptedException e) { }
                System.out.println(this.name + ": probuje znowu");
            }
        }
    }
```

BLOKADY

```
        public synchronized void release() {
            System.out.println(this.name + ": jestem znowu gotowy");
            this.lock.unlock();
            notifyAll();
        }
    }
    public static void main(String[] args) {
        final Worker w1 = new Worker("w1");
        final Worker w2 = new Worker("w2");
        new Thread(new Runnable() {
                        public void run() { w1.doWork(w2); }
                }).start();
        new Thread(new Runnable() {
                        public void run() { w2.doWork(w1); }
                }).start();
    }
}
```

EXECUTORS

Egzekutory oddzielają wątek od zarządzania nim

```
Executor exec = ...;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

Executor jest rozszerzany przez ExecutorService, który dodatkowo umożliwia uruchamianie wątków Callable, które mogą zwracać wynik działania.

Kolejnym rodzajem egzekutorów są tzw. Thread Pools – zbiory, wspólnie zarządzanych wątków. W JRE 7.0 wprowadzono ForkJoinPool, przeznaczony do wieloprocesorowych obliczeń równoległych.

EXECUTORS

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class Fibonacci extends RecursiveTask<Integer>{

    final int n;

    public Fibonacci(int n){
        this.n = n;
    }

    protected Integer compute() {
        if (n>2){
            doRecursion();
        }else{
            return 1;
    }
```

EXECUTORS

```
    public doRecursion(){
        Fibonacci f1 = new Fibonacci(n-1);
        Fibonacci f2 = new Fibonacci(n-2);
        f2.fork();
        return f1.compute()+f2.join();
    }

    public static void main(String[] args) {
        Fibonacci f = new Fibonacci(35);
        ForkJoinPool pool = new ForkJoinPool();
        System.out.println(pool.invoke(f));
    }
}
```

Ponadto wprowadzono także współbieżny generator liczb losowych:

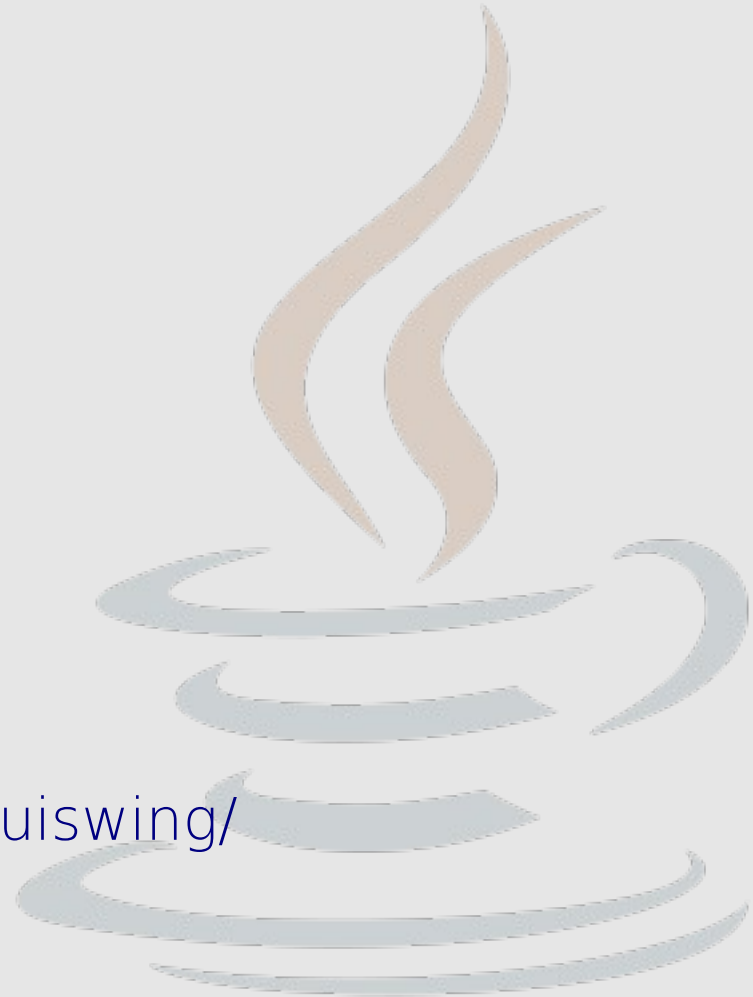
ThreadLocalRandom,

DZIĘKUJĘ ZA UWAGĘ

SWING

ZAGADNIENIA:

- wprowadzenie,
- kontenery I komponenty,
- LayoutManager,
- komponenty tekstowe.

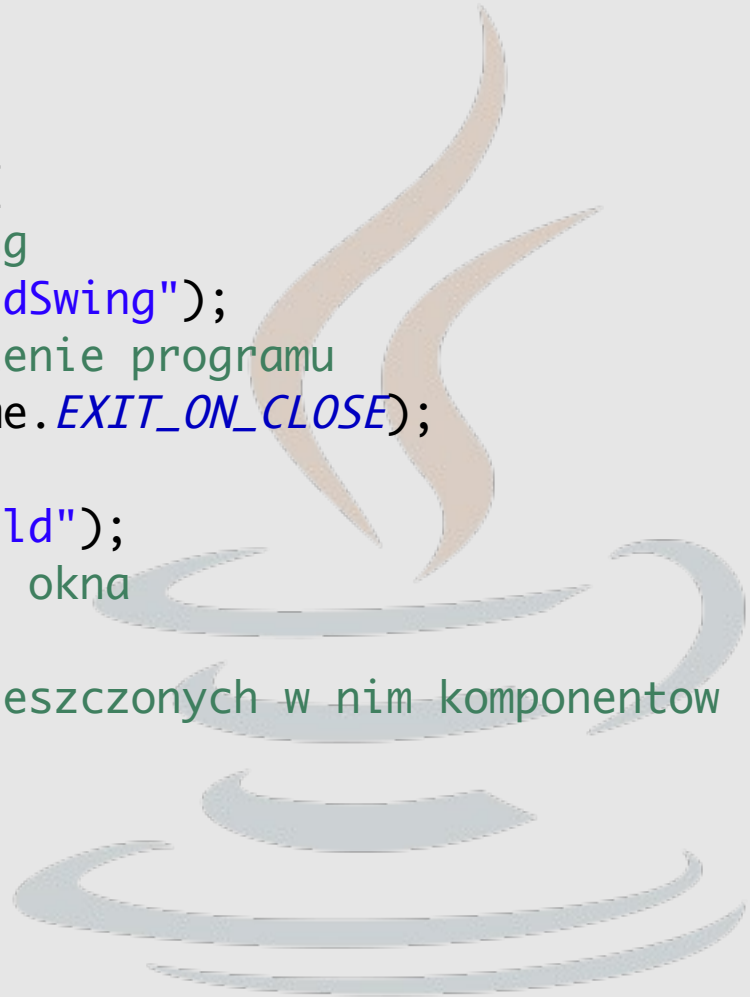


MATERIAŁY:

http://docs.oracle.com/javase/tutorial/uiswing/

SWING

```
import javax.swing.*;

public class HelloWorldSwing {
    private static void createAndShowGUI() {
        // nowe okno o tytule HelloWorldSwing
        JFrame frame = new JFrame("HelloWorldSwing");
        // zamkniecie okna spowoduje zakonczenie programu
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // nowy napis
        JLabel label = new JLabel("Hello World");
        // napis jest dodawany do zawartosci okna
        frame.getContentPane().add(label);
        // dopasowanie rozmiarow okna do umieszczonych w nim komponentow
        frame.pack();
        // wyswietlenie okna
        frame.setVisible(true);
    }
```



SWING

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

        // stworzenie nowego watku, w ktorym zostanie 'uruchomione' okno
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

KONTENERY

Każda aplikacja wykorzystująca Swing użuwa co najmniej jeden kontener najwyższego poziomu.

- JFrame,
- JDialog,
- JApplet.

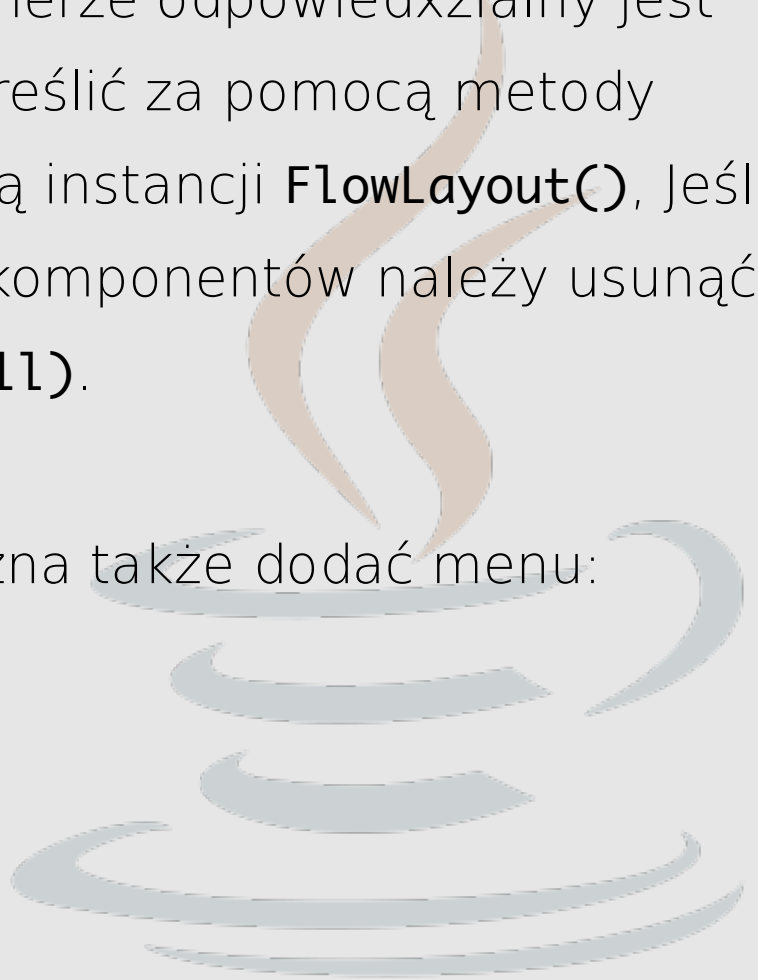
W takim kontenerze umieszcza się kolejne kontenery lub komponenty:

```
frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);

// A tutaj tworzymy panel (kontener) dodajemy do niego komponenty
// a nastepnie umieszczamy go w kontenerze najwyzszego poziomu (frame)
JPanel contentPane = new JPanel(new BorderLayout());
contentPane.setBorder(someBorder);
contentPane.add(someComponent, BorderLayout.CENTER);
contentPane.add(anotherComponent, BorderLayout.PAGE_END);
frame.getContentPane().add(contentPane)
```

KONTENERY

Za rozmieszczenie komponentów w kontenerze odpowiedxzialny jest obiekt typu `LayoutManager`. Możemy go określić za pomocą metody `setLayout()`. Domyślnie kontenery używają instancji `FlowLayout()`, Jeśli chcemy "samodzielnie" określać pozycje komponentów należy usunąć LayoutManagera: `container.setLayout(null)`.

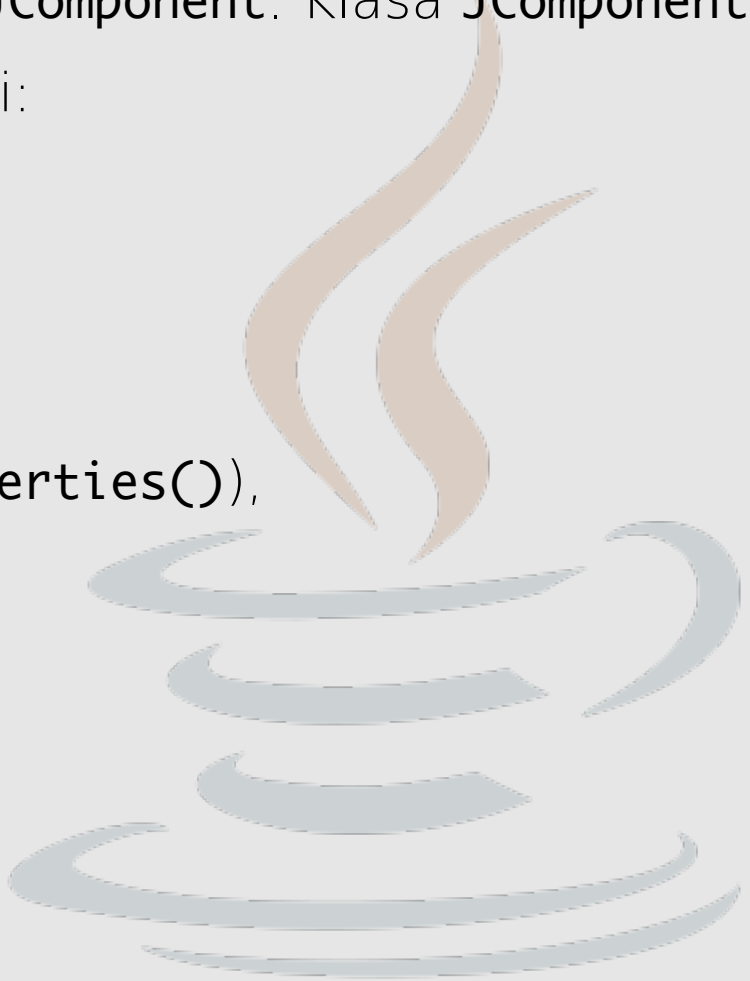
Do kontenerów najwyższego poziomu można także dodać menu: `frame.setJMenuBar()`.



KOMPONENTY

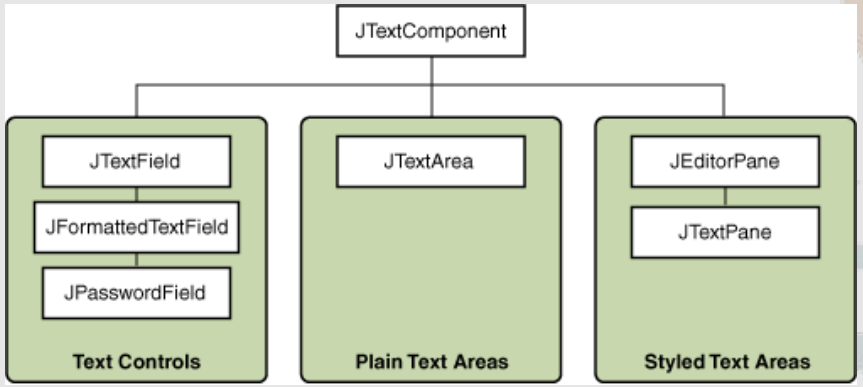
Wszystkie komponenty rozszerzają klasę `JComponent`. Klasa `JComponent` implementuje następujące funkcjonalności:

- podpowiedzi (`setToolTipText()`),
- ramki (`setBorder()`),
- styl (`UIManager.setLookAndFeel()`),
- dodatkowe właściwości (`setClientProperties()`),
- rozmiary (layout)
- przystępność (accessibility)
- przeciągnij i upuść,
- podwójne buforowanie,
- wiązanie klawiszy.



KOMPONENTY TEKSTOWE

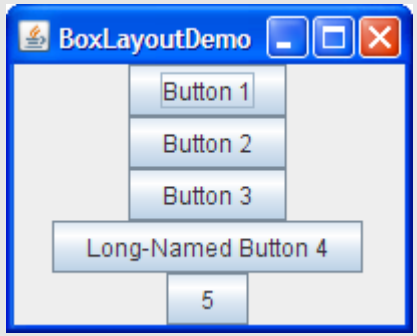
Komponenty tekstowe (**`JTextComponent`**) dzielą się na trzy kategorie:

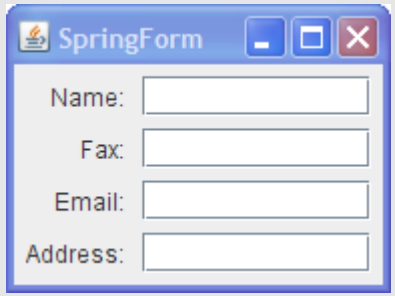


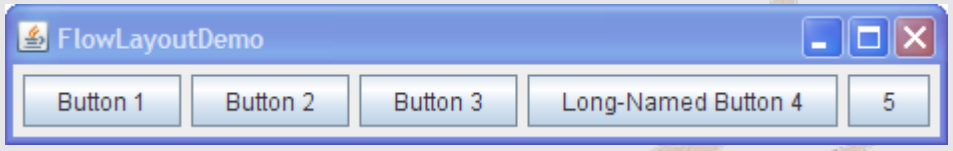
KOMPONENTY

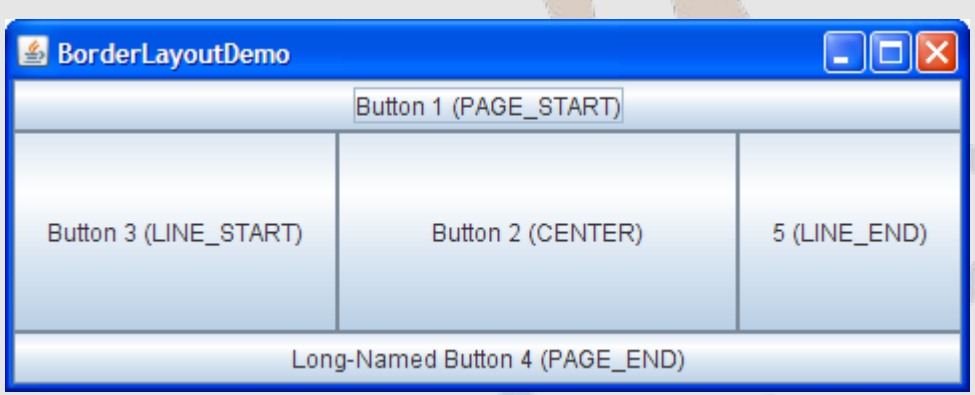
```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.URL;

import javax.swing.*;

public class Browser extends JFrame implements ActionListener {
    private static final String COMMAND_GO = "go";

    private JEditorPane webpage;
    private JTextField url;
    private JTextArea htmlPage;

    private JPanel createMainPanel() {
        JPanel mp = new JPanel();
        // panel z kolumnowym (Y_AXIS) ukladem elementow
        mp.setLayout(new BoxLayout(mp, BoxLayout.Y_AXIS));
```

LAYOUT MANAGER



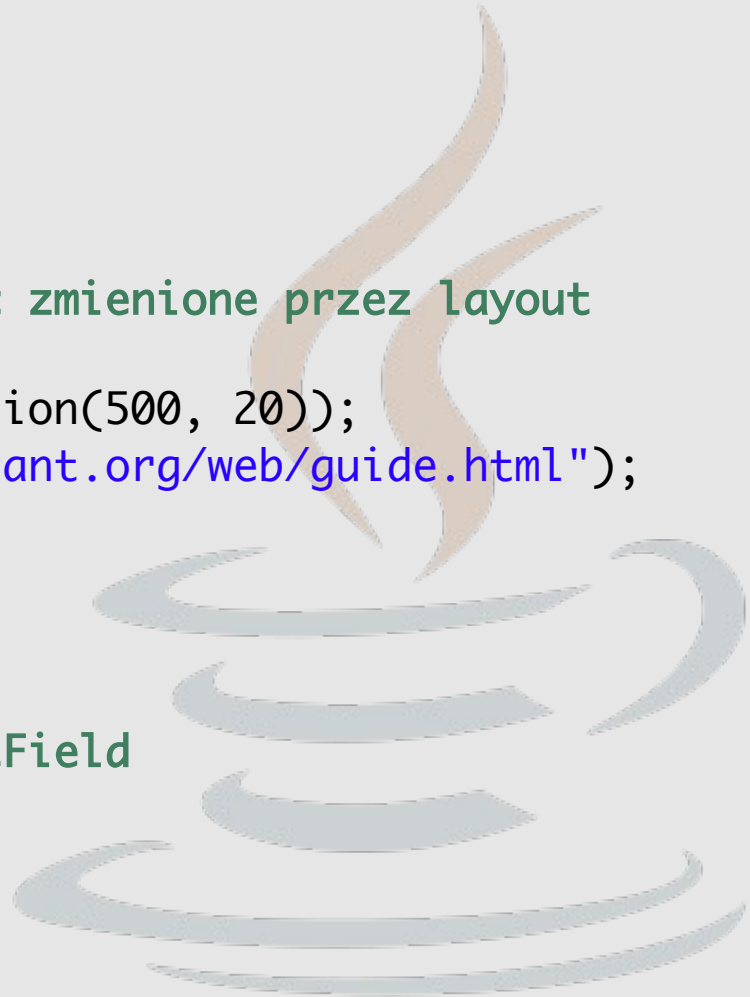






KOMPONENTY

```
// panel z domyślnym FlowLayout
JPanel p = new JPanel();
this.url = new JTextField();

// sugerowane rozmiary – mogą zostać zmienione przez layout
// managera
this.url.setPreferredSize(new Dimension(500, 20));
this.url.setText("http://www.simongrant.org/web/guide.html");
JLabel l = new JLabel("adres");
// label opisujacy url
l.setLabelFor(this.url);

// dodajemy do panelu JLabel I JTextField
p.add(l);
p.add(this.url);
```



KOMPONENTY

```
// tworzymy przycisk
JButton b = new JButton("Go");
b.setActionCommand(COMMAND_GO);
// ktorego “akcje” beda obslugiwane przez biezacy obiekt
// (implementujacy interfejs ActionListener. Akcja “produkowana”
// przez przycisk będzie identyfikowane Stringiem COMMAND_GO
b.addActionListener(this);
b.setPreferredSize(new Dimension(100, 40));
// dodanie przycisku do panelu
p.add(b);
// dodanie panelu p (z komponentami rozmieszczonymi przez
// FlowLayout) do panelu, w ktorym obowiazuje BoxLayout
mp.add(p);
```

KOMPONENTY

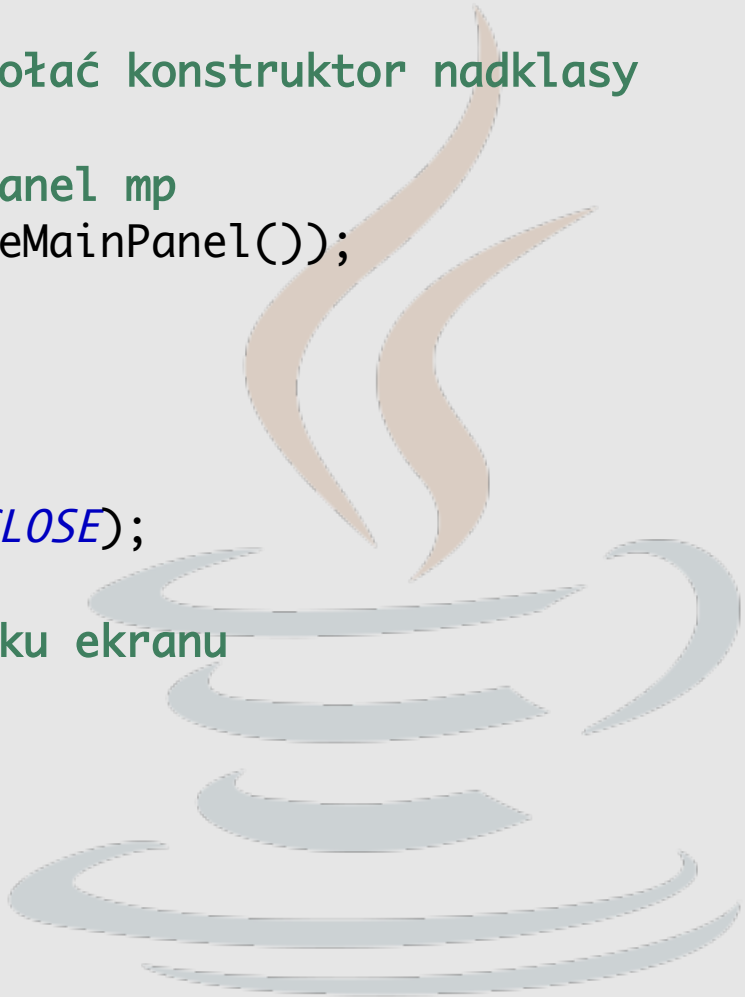
```
this.webpage = new JEditorPane();
this.htmlPage = new JTextArea();
try {
    // wczytujemy zawartosc strony “startowej”
    this.setPage(
            new URL("http://www.simongrant.org/web/guide.html"));
} catch (IOException e) { }
// Tworzymy panel z zakladkami
JTabbedPane tp = new JTabbedPane();
tp.setPreferredSize(new Dimension(600, 400));
// pole tekstowe webpage umieszczamy wewnatrz panela
// scrollowanego. Dzieki temu zawartosc okienka bedzie mogla
// zajmowac wiecej miejsca niz widok
JScrollPane sp = new JScrollPane(this.webpage);
// zakladka “page” bedzie zawierac webpage (wewnatrz JScrollPane)
tp.add("page", sp);
```

KOMPONENTY

```
        // zakladka “html” bedzie zawierac htmlPage (wewnatrz JscrollPane)
        sp = new JScrollPane(this.htmlPage);
        tp.add("html", sp);

        // przygotowany JTabbedPane zostaje dodany do panelu mp
        mp.add(tp);
        return mp;
    }

    private void setPage(URL page) throws IOException {
        String s;
        this.webpage.setPage(page);
        BufferedReader br = new BufferedReader(new InputStreamReader(
                                                page.openStream()));

        while ((s = br.readLine()) != null)
            this.htmlPage.append(s + "\n");
    }
```

KOMPONENTY

```
public Browser() {
    // zawsze na początku powinniśmy wywołać konstruktor nadklasy
    super();
    // zawartoscia okna Browser bedzie panel mp
    this.getContentPane().add(this.createMainPanel());
}

public static void createAndShow() {
    Browser b = new Browser();
    b.setDefaultCloseOperation(EXIT_ON_CLOSE);
    b.pack();
    // okno zostanie umieszczone na srodku ekranu
    b.setLocationRelativeTo(null);
    b.setVisible(true);
}
```

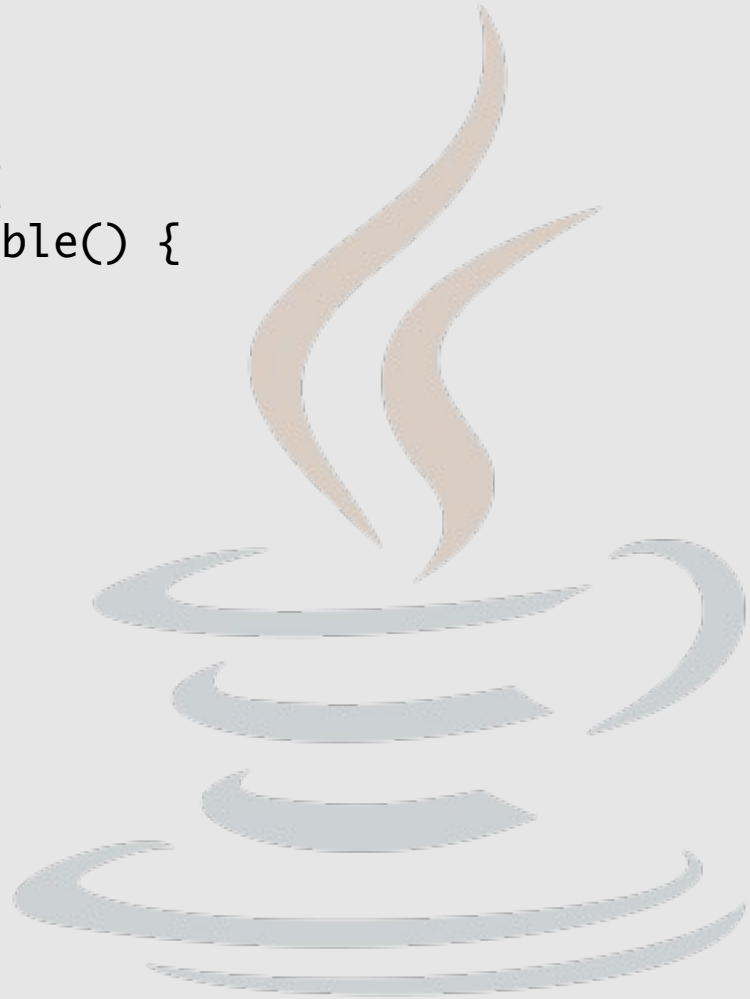


KOMPONENTY

```
// Interfejs ActionListener implementuje jedna metode, ktora jest
// uruchamiana gdy nastapi zdarzenie na komponencie nasluchiwanym
// przez ten obiekt. Informacje o zrodle akcji sa przekazywane przez
// argument ActionEvent
@Override
public void actionPerformed(ActionEvent e) {
    if (COMMAND_GO.equals(e.getActionCommand())) {
        try {
            // przeladowujemy strone
            this.setPage(new URL(this.url.getText()));
        } catch (IOException e2) {
            this.webpage.setText(
                            "Problem z adresem " + this.url.getText());
            this.htmlPage.setText(
                            "Problem z adresem " + this.url.getText());
        }
    }
}
```

KOMPONENTY

```
    // uruchomienie programu
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShow();
        }
    });
}
```



DZIĘKUJĘ ZA UWAGĘ

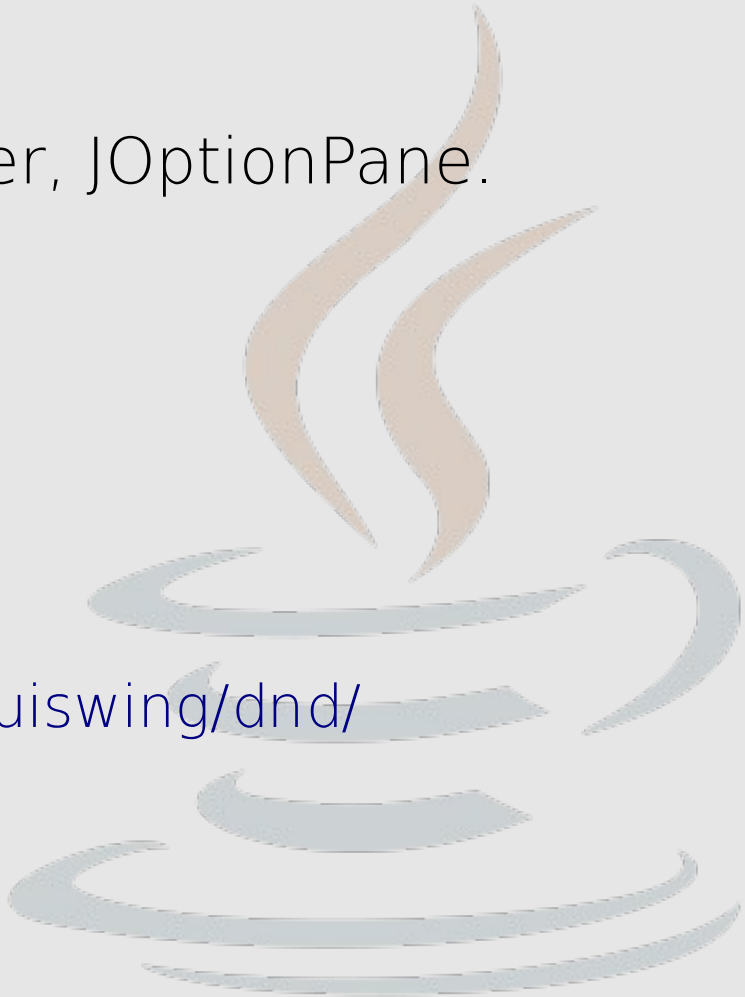
SWING c.d.

ZAGADNIENIA:

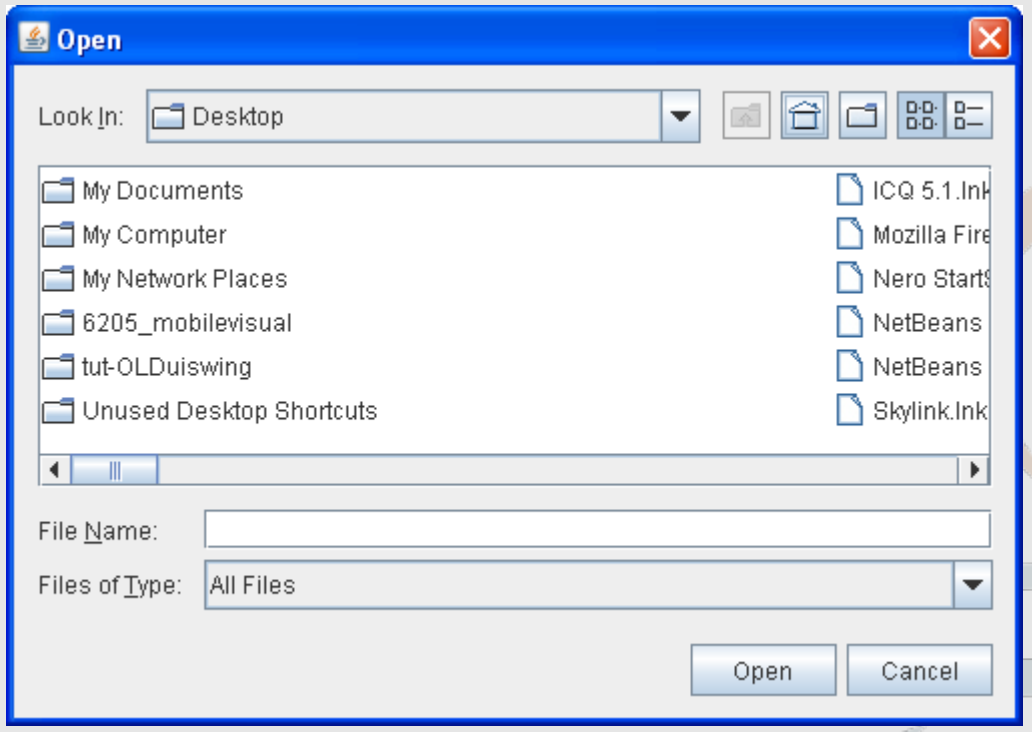
- przydatne narzędzia: JFileChooser, JOptionPane.
- drag'n drop,
- menu kontekstowe.

MATERIAŁY:

http://docs.oracle.com/javase/tutorial/uiswing/dnd/



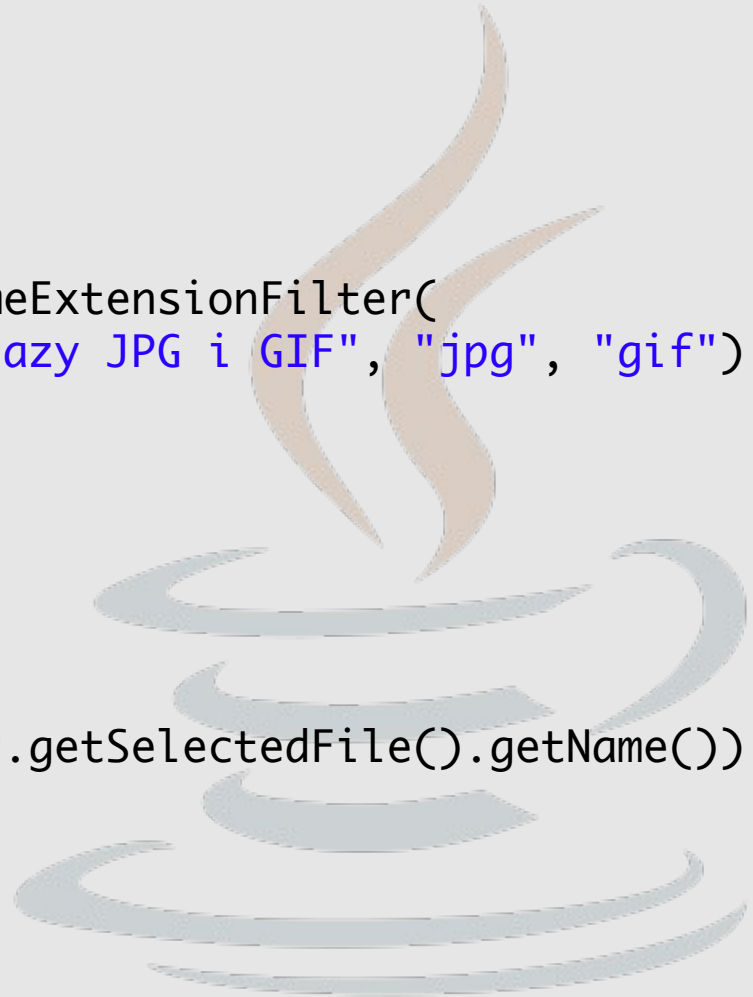
JFileChooser



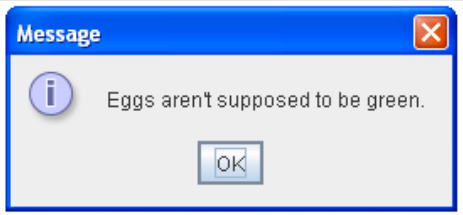
http://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html

JFileChooser

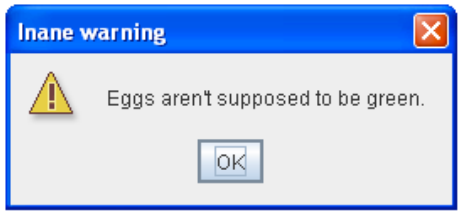
```
JFileChooser chooser = new JFileChooser();
// filtr plików
FileNameExtensionFilter filter = new FileNameExtensionFilter(
                                        "Obrazy JPG i GIF", "jpg", "gif");
chooser.setFileFilter(filter);
// wyświetlenie okienka
int ret = chooser.showOpenDialog(parent);
// Jesli wcisnieto ok lub open
if(ret == JFileChooser.APPROVE_OPTION) {
   System.out.println("Wybrales plik: " +
                                    chooser.getSelectedFile().getName());
}
```



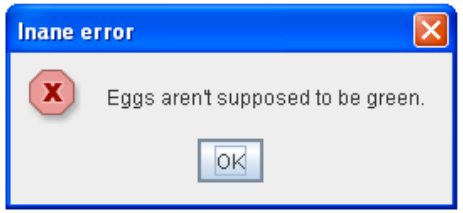
JOptionPane



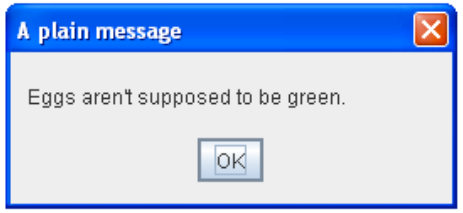

```
//default title and icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.");
```



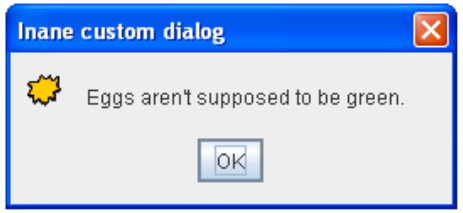

```
//custom title, warning icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.",
    "Inane warning",
    JOptionPane.WARNING_MESSAGE);
```




```
//custom title, error icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.",
    "Inane error",
    JOptionPane.ERROR_MESSAGE);
```



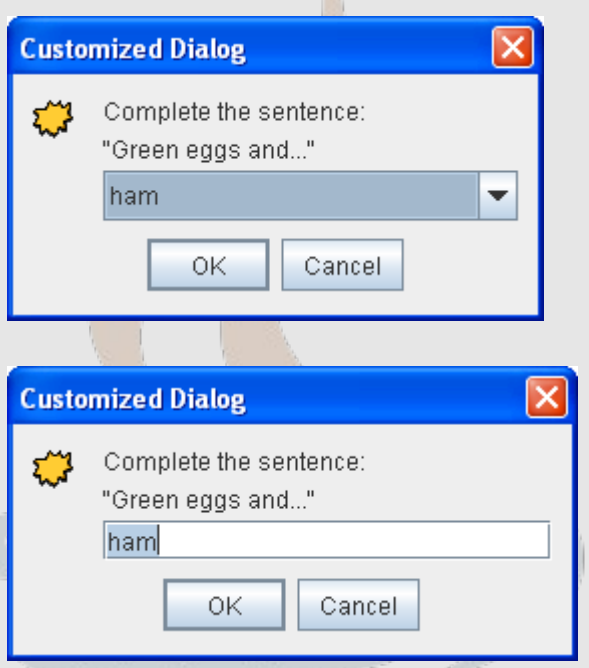

```
//custom title, no icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.",
    "A plain message",
    JOptionPane.PLAIN_MESSAGE);
```




```
//custom title, custom icon
JOptionPane.showMessageDialog(frame,
    "Eggs are not supposed to be green.",
    "Inane custom dialog",
    JOptionPane.INFORMATION_MESSAGE,
    icon);
```

JOptionPane

```
Object[] possibilities = {"ham", "spam", "yam"};
String s = (String)JOptionPane.showInputDialog(
                    frame,
                    "Complete the sentence:\n"
                    + "\"Green eggs and...\"",
                    "Customized Dialog",
                    JOptionPane.PLAIN_MESSAGE,
                    icon,
                    possibilities, // null
                    "ham");
```



http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html#features

Drag and Drop

Wiele komponentów standardowo obsługuje drag'n drop (a dokładniej drop). Jeśli źródłem danych ma być komponent Swing, to należy na nim wykonać metodę: `setDragEnabled(true)`.

Drag'n Drop funkcjonalnie jest równoważne kopiowaniu przez schowek (`ctrl-c`, `ctrl-v`). Aby całkowicie kontrolować ten proces należy poznać kilka klas odpowiedzialnych za jego realizację.

http://docs.oracle.com/javase/tutorial/uiswing/dnd/index.html

Transferable

Przenoszone dane są reprezentowane przez interfejs **Transferable**.

- `Object getTransferData(DataFlavor flavor)` - zwraca transferowany obiekt
- `DataFlavor[] getTransferDataFlavors()` - zwraca wszystkie dostępne postacie przenoszonego obiektu,
- `boolean isDataFlavorSupported(DataFlavor flavor)` – zwraca informację, czy obiekt jest dostępny w odpowiedniej postaci

DataFlavor

Przenoszone obiekt jest zwykle dostępny w wielu postaciach. Przykładowo, zdjęcie może być reprezentowane przez zbiór pixeli (grafika), nazwę pliku, w którym jest zapisane, zbiór bajtów, czy też zakodowaną w formacie **Base64** zawartość.

W związku z tym, w zależności od tego, gdzie takie zdjęcie przenosimy (wklejamy) możemy zobaczyć je pod inną postacią. Te postacie są reprezentowane przez klasę `DataFlavor`.

TransferHandler

Obiekt, który zarządza procesem drag'n drop jest TransferHandler.

Posiada on metody służące do eksportu danych:

```
int getSourceActions(JComponent c) {
    return COPY_OR_MOVE;
}

Transferable createTransferable(JComponent c) {
    return new StringSelection(c.getSelection());
}

void exportDone(JComponent c, Transferable t, int action) {
    if (action == MOVE) {
        c.removeSelection();
    }
}
```

TransferHandler

oraz importu:

`canImport(TransferHandler.TransferSupport)` — zwraca true jeśli komponent znajdujący się pod kursorem myszki akceptuje przenoszony obiekt

`importData(TransferHandler.TransferSupport)` — metoda jest wywoływana po upuszczeniu (drop) obiektu. zwraca true jeśli import obiektu zakończył się powodzeniem.

PRZYKŁAD

```
import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

public class JDnDFrame extends JFrame {

    public JDnDFrame() {
        FileTransferHandler fth = new FileTransferHandler();
        JTextArea ta = new JTextArea();
        ta.setTransferHandler(fth);
        ta.setPreferredSize(new Dimension(600, 400));
        File[] fa = (new File(".")).listFiles();
        JList<File> fl = new JList<File>(fa);
        fl.setTransferHandler(fth);
        fl.setDragEnabled(true);
```

PRZYKŁAD

```
        JSplitPane sp = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                                        ta, fl);
        this.getContentPane().add(sp);
    }

    public static void createAndShow() {
        JDnDFrame f = new JDnDFrame();
        f.setDefaultCloseOperation(EXIT_ON_CLOSE);
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {  createAndShow();  }
        });
    }
```

PRZYKŁAD

```
private static class FileTranferable implements Transferable {

    public static DataFlavor fileFlavor =
                                        new DataFlavor(File.class, "file");

    // tutaj przechowujemy transferowany plik
    private File file;

    public FileTranferable(File f) {
        this.file = f;
    }

    @Override
    public DataFlavor[] getTransferDataFlavors() {

        // tylko jedna postać
        DataFlavor[] df = new DataFlavor[1];
        df[0] = fileFlavor;
        return df;
    }
```

PRZYKŁAD

```
    @Override
    public boolean isDataFlavorSupported(DataFlavor flavor) {
        return (flavor == fileFlavor);
    }

    @Override
    public Object getTransferData(DataFlavor flavor)
                        throws UnsupportedFlavorException, IOException {
        if (isDataFlavorSupported(flavor))
            return this.file;

        // nie obsługujemy innych postaci obiektu
        throw new UnsupportedFlavorException(flavor);
    }
}
```

PRZYKŁAD

```
private static class FileTransferHandler extends TransferHandler
                                          implements ActionListener{
    private JPopupMenu popup;

    // bez tej metody nie rozpocznie się eksport obiektu
    public int getSourceActions(JComponent c) {
        return COPY_OR_MOVE;
    }
    // tworzy obiekt Transferable zawierający przenoszony element
    protected Transferable createTransferable(JComponent c) {
        Object obj = ((JList<File>) c).getSelectedValue();
        return new FileTranferable((File) obj);
    }
    // handler akceptuje wszystko. W praktyce to powinno być
    // zaimplementowane porządnie, ale tutaj takie zachowanie nie
    // będzie przeszkadzać
    public boolean canImport(TransferSupport ts) {
        return true;
    }
```

PRZYKŁAD

```
private JTextArea destination;
private File file;

public boolean importData(TransferSupport ts) {
    // tutaj niepotrzebne, ale ogólnie powinno być – żeby nie
    // importować nieobsługiwanych obiektów
    if (!canImport(ts))
        return false;
    try {
        this.file = (File)
                    ts.getTransferable().getTransferData(
                                        FileTranferable.fileFlavor);
        this.destination = (JTextArea) ts.getComponent();
        Point p = ts.getDropLocation().getDropPoint();
        this.popup.show(ts.getComponent(), p.x, p.y);
        return true;
    } catch (UnsupportedFlavorException | IOException e) { }
    return false;
}
```

PRZYKŁAD

```
// Konstruktor, w którym tworzymy jeszcze menu kontekstowe,
// które będzie nam określać sposób transferu danych
public FileTransferHandler(){
    super();
    popup = new JPopupMenu();
    JMenuItem mi = new JMenuItem("nazwa");
    // TranferHandler jest również ActionListenerem.
    // Dzięki temu nie trzeba dodatkowo przekazywać
    // listenerowi informacji o transferowanym obiekcie
    mi.addActionListener(this);
    mi.setActionCommand("name");
    popup.add(mi);
    mi = new JMenuItem("zawartosc");
    mi.addActionListener(this);
    mi.setActionCommand("content");
    popup.add(mi);
}
```

PRZYKŁAD

```
@Override
public void actionPerformed(ActionEvent e) {
    // przenosimy plik pod jako nazwe
    if (e.getActionCommand().equals("name")){
        // dodajemy ją na koncu teksu, ale możnaby też sprawdzić
        // pozycje kursora i umieścić tewn tekst w okteślonej
        // pozycji
        this.destination.append(this.file.getName() + "\r\n");
    }else if (e.getActionCommand().equals("content")){
        // zawartosc katalogu to nazwy plików, które zawiera
        if (this.file.isDirectory()) {
            for(String s : this.file.list())
                this.destination.append(s + "\r\n");
        }
```

PRZYKŁAD

```
                // w przypadku plików obsługujemy tylko zawartosc plikow
                // tekstowych
                else if(this.file.getName().endsWith(".txt")){
                    BufferedReader br;
                    try {
                        br = new BufferedReader(
                                            new FileReader(this.file));
                        String s;
                        while((s=br.readLine())!=null){
                            this.destination.append(s + "\r\n");
                        }
                        br.close();
                    } catch (IOException e1) { }
                }
            }
        }
    }
}
```

DZIĘKUJĘ ZA UWAGĘ

JAVA I SIECI

ZAGADNIENIA:

- URL,
- Interfejs gniazd,
- transmisja SSL,
- protokół JNLP.

MATERIAŁY:

http://docs.oracle.com/javase/tutorial/networking/index.html

URL

URL – Unified Resource Locator jest podstawową klasą identyfikującą zasoby w internecie:

```
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();
    }
}
```

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();
    }
}
```

URLConnection

URLConnection umożliwia także zapis do wskazanego zasobu przez obiekt URL.

```
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());

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

URLConnection

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

- http,
- https,
- ftp, ftp://login:haslo@serwer:port/katalog/podkatalog/plik
- file,
- jar.

Obsługa innych protokołów wymaga implementacji klasy URLStreamHandler.

INTERFEJS GNIAZD

Standardowa obsługa sieci w Javie opiera się o tzw. interfejs gniazd. 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.

PROGRAM KLIENCKI

```
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));
```

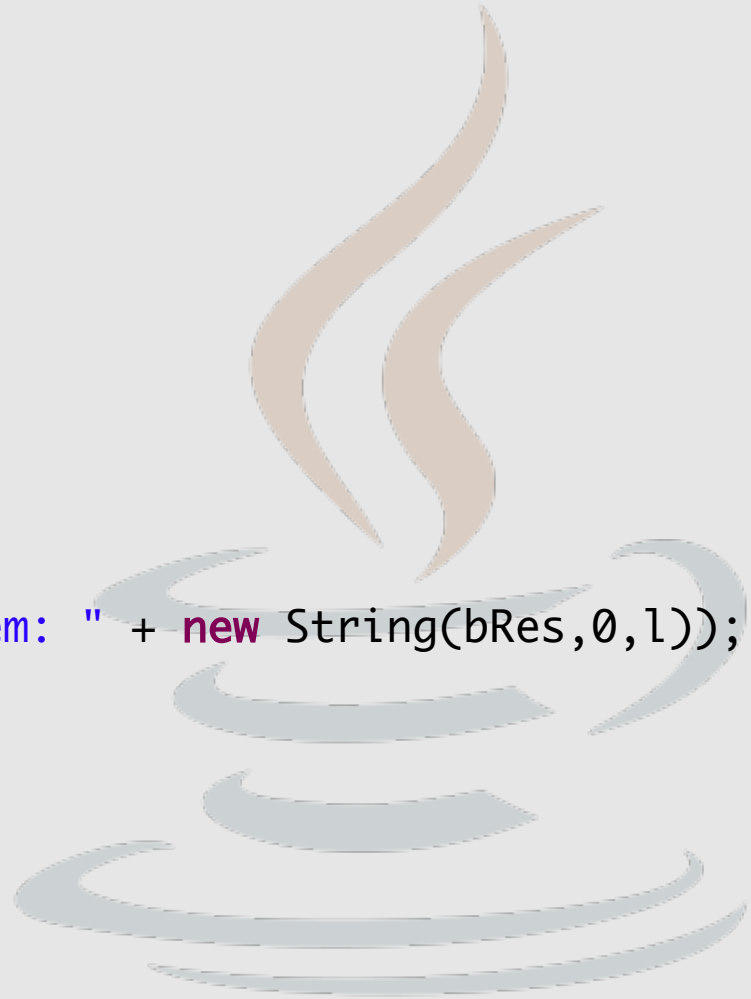
PROGRAM KLIENCKI

```
        // 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
```

PROGRAM KLIENCKI

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();
```

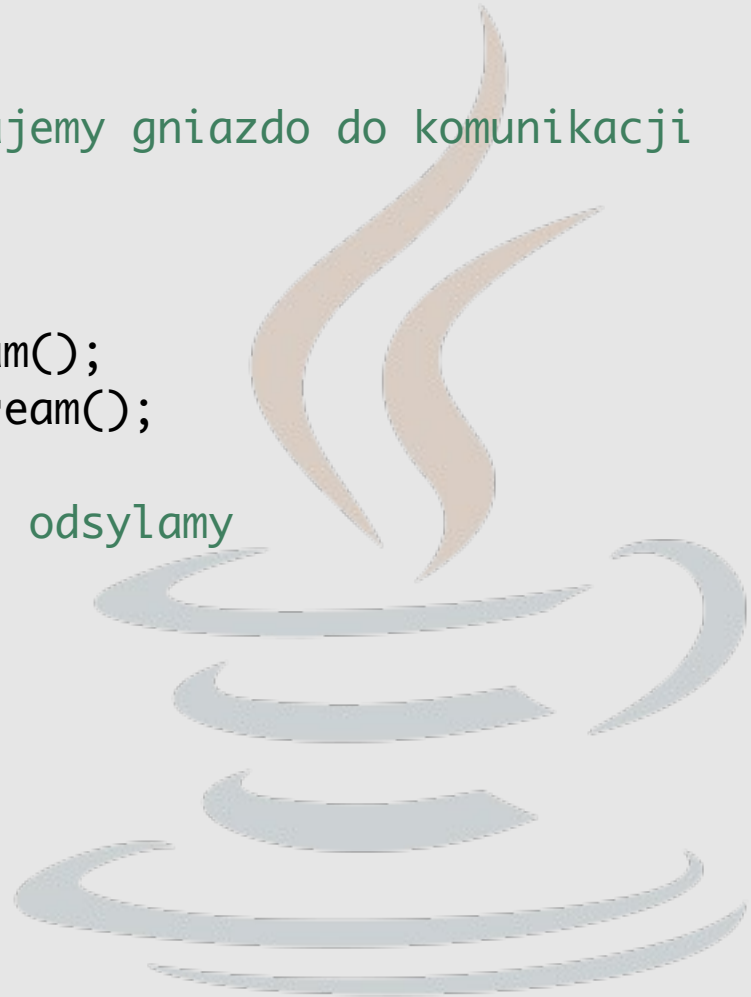


SERWER ECHO

```
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]));
```

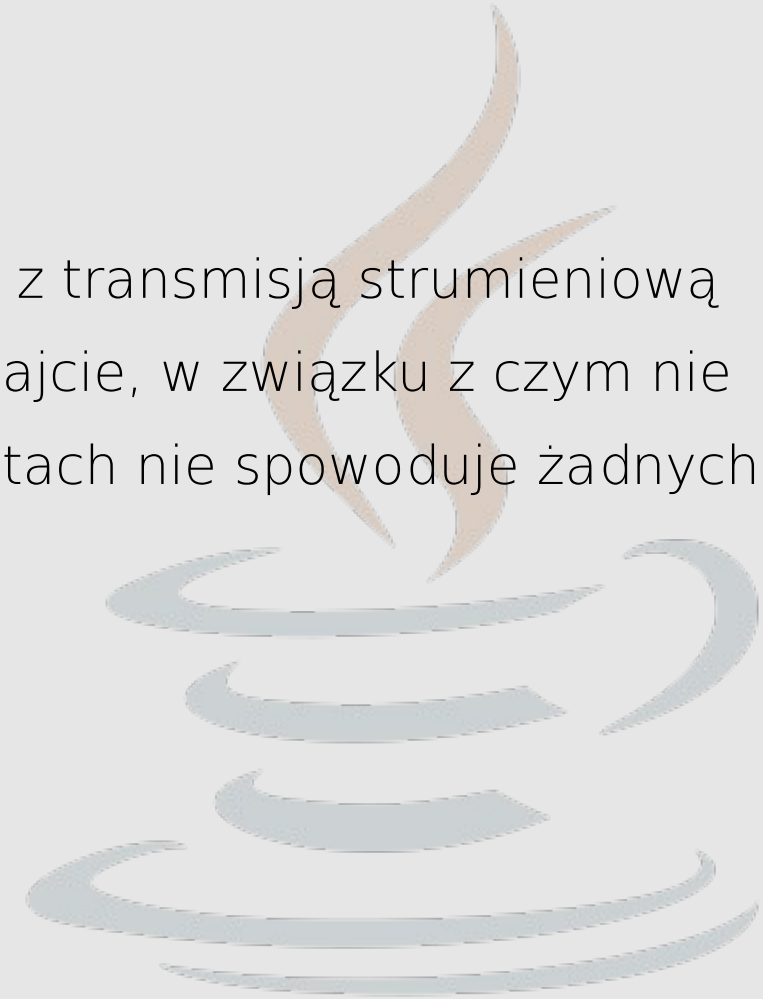
SERWER ECHO

```
        // 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();
        }
    }
}
```



SERWER ECHO

W serwerze nie ma problemu związanego z transmisją strumieniową ponieważ dane są przetwarzane bajt po bajcie, w związku z czym nie sytuacja, gdy dane dotrą w różnych pakietach nie spowoduje żadnych efektów ubocznych.



SSL W JAVIE

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 (http://www.jcraft.com/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();
        }
    }
}
```

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();
        }
    }
}
```

SSL W JAVIE

Pierwsza czynność to wygenerowanie klucza:

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

Uruchomienie serwera:

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

Uruchomienie klienta:

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

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
```

Przykład ze strony: http://stilius.net/java/java_ssl.php.

SSL W JAVIE

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

JAVA WEB START

Technologia Java Web Start jest stosowana do lokalnego uruchamiania programów w Javie umieszczonych w sieci.

JWS:

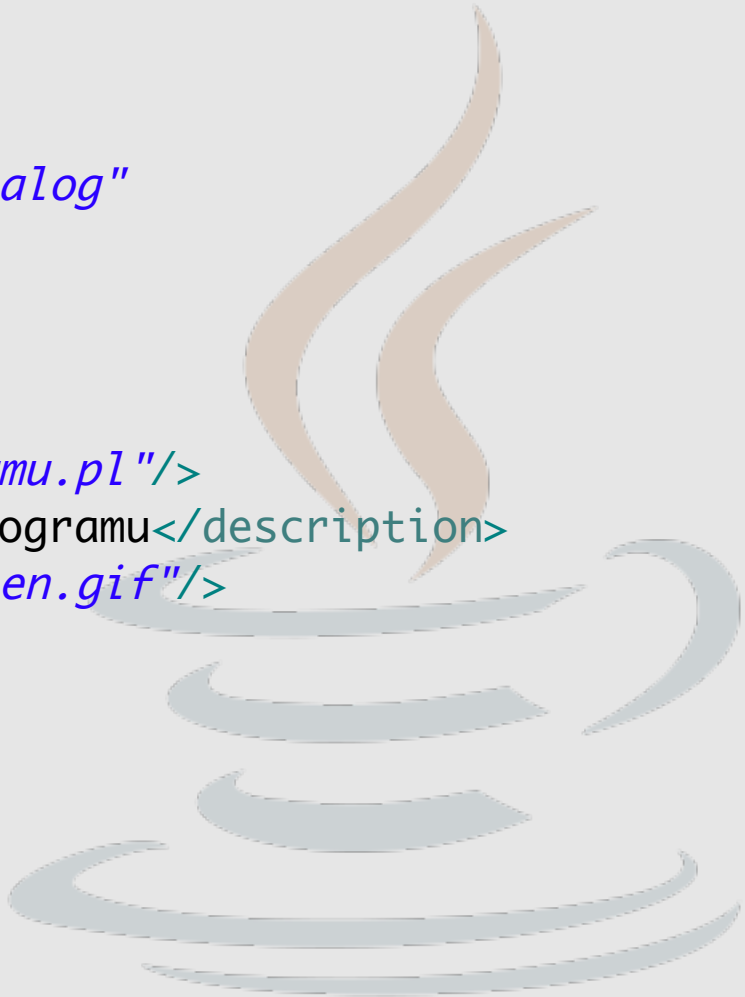
- jest w pełni niezależna od używanych przeglądarek internetowych,
- umożliwia automatyczne pobranie właściwej wersji środowiska JRE,
- pobierane są tylko pliki, które zostały zmienione,
- obsługuje prawa dostępu do zasobów lokalnego komputera (dysk, sieć, itp.),
- do opisu zadania do uruchomienia wykorzystuje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

Więcej informacji:

http://docs.oracle.com/javase/tutorial/deployment/webstart/index.html

PROTOKÓŁ JNLP

```
<?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>
```



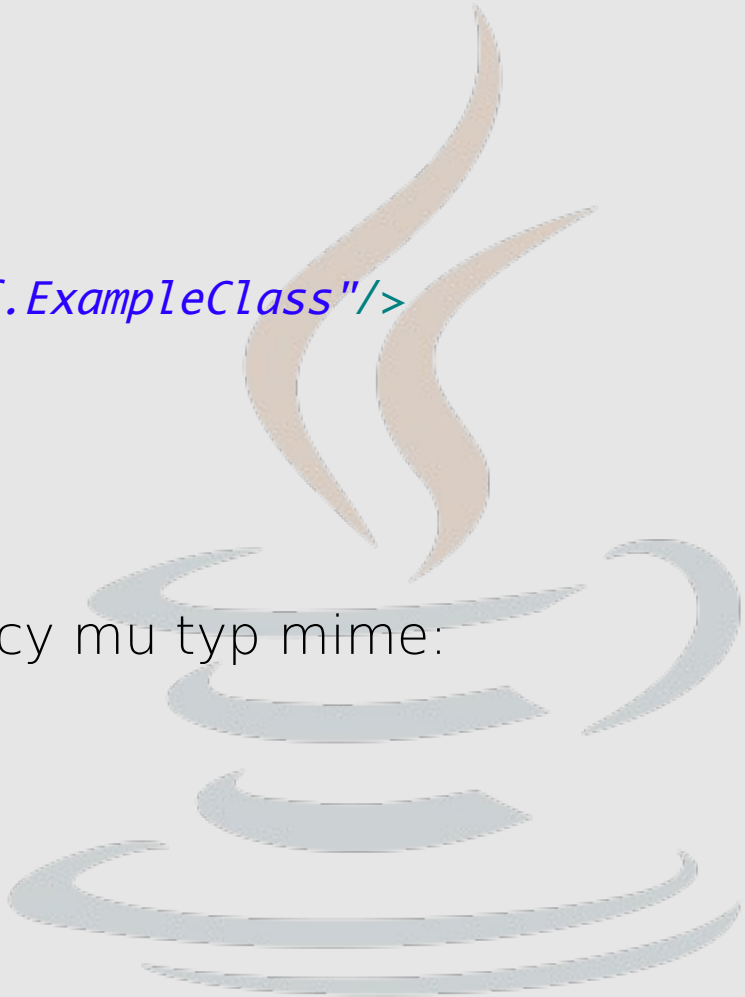
PROTOKÓŁ JNLP

```
  <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>
```

Plik jnlp umieszczamy na serwerze www.

Często należy skonfigurować odpowiadający mu typ mime:

application/x-java-jnlp-file JNLP

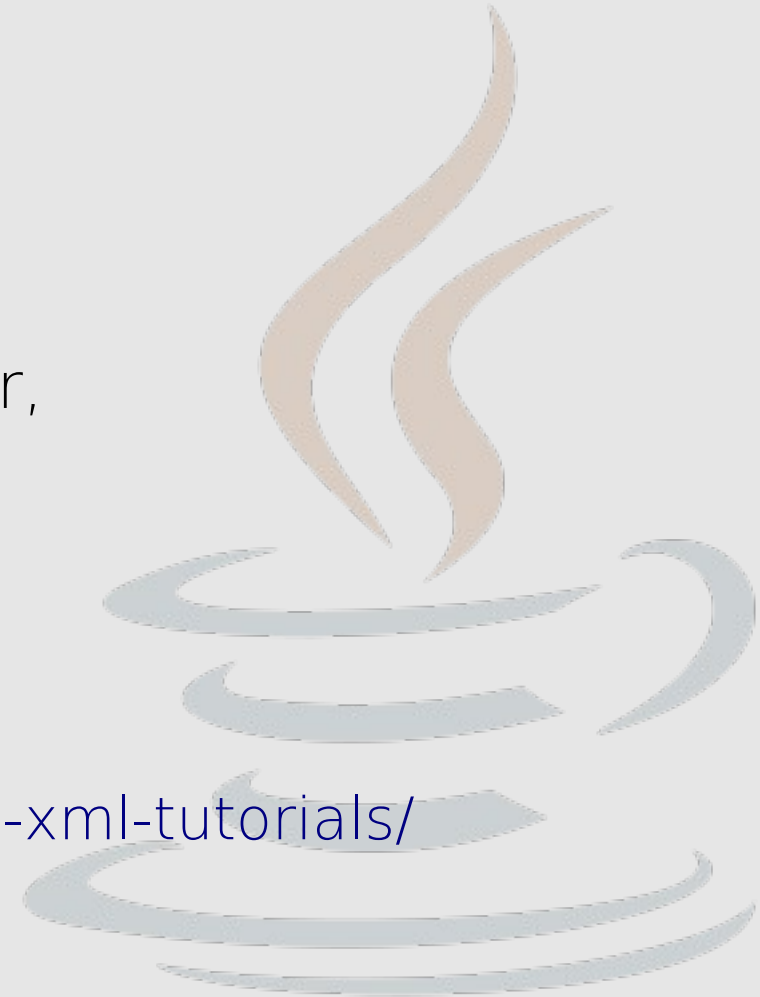


DZIĘKUJĘ ZA UWAGĘ

JAVA I XML

ZAGADNIENIA:

- DOM,
- SAX,
- JAXB, XMLDecoder i XMLEncoder,
- ANT.



MATERIAŁY:

http://www.mkyong.com/tutorials/java-xml-tutorials/
http://ant.apache.org

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:dc="http://purl.org/dc/elements/1.1/" version="2.0">
  <channel>
    <title>Aktualności</title>
    <link>http://www.uj.edu.pl/uniwersytet/aktualnosci/...</link>
    <description>Aktualności Uniwersytetu Jagiellońskiego</description>
    <item>
      <title>Lista Pamięci - Stanisław Szczur</title>
      <link>http://www.uj.edu.pl/uniwersytet/aktualnosci/...</link>
      <description />
      <pubDate>Thu, 20 Dec 2012 07:44:00 GMT</pubDate>
      <dc:creator>Jolanta Herian-Ślusarska</dc:creator>
      <dc:date>2012-12-20T07:44:00Z</dc:date>
    </item>
    <item>...</item>
    ...
  </channel>
</rss>
```

DOM – ODCZYT

Parsery DOM (*Document Object Model*) tworzą drzewo reprezentujące dane zawarte w dokumencie XML. Po zbudowaniu DOM przetwarzanie odbywa się na modelu w pamięci operacyjnej.

```
import java.io.IOException;
import java.net.URL;

import javax.xml.parsers.*;

import org.w3c.dom.*;
import org.xml.sax.SAXException;

public class DOMExample {

    public static void main(String[] args) throws
                    ParserConfigurationException,SAXException,IOException {
```

DOM – ODCZYT

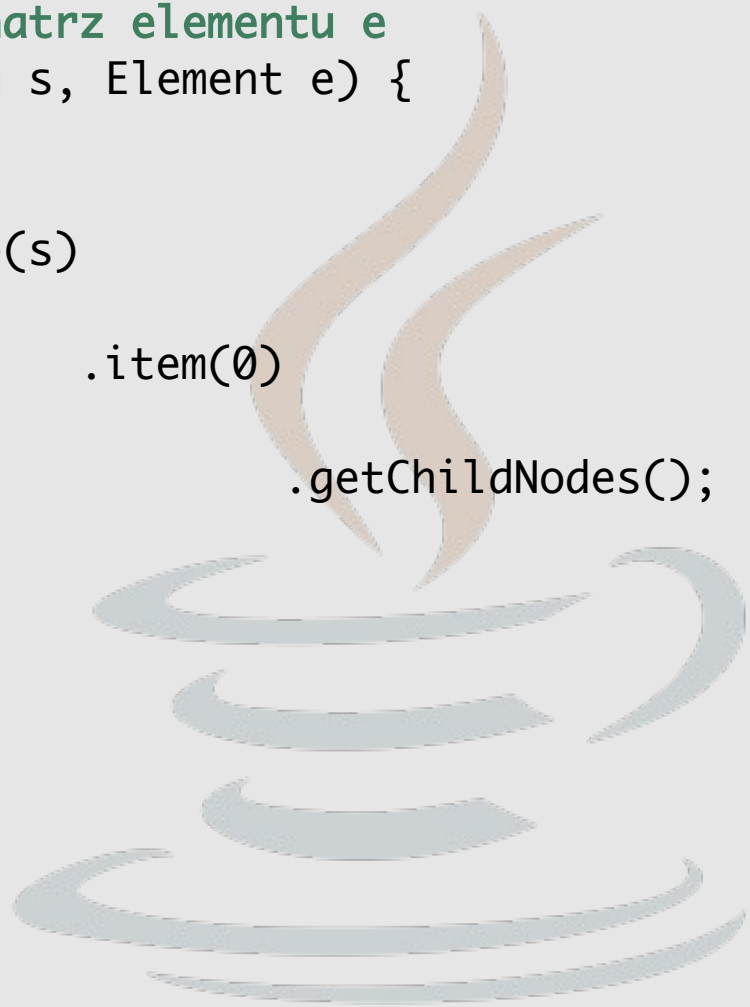
```
        URL url = new URL(
"http://www.uj.edu.pl/uniwersytet/aktualnosci/-/journal/rss/10172/36018?
doAsGroupId=10172&refererPlid=10181"); // jakis RSS

        // domyslna fabryka “obiektow tworzących dokumenty”
        DocumentBuilderFactory dbFactory =
                                        DocumentBuilderFactory.newInstance();

        // obiekt “tworzący dokumenty”
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();

        // tworzenie modelu DOM na podstawie zrodla XML (parsowanie XML'a)
        Document doc = dBuilder.parse(url.openStream());

        // obiekt doc umozliwia dostep do wszystkich danych zawartych
        // w dokumencie XML
        System.out.println("Root element :" +
                                    doc.getDocumentElement().getNodeName());
```

DOM – ODCZYT

```
        NodeList nList = doc.getElementsByTagName("item");
        for (int i = 0; i < nList.getLength(); i++) {

            // lista “dzieci” i-tego itema
            Node n = nList.item(i);

            // czy “dziecko” jest elementem?
            if (n.getNodeType() == Node.ELEMENT_NODE) {
                Element e = (Element) n;
                System.out.println("Tytul        : "
                                   + getTagValue("title", e));
                System.out.println("Link         : "
                                   + getTagValue("link", e));
                System.out.println("dodane przez : "
                                   + getTagValue("dc:creator", e));
            }
        }
    }
```

DOM – ODCZYT

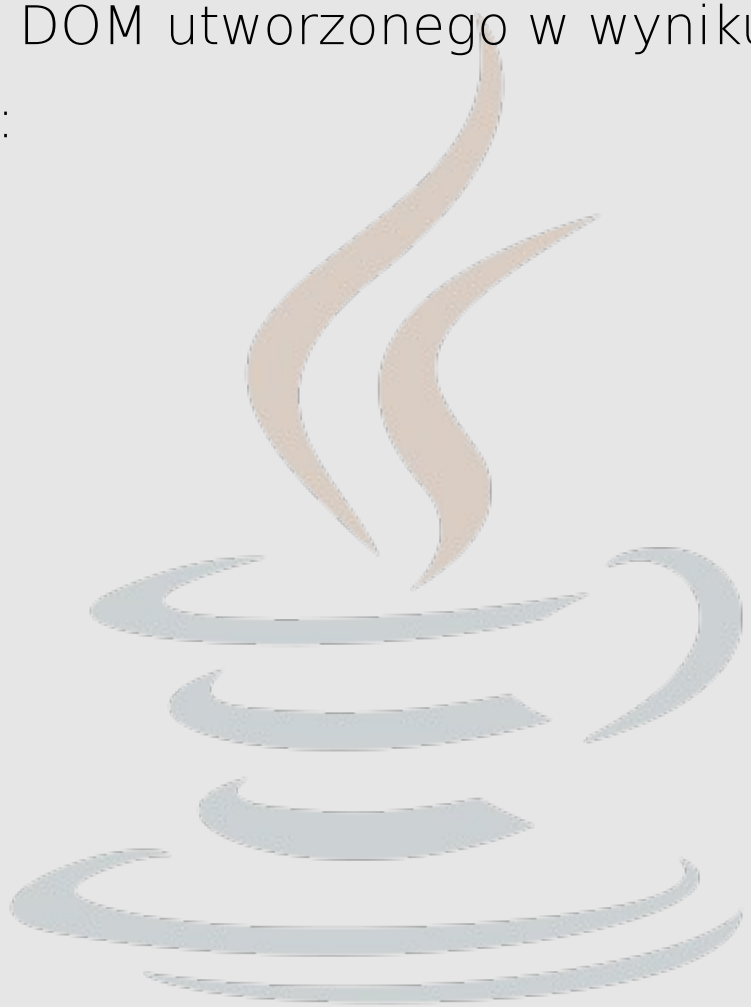
```
    // zwraca wartosc zapisana w tagu s wewnatrz elementu e
    private static String getTagValue(String s, Element e) {

        // lista “dzieci” e o nazwie s
        NodeList nl = e.getElementsByTagName(s)
        // pierwszy wpis z tej listy
                                                .item(0)
        // to co on zawiera – jego “dzieci”
                                                        .getChildNodes();


        // pierwsze z tych dzieci
        Node n = (Node) nl.item(0);
        // zawartosc, ktora tam jest
        return n.getNodeValue();
    }

}
```



DOM – MODYFIKACJA

Możliwa jest także modyfikacja zawartości DOM utworzonego w wyniku parsowania dokumentu XML. Przykładowo:

```
// pobieramy element title i zmieniamy go
if ("title".equals(node.getNodeName())) {
    node.setTextContent("Nowy tytul");
}
// kasujemy link
if ("link".equals(node.getNodeName())) {
    itemElement.removeChild(node);
}

// tworzenie nowego dokumentu
Document doc = docBuilder.newDocument();
Element e = doc.createElement("root");
doc.appendChild(e);
```



DOM – ZAPIS

Zapis dokumentu:

```
// domyslna fabryka transformatorow
TransformerFactory transformerFactory = TransformerFactory.newInstance();

// nowy tranformator
Transformer transformer = transformerFactory.newTransformer();

// wejscie transformatora (skad transformator bierze dane)
DOMSource source = new DOMSource(doc);
// wyjscie transformatora (gdzie “zmienione” dane zostana zapisane)
StreamResult result = new StreamResult(new File("file.xml"));

// uruchomienie transformatora – zapis DOM do pliku w formacie XML
transformer.transform(source, result);
```

SAX – ODCZYT

SAX (*Simple API for XML*) w miarę czytania dokumentu wywołuje zdarzenia związane z parsowaniem. Parsery SAX są szybsze i nie wymagają tak dużej ilości pamięci jak DOM.

```
import java.io.IOException;
import java.net.URL;
import javax.xml.parsers.*;
import org.xml.sax.*;
public class SAXExample {
    public static void main(String[] args) throws SAXException,
                                    IOException, ParserConfigurationException{
        URL url = new URL("http://www.uj.edu.pl/...");
        SAXParserFactory f = SAXParserFactory.newInstance();
        SAXParser saxParser = f.newSAXParser();
         DefaultHandler handler = new ExampleSAXHandler();
        saxParser.parse(url.openStream(), handler);
    }
}
```

DefaultHandler

- `void characters(char[] ch, int start, int length)` – wywoływane przy odczycie znaku wewnątrz elementu,
- `void endDocument()` – wywoływane gdy koniec dokumentu
- `void endElement(String uri, String localName, String qName)` – koniec elementu,
- `void error(SAXParseException e)` – błąd (możliwy do naprawienia),
- `void fatalError(SAXParseException e)` – błąd,
- `void ignorableWhitespace(char[] ch, int start, int length)` – ignorowany pusty znak,
- `void startDocument()` – początek dokumentu,
- `void startElement(String uri, String localName, String qName, Attributes attributes)` – początek elementu
- `void warning(SAXParseException e)` – ostrzeżenie parsera.

...

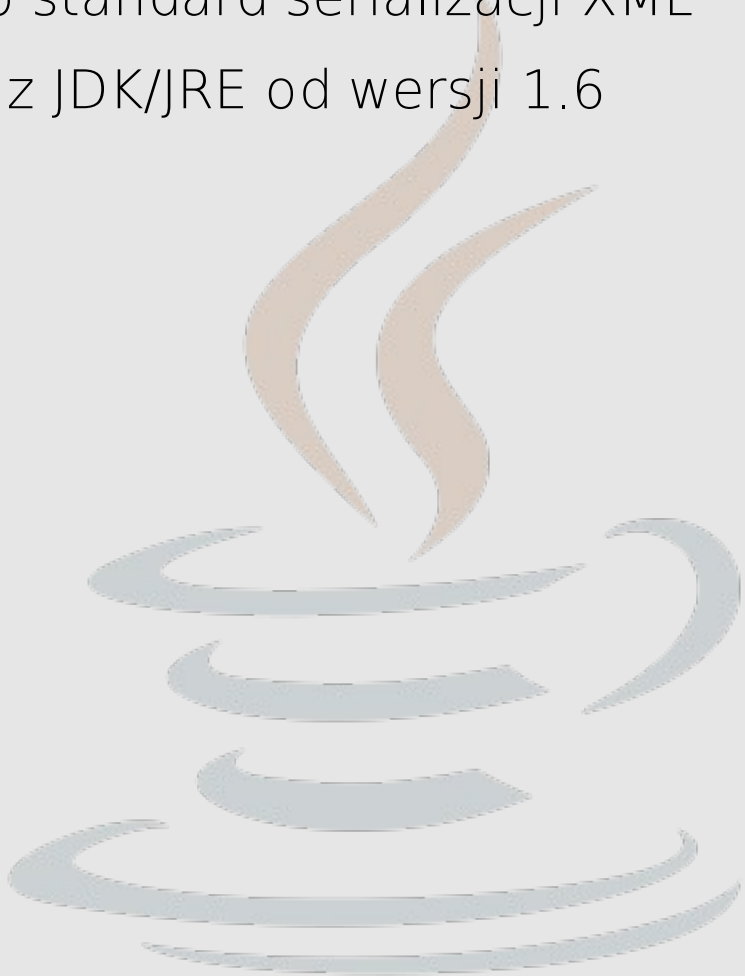
DefaultHandler

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

class ExampleSAXHandler extends DefaultHandler {

    public void startElement(String uri, String localName, String qName,
                             Attributes attributes) throws SAXException {
        System.out.println("Element :" + qName);
    }
    public void endElement(String uri, String localName, String qName)
                                                    throws SAXException {
        System.out.println("Konie elementu :" + qName);
    }
    public void characters(char ch[], int start, int length)
                                                    throws SAXException {
        System.out.println("zawartosc : "+new String(ch, start, length));
    }
}
```

JAXB

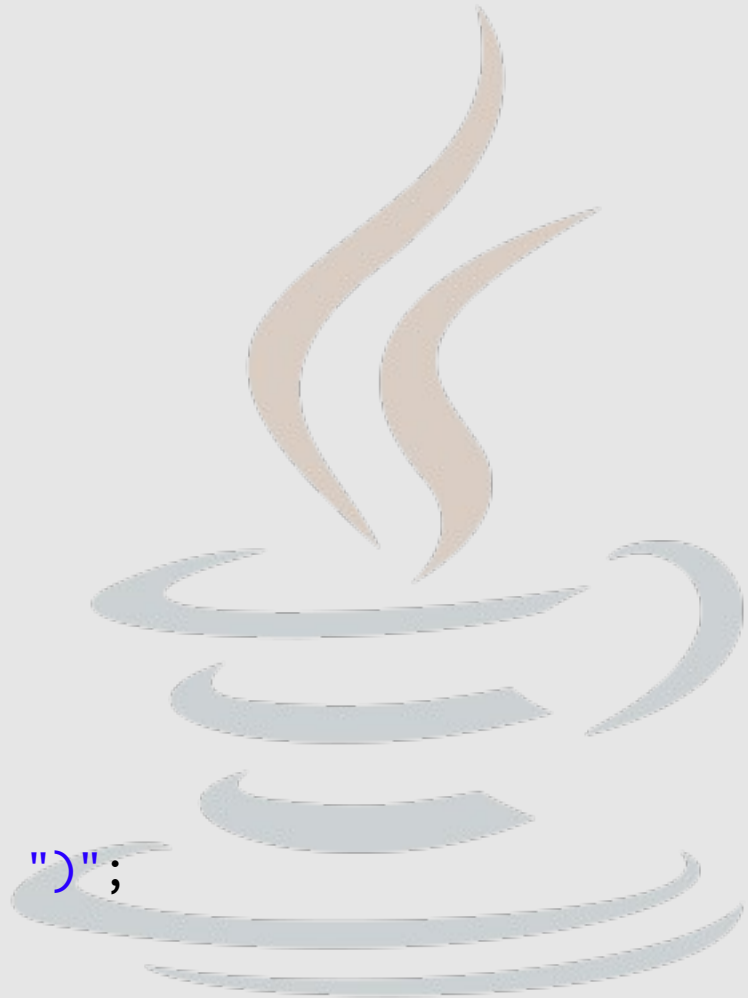
JAXB (*Java Architecture for XML Binding*) to standard serializacji XML dla obiektów Javy. Został on zintegrowany z JDK/JRE od wersji 1.6

```
import java.io.File;

import javax.xml.bind.*;
import javax.xml.bind.annotation.*;

@XmlRootElement
class Person {

    String name;
    int age;

    public String getName() {
        return name;
    }
```



JAXB

```
    @XmlElement
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    @XmlElement
    public void setAge(int age) {
        this.age = age;
    }

    public void toString() {
        return this.name + " (" + this.age + ")";
    }
}
```



JAXB

```
public class JAXBExample{
    public static void main(String[] args) throws JAXBException {

        Person p = new Person();
        p.setName("Barnaba");
        p.setAge(33);

        File f = new File("person.xml");
        JAXBContext ctx = JAXBContext.newInstance(Person.class);
        Marshaller marshaller = ctx.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        marshaller.marshal(p, System.out);
        marshaller.marshal(p, f);
        p = null;
        Unmarshaller unmarshaller = ctx.createUnmarshaller();
        p = (Person)unmarshaller.unmarshal(f);
        System.out.println(p);
    }
}
```

SERIALIZACJA I XML

Inna metoda serializacji niektórych obiektów do plików tekstowych w formacie XML:

XMLEncoder:

```
XMLEncoder e = new XMLEncoder(new FileOutputStream("jbutton.xml"));
e.writeObject(new JButton("Hello world"));
e.close();
```

XMLDecoder:

```
XMLDecoder d = new XMLDecoder(new FileInputStream("jbutton.xml"));
obj = d.readObject();
d.close();
```

ANT

Ant jest narzędziem umożliwiającym automatyzację procesów związanych z budowaniem programów. Jego podstawowe cechy to:

- konfiguracja zadań zapisana w formacie XML,
- wieloplatformowość – m. in. Linux, Unix (np. Solaris and HP-UX), Windows 9x i NT, OS/2 Warp, Novell Netware 6 oraz MacOS X.
- rozszerzalność w oparciu o klasy napisane w Javie.

Ant jest rozwijany w ramach Apache Software Foundation. Strona domowa projektu: http://ant.apache.org.

ANT

Przykładowy plik konfiguracyjny dla anta (domyślnie **build.xml**)

```
<project name="project" default="default" basedir=".">
    <description>
            jakis opis
    </description>

    <target name="default" depends="depends" description="description">
       <jar destfile="bendmetter.jar" includes="**/*.class"
                                 excludes="tests/*.*" basedir="bin"></jar>
    </target>

    <target name="depends">
    </target>
</project>
```

Aby go "wykonać" wpisujemy w konsoli polecenie **ant**.

DZIĘKUJĘ ZA UWAGĘ

JAVA I BAZY DANYCH

ZAGADNIENIA:

- wprowadzenie;
- JDBC;
- komunikacja z bazą danych;
- HSQLDB.

MATERIAŁY:

http://docs.oracle.com/javase/tutorial/jdbc/basics/index.html

BAZY DANYCH - SQL

Tabela				
id	Imie	Nazwisko	Wiek	numer
1	Barbara	Tokarska	47	FSX458721
2	Tomasz	Czyżewski	24	HGJ874398
3	Jerzy	Pietras	35	VTF937836
4	Agata	Kałuża	17	JKD259077

```
SELECT Imie,Nazwisko FROM Tabela WHERE id = 3;

UPDATE Tabela SET Imie='Marek' WHERE id=4;

DELETE FROM Tabela WHERE Imie='Tomasz';

INSERT INTO Tabela (id, Imie) VALUES (5, 'Dorota');
```

JDBC

Java Database Connectivity (JDBC) to specyfikacja określająca zbiór klas i interfejsów napisanych w Javie, które mogą być wykorzystane przez programistów tworzących oprogramowanie korzystające z baz danych. Implementacja JDBC jest dostarczany przez producentów baz danych.

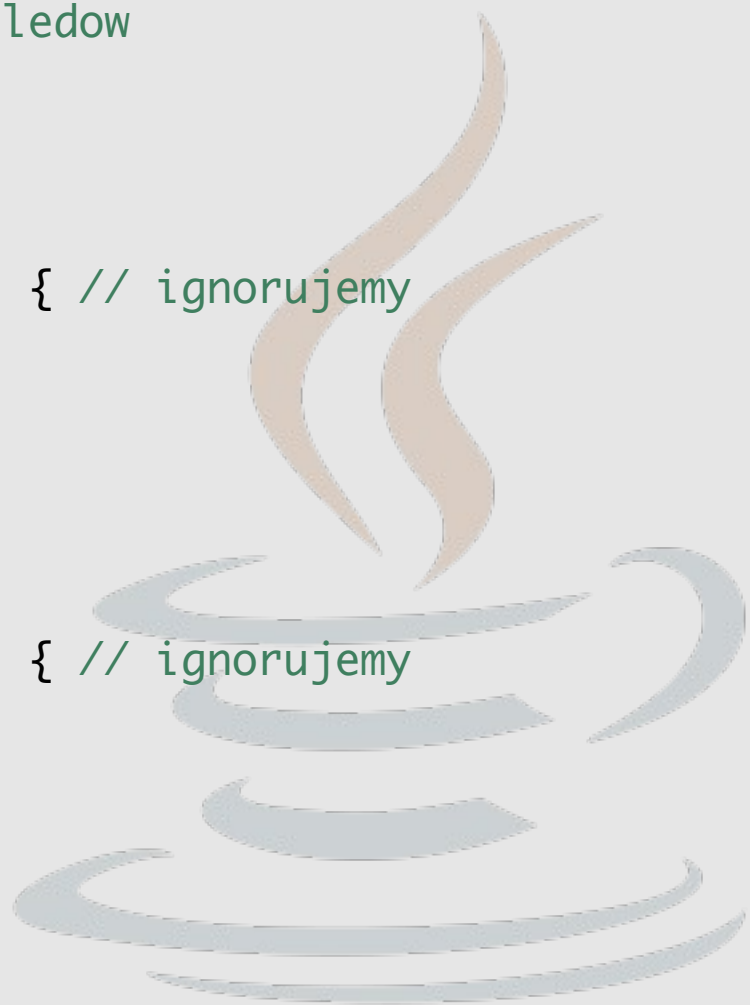
Jedną z ważniejszych zalet takiego rozwiązania jest ukrycie przed programistą kwestii technicznych dotyczących komunikacji z bazą danych. Dzięki temu ten sam program napisany w Javie może współpracować z różnymi systemami baz danych (np. Oracle, Sybase, IBM DB2). Wystarczy podmienić odpowiednie biblioteki implementujące JDBC.

PRZYKŁAD

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestDriver {
    public static void main(String[] args) {
        Statement stmt = null;
        ResultSet rs = null;
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            Connection con = DriverManager.getConnection(
                    "jdbc:mysql://localhost/test?user=monty&password=");
            stmt = con.createStatement();
            rs = stmt.executeQuery("SELECT * FROM Tabela");
            while (rs.next()) {
                System.out.println(rs.getString("Imie"));
            }
        }
```

PRZYKŁAD

```
        catch (Exception ex) {  // obsluga bledow
        } finally {  // zwalnianie zasobow
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException sqlEx) { // ignorujemy
                }
                rs = null;
            }
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException sqlEx) { // ignorujemy
                }
                stmt = null;
            }
            // analogicznie con.close();
        }
    }
}
```



NAWIĄZANIE POŁĄCZENIA

Istnieją dwie metody nawiązania połączenia z bazą danych:

- za pomocą klasy **DriverManager**:

```
String url = "jdbc:odbc:bazadanych";
Connection con = DriverManager.getConnection(url, "login", "haslo");
```

- za pomocą klasy **DataSource** i usługi JNDI:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/MojaDB");
Connection con = ds.getConnection("myLogin", "myPassword");
```

DriverManager

`DriverManager` jest tradycyjną warstwą zarządzającą JDBC pomiędzy użytkownikiem a sterownikiem. Aktualną listę dostępnych sterowników można uzyskać za pomocą metody: Enumeration `DriverManager.getDrivers()`.

Aby załadować dodatkowy sterownik należy skorzystać z metody: `Class Class.forName(String)` podając jako argument klasę ze sterownikiem np:

```
Class.forName("jdbc.odbc.JdbcOdbcDriver"); // protokół jdbc:odbc
Class.forName("com.mysql.jdbc.Driver"); //protokół jdbc:mysql..
```

DataSource

DataSource reprezentuje źródło danych. Zawiera informacje identyfikujące i opisujące dane. Obiekt **DataSource** współpracuje z technologią *Java Naming and Directory Interface* (JNDI), jest tworzony i zarządzany niezależnie od używającej go aplikacji.

Korzystanie ze źródła danych zarejestrowanego w JNDI zapewnia:

- brak bezpośredniego odwołania do sterownika przez aplikację,
- umożliwia implementację grupowania połączeń (pooling) oraz rozproszonych transakcji.

Te cechy sprawiają, że korzystanie z klasy **DataSource** jest zalecaną metodą tworzenia połączenia z bazą danych, szczególnie w przypadku dużych aplikacji rozproszonych.

PRZESYŁANIE ZAPYTAŃ

Do przesyłania zapytań do bazy danych służą obiekty klasy:

- `Statement` – typowe pytania (bezparametrowe),
- `PreparedStatement` – prekompilowany pytania zawierające parametry wejściowe,
- `CallableStatement` – procedury zapisane w bazie danych.

Obiekt `Statement` tworzy się w ramach nawiązanego wcześniej połączenia:

```
Connection con = DriverManager.getConnection(url, login, pass);
Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT INTO table(name, price) VALUE 'ser', 2.0");
```

PRZESYŁANIE ZAPYTAŃ

Do przesyłania zapytań służą metody:

- `executeQuery` – pytania zwracające dane: SELECT,
- `executeUpdate` – pytania zmieniające dane: INSERT, UPDATE, CREATE TABLE, ...
- execute – dowolne zapytania. Dzięki tej instrukcji można wykonać sekwencje pytań, przekazać i odebrać dodatkowe parametry.

W ramach jednego obiektu Statement można wykonać sekwencyjnie kilka zapytań. Po zakończeniu używania obiektu zaleca się wywołanie metody `close()`.

ODBIERANIE WYNIKÓW

Wyniki zwrócone w wyniku wykonania zapytania są dostępne poprzez obiekt typu **ResultSet**. Przykład:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM table");
while (rs.next()) {
    // odebranie i wypisanie wyników w bieżącym rekordzie
    int i = rs.getInt("a");
    String s = rs.getString("b");
    float f = rs.getFloat("c");
    System.out.println("Rekord = " + i + " " + s + " " + f);
}
```

ODBIERANIE WYNIKÓW

	getObject	getURL	getCharacterStream	getRef	getArray	getBlob	getClob	getBinaryStream	getUnicodeStream	getAsciiStream	getTimestamp	getTime	getDate	getBytes	getString	getBoolean	getBigDecimal	getDouble	getFloat	getLong	getInt	getShort	getByte
TINYINT	x														x	x	x	x	x	x	x	x	**X**
SMALLINT	x														x	x	x	x	x	x	x	**X**	x
INTEGER	x														x	x	x	x	x	x	**X**	x	x
BIGINT	x														x	x	x	x	x	**X**	x	x	x
REAL	x														x	x	x	x	**X**	x	x	x	x
FLOAT	x														x	x	x	**X**	x	x	x	x	x
DOUBLE	x														x	x	x	**X**	x	x	x	x	x
DECIMAL	x														x	x	**X**	x	x	x	x	x	x
NUMERIC	x														x	x	**X**	x	x	x	x	x	x
BIT	x														x	**X**	x	x	x	x	x	x	x
BOOLEAN	x															**X**	x	x	x	x	x	x	x
CHAR	x		x						x	x	x	x	x		**X**	x	x	x	x	x	x	x	x
VARCHAR	x		x						x	x	x	x	x		**X**	x	x	x	x	x	x	x	x
LONGVARCHAR	x		**X**						**X**	**X**	x	x	x		x	x	x	x	x	x	x	x	x
BINARY	x		x					x	x	x				**X**	x								
VARBINARY	x		x					x	x	x				**X**	x								
LONGVARBINARY	x		x					**X**	x	x				x	x								
DATE	x										x		**X**		x								
TIME	x										x	**X**			x								
TIMESTAMP	x										**X**	x	x		x								
CLOB	x						**X**																
BLOB	x					**X**																	
ARRAY	x				**X**																		
REF	x			**X**																			
STRUCT	**X**																						
DATALINK	x	**X**													x								
JAVA_OBJECT	**X**																						

HSQLDB

HSQLDB to system zarządzania relacyjnymi bazami danych w całości napisany w Javie (open source).

Strona domowa projektu: http://www.hsqldb.org.

Niektóre własności:

- obsługa SQL'a,
- obsługa transakcji (COMMIT, ROLLBACK, SAVEPOINT), zdalnych procedur i funkcji (mogą być pisane w Javie), triggerów itp.
- możliwość dołączania do programów i appletów. Działanie w trybie read-only,
- rozmiar tekstowych i binarnych danych ograniczony przez rozmiar pamięci.

HSQLDB - SERWER

- HyperSQL Server – tryb preferowany. Klasa `org.hsqldb.server.Server`,
- HyperSQL WebServer – używany, jeśli serwer może używać tylko protokołu HTTP lub HTTPS. W przeciwnym razie niezalecany. Klasa `org.hsqldb.server.WebServer`,
- HyperSQL Servlet – używa tego samego protokołu co WebServer. Wymaga osobnego kontenera serwletów (np. Tomcat).
- *Stand-alone* – "serwer" uruchamiany w ramach procesu, w którym nawiązujemy połączenie z bazą.

Wszystkie tryby pracy serwera umożliwiają korzystanie z JDBC.

HSQLDB - POŁĄCZENIE

```
try {
     Class.forName("org.hsqldb.jdbc.JDBCDriver" );
 } catch (Exception e) {
     System.err.println("ERROR: failed to load HSQLDB JDBC driver.");
     e.printStackTrace();
     return;
 }

Connection c =
DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/xdb", "SA", "");

Connection c =
DriverManager.getConnection("jdbc:hsqldb:http://localhost/xdb", "SA", "");

Connection c =
DriverManager.getConnection("jdbc:hsqldb:file:/opt/db/testdb;shutdown=true
", "SA", "");
```

HSQLDB *stand-alone*

W trybie *stand-alone* „serwer" bazy danych działa w ramach tej samej wirtualnej maszyny Javy, co korzystający z niego program „kliencki".

Przykład uruchomienia bazy:

```
Connection c = DriverManager.getConnection(
                "jdbc:hsqldb:file:/opt/db/testdb", "sa", "");
```

Niewielkie bazy danych mogą być uruchamiane do pracy w pamięci operacyjnej komputera:

```
Connection c = DriverManager.getConnection(
                "jdbc:hsqldb:mem:testdb", "sa", "");
```

W obecnej wersji HSQLDB istnieje możliwość jednoczesnego używania wielu „serwerów" baz danych działających w trybie *stand-alone*.

PODSUMOWANIE

Podstawowe klasy służące do interakcji z systemami bazodanowymi są udostępniane za pośrednictwem obiektów: `Connection`, `Statement` i `ResultSet`. Wszystkie te klasy należą do pakietu `java.sql`. Klasy rozszerzające funkcjonalność JDBC (np. `DataSource`) znajdują się w pakiecie `javax.sql`.

Korzystanie z interfejsu JDBC umożliwia jednolity sposób dostępu do różnych systemów bazodanowych. Jako przykład przedstawiono bazę HSQLDB. Jej ważną zaletą jest możliwość działania w ramach jednej Wirtualnej Maszyny Javy wraz z programem klienckim.

DZIĘKUJĘ ZA UWAGĘ

REFLEKSJE

ZAGADNIENIA:

- Klasy,
- Atrybuty i metody,
- Dynamiczne klasy proxy.

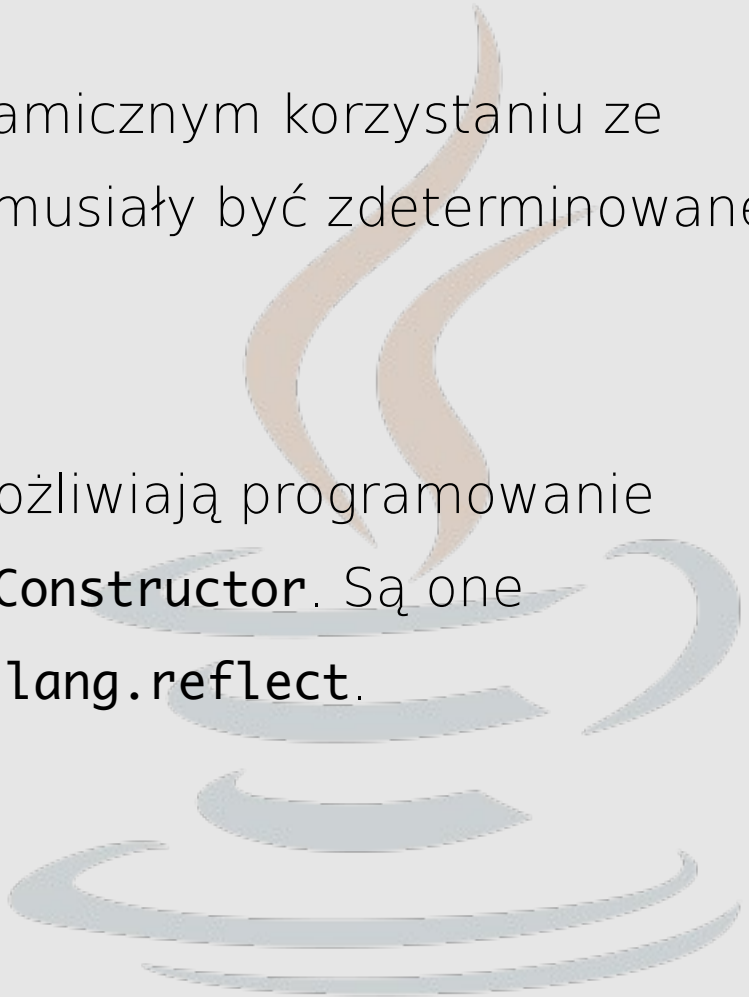
MATERIAŁY:

http://docs.oracle.com/javase/tutorial/reflect/

WPROWADZENIE

Programowanie refleksyjne polega na dynamicznym korzystaniu ze struktur języka programowania, które nie musiały być zdeterminowane w momencie tworzenia oprogramowania.

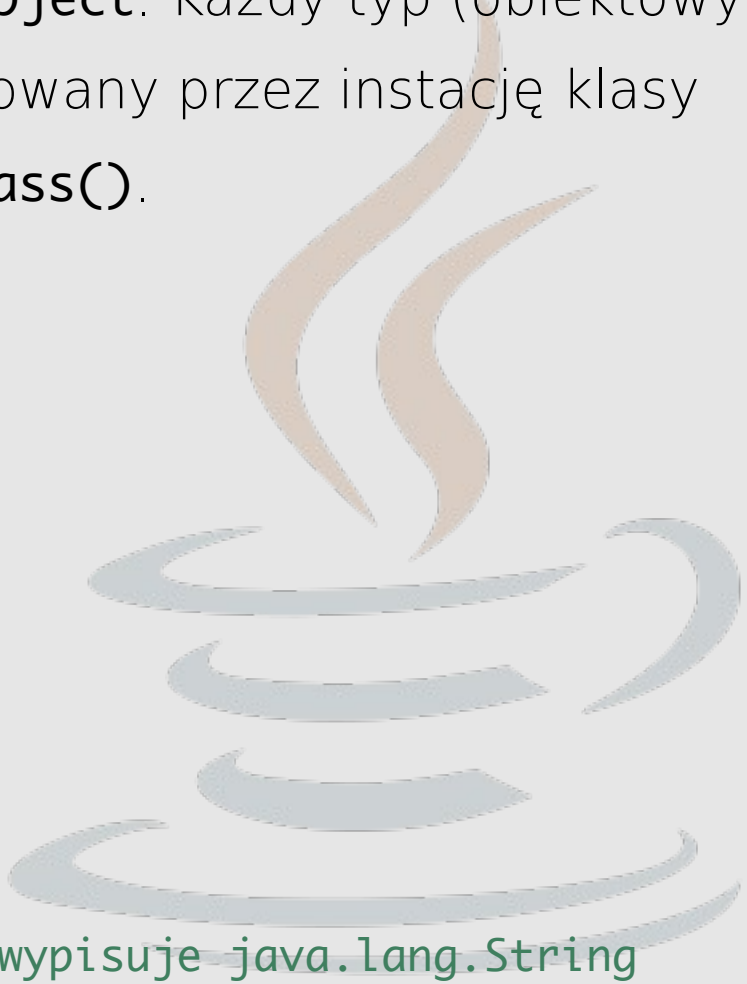
Najważniejsze klasy języka Java, które umożliwiają programowanie refleksyjne to `Class`, `Field`, `Method`, `Array`, `Constructor`. Są one zgrupowane w pakietach `java.lang` i `java.lang.reflect`.



KLASY

Każdy obiekt w Javie jest instancją klasy **Object**. Każdy typ (obiektowy, prymitywny, tablicowy itp.) jest reprezentowany przez instację klasy **Class**, którą uzyskujemy za pomocą **getClass()**.

```
import java.util.HashSet;
import java.util.Set;

enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

public class ReflectionExample {
    public static void main(String[] args){
        Class c;
        c = "foo".getClass();
        System.out.println(c.getName()); // wypisuje java.lang.String
```



KLASY

```
        c = "foo".getClass();
        System.out.println(c.getName()); // wypisuje java.lang.String

        c = System.out.getClass();
        System.out.println(c.getName()); // wypisuje java.io.PrintStream

        c = Day.SUNDAY.getClass();
        System.out.println(c.getName()); // wypisuje Day

        byte[] bytes = new byte[1024];
        c = bytes.getClass();
        System.out.println(c.getName()); // wypisuje [B

        Set<String> s = new HashSet<String>();
        c = s.getClass();
        System.out.println(c.getName()); // wypisuje java.util.HashSet
    }
}
```

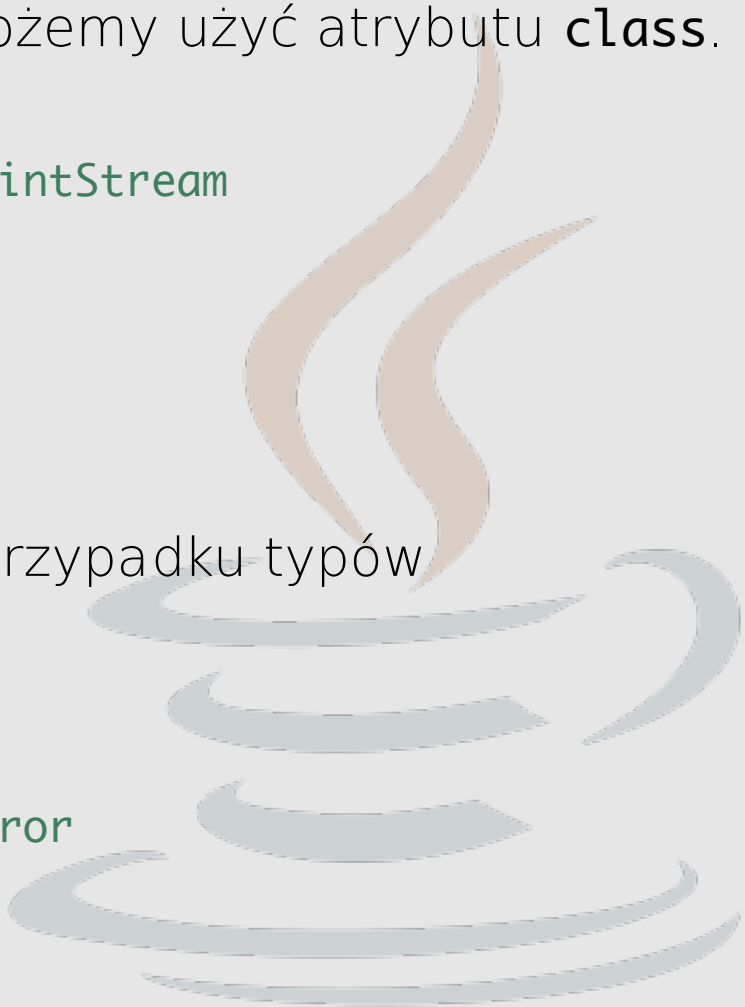
class

Jeśli nie mamy obiektu (instancji klasy) możemy użyć atrybutu **class**.

```
c = java.io.PrintStream.class; // java.io.PrintStream

c = int[][][].class; // [[[I

c = boolean.class;  // boolean
```

Ten sposób jest szczególnie użyteczny w przypadku typów prymitywnych:

```
boolean b;
Class c = b.getClass();   // compile-time error
```

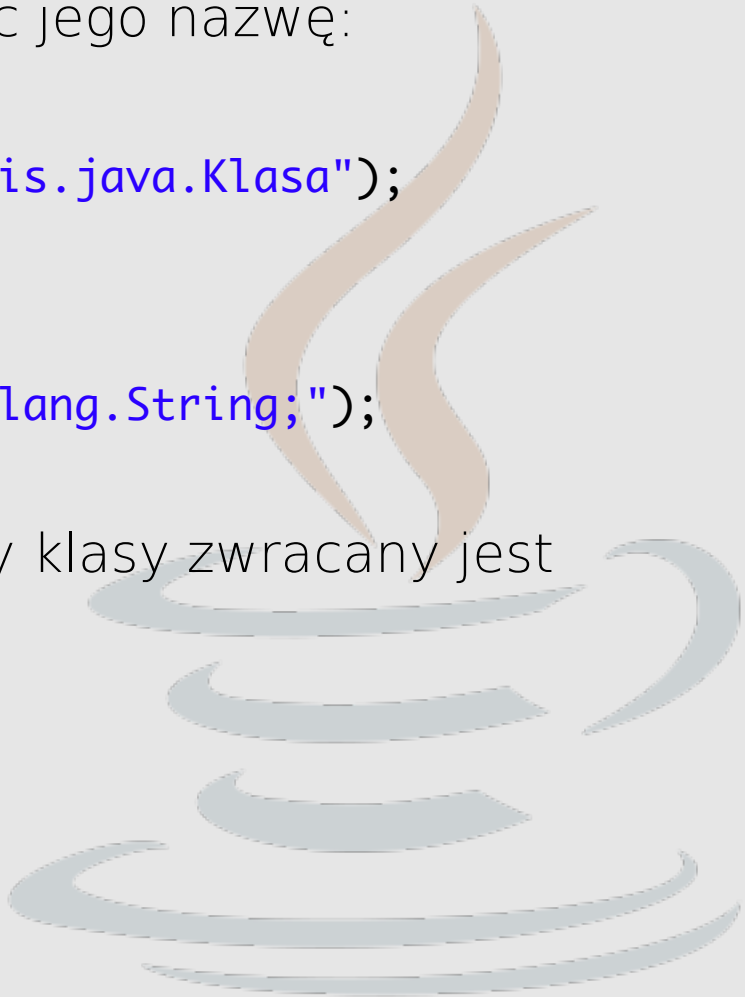


Class.forName()

Obiekt **Class** można także otrzymać znając jego nazwę:

```
Class cMyClass = Class.forName("pl.edu.uj.fais.java.Klasa");

Class cDoubleArray = Class.forName("[D");

Class cStringArray = Class.forName("[[Ljava.lang.String;");
```

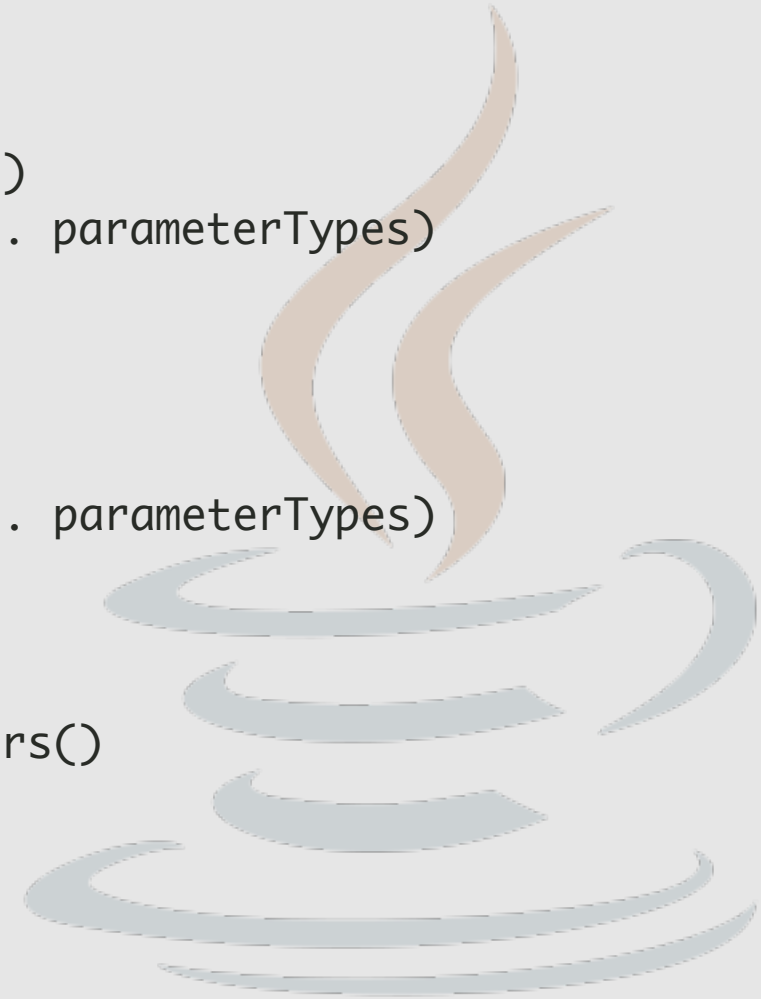
W przypadku podania niepoprawnej nazwy klasy zwracany jest wyjątek **ClassNotFoundException**.



KLASY

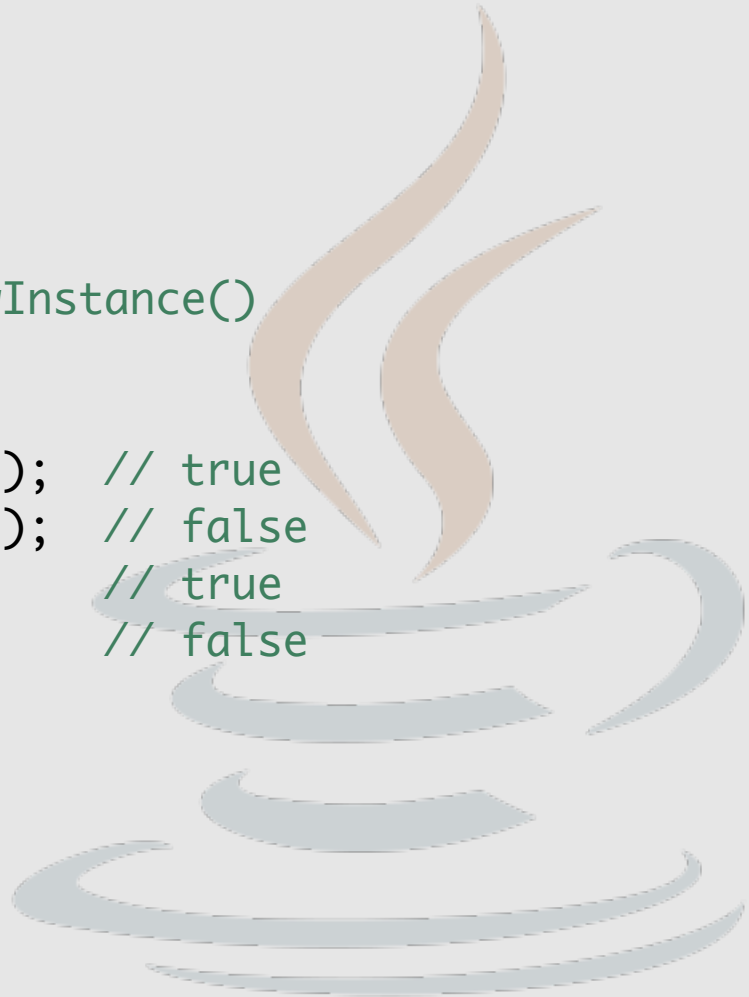
Wybrane metody klasy Class:

- `static Class<?> forName(String className)`
- `Constructor<T> getConstructor(Class<?>... parameterTypes)`
- `Constructor<?>[] getConstructors()`
- `Field getField(String name)`
- `Field[] getFields()`
- `Class<?>[] getInterfaces()`
- `Method getMethod(String name, Class<?>... parameterTypes)`
- `Method[] getMethods()`
- `int getModifiers()`
- `Class<? super T> getSuperclass()`
- `TypeVariable<Class<T>>[] getTypeParameters()`
- `boolean isArray()`
- `boolean isInterface()`
- `boolean isPrimitive()`
- `T newInstance()`



KLASY

```
class C1{}
class C2 extends C1{}
...
C1 o1 = new C1(); // rownowazne C1.class.newInstance()
C2 o2 = new C2();

o1.getClass().isAssignableFrom(o2.getClass());  // true
o2.getClass().isAssignableFrom(o1.getClass());  // false
o1.getClass().isInstance(o2);                   // true
o2.getClass().isInstance(o1);                   // false
...
```



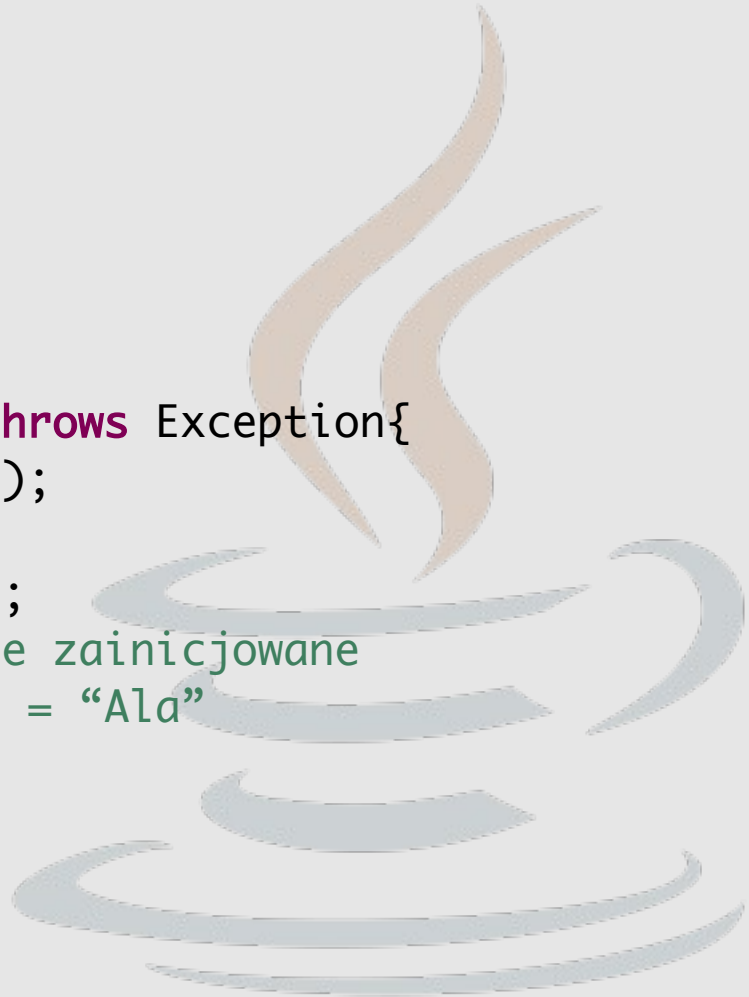
ATRYBUTY

Atrybut jest reprezentowany przez instancję klasy **`Field`**. Wybrane metody:

- `Object get(Object obj) // zwraca wartość atrybutu w obiekcie obj`
- `int getInt(Object obj) // zwraca wartość atrybutu typu int`
- `int getModifiers() // modyfikatory dostępu: public, private, ...`
- `Class<?> getType() // klasa reprezentująca typ atrybutu`
- `void set(Object obj, Object value) // ustawia wartosć atrybutu w obiekcie obj`
- `void setInt(Object obj, int i) // ustawia wartosć atrybutu typu int`

ATRYBUTY

```
import java.lang.reflect.Field;

public class FieldExample {
    public static String s;
    public int i;

    public static void main(String[] args) throws Exception{
        FieldExample fex = new FieldExample();
        Field f;
        f = FieldExample.class.getField("s");
        f.get(null)); // zwróci null bo s nie zainicjowane
        f.set(null, "Ala"); // FieldExamle.s = “Ala”

        f = fex.getClass().getField("i");
        f.set(fex, 10); // fex.i = 10
    }
}
```



METODY

Metoda jest reprezentowana przez instancję klasy **Method**. Wybrane metody:

- `Class<?>[] getExceptionTypes() // zwraca clasy reprezentujące zadeklarowane, zwracane wyjątki`
- `int getModifiers() // private, public, static, ...`
- `Object invoke(Object obj, Object... args) // wywołuje metodę na rzecz obiektu obj. Argumenty wywołania znajdują się w tablicy args. Watrość zwracana jest wynikiem działania wywołanej metody.`
- `boolean isVarArgs() // czy metoda ma nieokreśloną liczbę argumentów`

invoke()

Przykład:

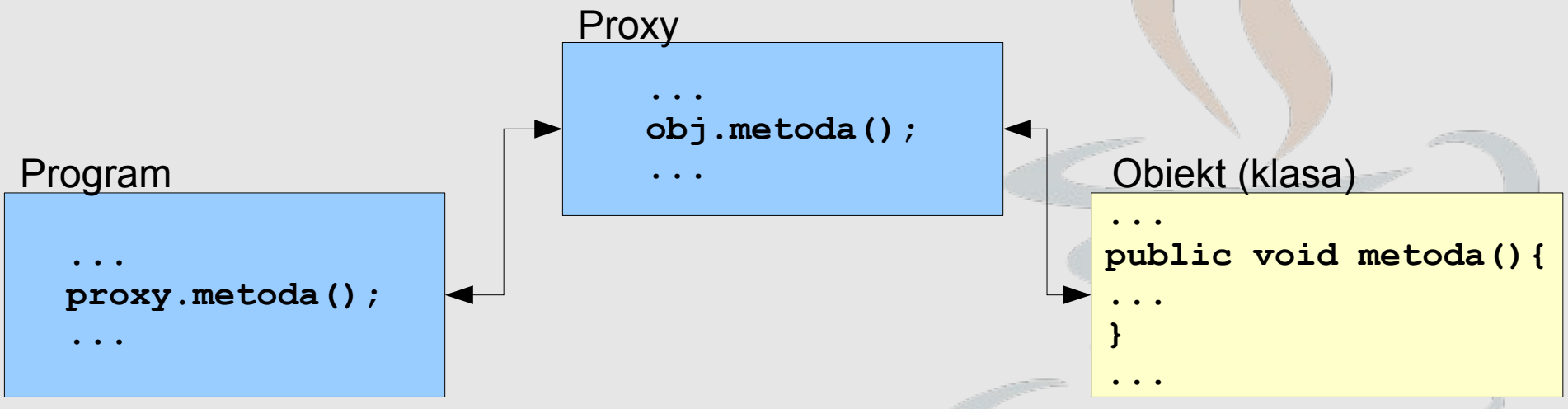
```
...
Method m = Class.forName("MyClass").getDeclaredMethod(
                                                "example3", null);
m.invoke(null, null);
...
```

metoda **invoke** może zwrócić kilka rodzajów wyjątków związanych z dostępem do metody i zgodnością typów argumentów. Wady wynikające z używania invoke:

- brak kontroli (w trakcie kompilacji) typów przekazywanych parametrów,
- ograniczenie obsługi wyjątków do Throwable.

Rozwiązaniem tych problemów może być tzw. *Dynamic Proxy Class*.

DYNAMIC PROXY



klasa Proxy powinna być tworzona dynamicznie.

PROXY

Klasa **Proxy** udostępnia metody statyczne służące do tworzenia tzw. dynamicznych klas proxy oraz ich instancji. Utworzenie proxy dla określonego interfejsu (np. **MyInterface**):

```
InvocationHandler handler = new MyInvocationHandler(...);

Class proxyClass = Proxy.getProxyClass(
                                MyInterface.class.getClassLoader(),
                                new Class[] { MyInterface.class });

MyInterface mi = (MyInterface) proxyClass.getConstructor(
        new Class[] { InvocationHandler.class }).newInstance(
                                                new Object[] { handler });
```

PROXY

lub alternatywnie:

```
MyInterface mi = (MyInterface) Proxy.newProxyInstance(
                 MyInterface.class.getClassLoader(),  // loader dla zasobów
                 new Class[] { MyInterface.class },   // tablica interfejsów
                 handler);  // obiekt do którego będą przekazywane
                            // wywołania
```

Stworzono obiekt **mi**, który „z zewnątrz" wygląda jak klasa implementująca MyInterface, natomiast obsługa metod będzie w rzeczywistości realizowana przez **handler**.

PROXY

Obiekt **handler** nie musi implementować **MyInterface**!

Musi za to implementować interfejs **InvocationHandler**, czyli metodę:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Jak to działa? Wszystkie wywołania metod na obiekcie **mi** są przekierowane do metody **invoke** obiektu **handler**, przy czym pierwszym parametrem jest obiekt **proxy**.

PROXY

```
...
Pisarz obj = new PisarzImpl();
Method m = obj.getClass().getMethod(
        "pisz",
        new Class[] {String.class});
m.invoke(obj, new Object[]{"hello world"});
...
```

```
class PisarzImpl implements Pisarz {
    ...
    public void pisz(String s){
        ...
    }
    ...
}
```

```
interface Pisarz {
    public void pisz(String s);
}
```

PROXY

```
...
Pisarz p = (Pisarz) Proxy.newProxyInstance(
    Pisarz.class.getClassLoader(),
    new Class[] { Pisarz.class },
    new MyHandler());
p.pisz("hello world");
...
```

```
class PisarzImpl implements Pisarz {
    ...
    public void pisz(String s){
        ...
    }
    ...
}
```

```
class MyHandler{
  private Pisarz obj = new PisarzImpl();
  public static Object invoke(Object proxy,
              Method m, Object[] args)){
    return m.invoke(obj, args);
  }
}
```

```
interface Pisarz {
    public void pisz(String s);
}
```

DZIĘKUJĘ ZA UWAGĘ

JAVA NATIVE INTERFACE

ZAGADNIENIA:

- wprowadzenie
- przykład HelloWorld
- podstawy JNI (przekazywanie zmiennych, stringów, obiektów)

MATERIAŁY:

http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html

WPROWADZENIE

Java Native Interface umożliwia:

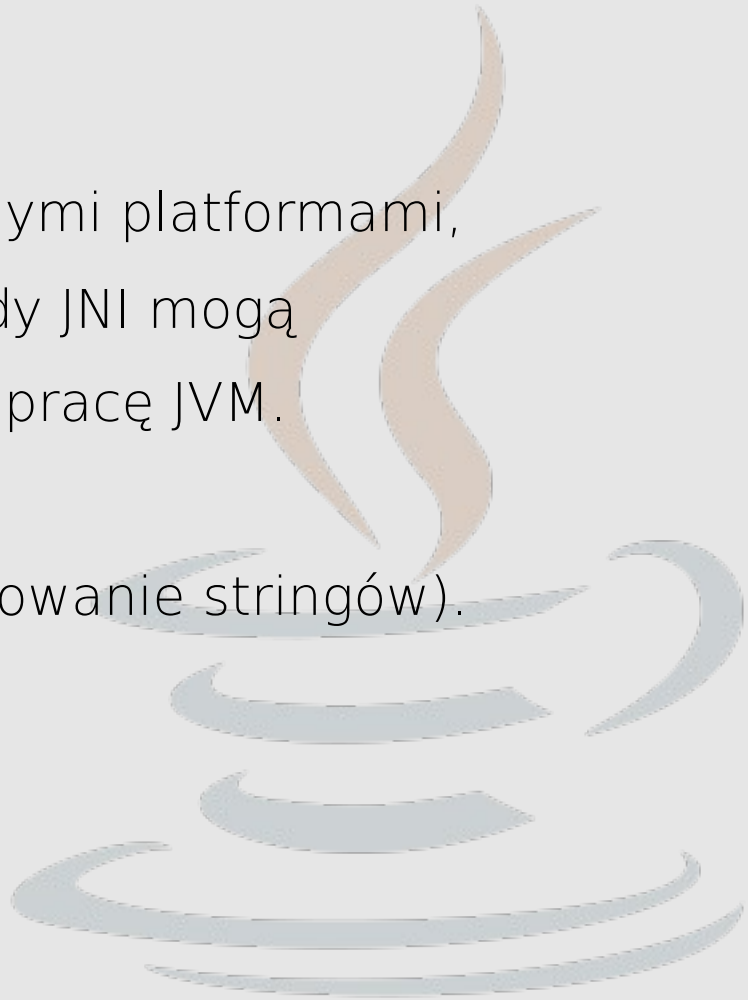
- korzystanie z bibliotek systemowych (napisanych np. w C/C++) wewnątrz programu w Javie.
- korzystanie z obiektów Javy wewnątrz programów w innych językach programowania (np. C/C++).

Najczęściej JNI wykorzystuje się do realizacji niskopoziomowej komunikacji IO (np. obsługa interfejsów USB, bluetooth), która nie może być zrealizowana z poziomu JVM, gdyż nie ma ona bezpośredniego dostępu do sprzętu.

WPROWADZENIE

Najważniejsze ograniczenia i wady JNI.

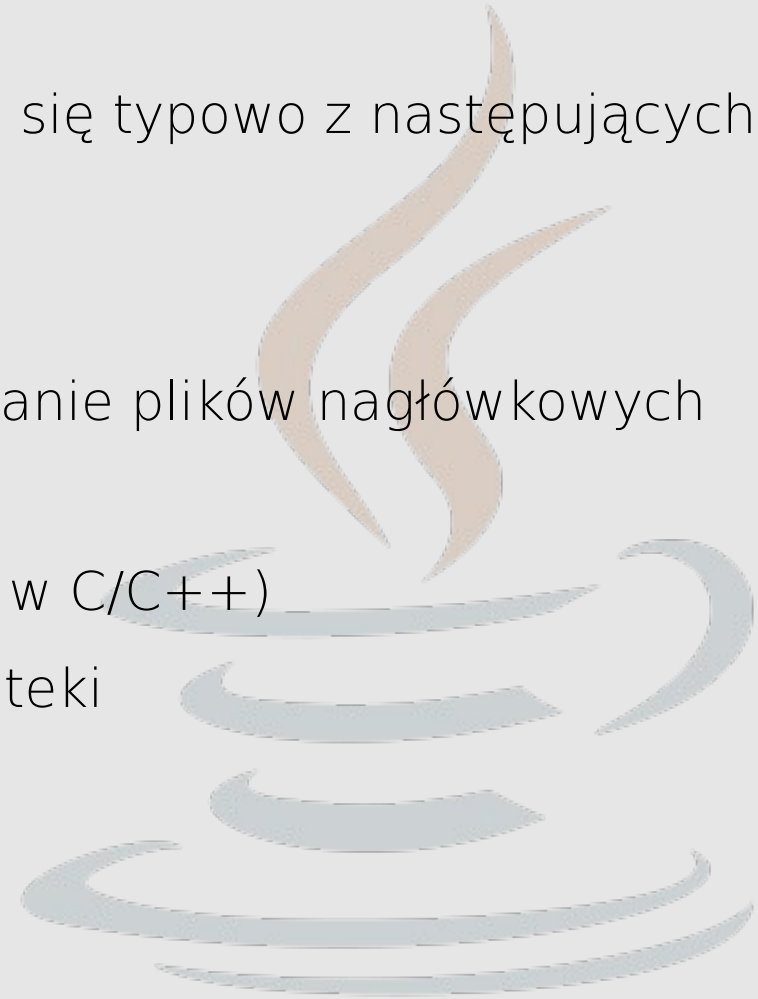
- brak przenośności kodu pomiędzy różnymi platformami,
- utrudnione debugowanie – drobne błędy JNI mogą "niedeterministycznie" destabilizować pracę JVM.
- brak garbage collectora,
- problemy z konwersją danych (np. kodowanie stringów).



HELLO WORLD

Stworzenie aplikacji używającej JNI składa się typowo z następujących kroków:

- napisanie programu w Javie
- skompilowanie programu I wygenerowanie plików nagłówkowych dla metod natywnych
- implementacja metod natywnych (np. w C/C++)
- kompilacja metod natywnych do biblioteki
- uruchomienie programu.



PROGRAM W JAVIE

```
public class JNIHello {

    static {
          System.loadLibrary("hello"); // nazwa biblioteki z kodem
                    // natywnym, moze zalezec od systemu operacyjnego
    }

    // deklaracja metody natywnej (ktora znajduje się w bibliotece)
    private native void sayHello();

    public static void main(String[] args) {
        JNIHello h = new JNIHello();
        h.sayHello();  // wywolanie metody natywnej
    }
}
```

KOMPILACJA

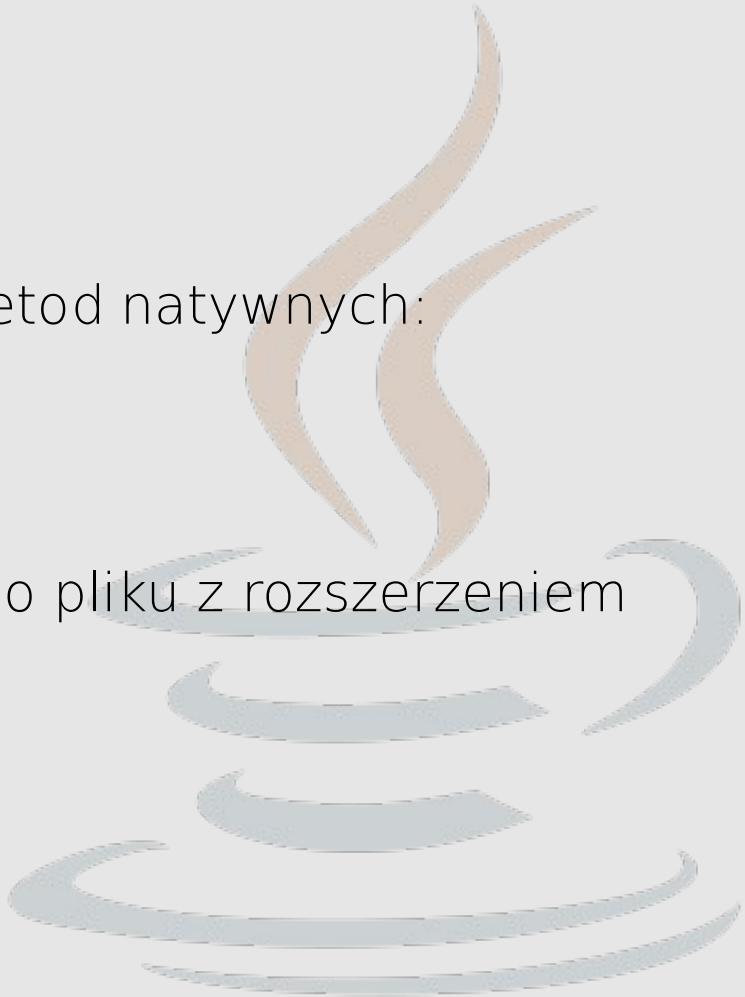
Kompilacja programu:

`javac JNIHello.java`

Generowanie plików nagłówkowych dla metod natywnych:

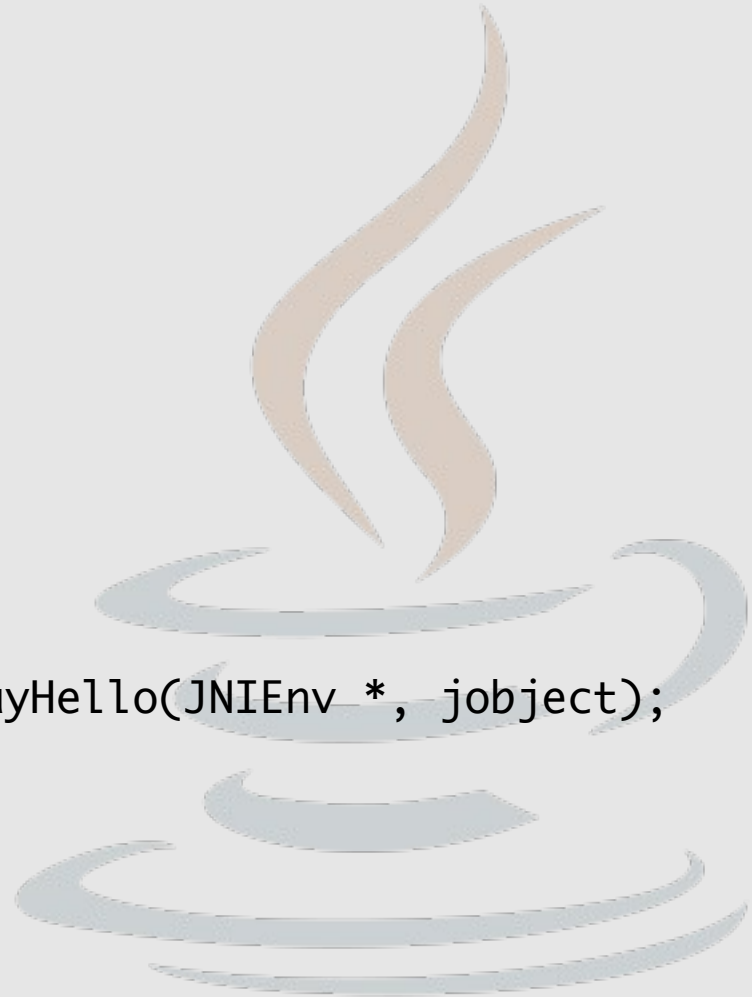
`javah JNIHello`

Do generowania używamy skompilowanego pliku z rozszerzeniem `class`.



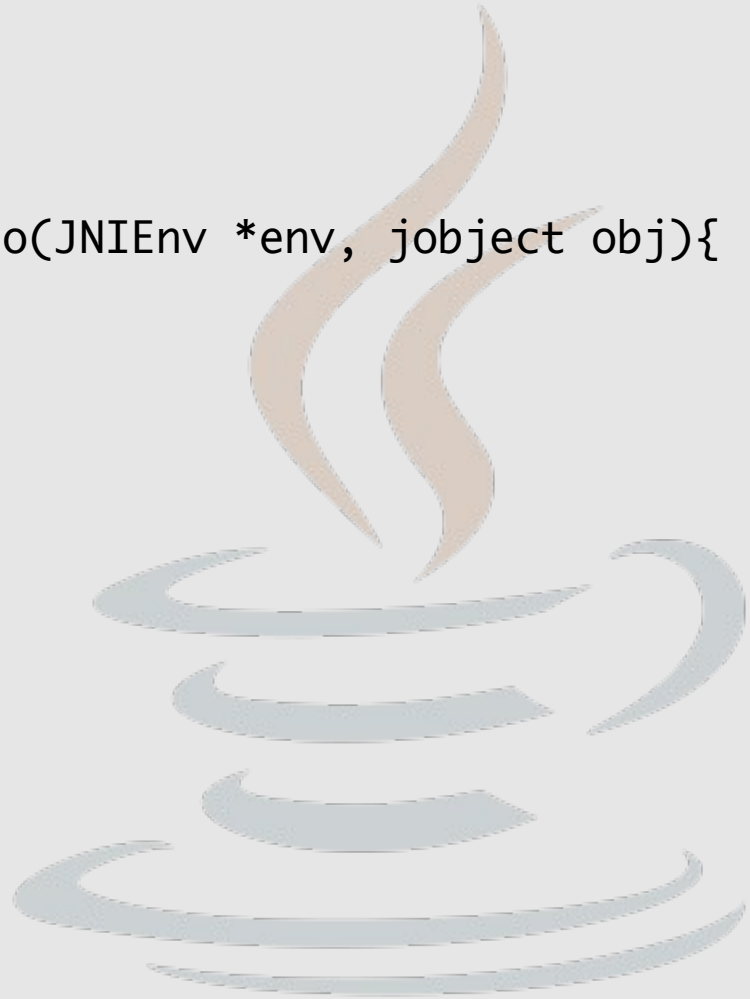
PLIK NAGŁÓWKOWY

```
#include <jni.h>
/* Header for class JNIHello */
#ifndef _Included_JNIHello
    #define _Included_JNIHello
    #ifdef __cplusplus
        extern "C" {
    #endif
    /*
     * Class:     JNIHello
     * Method:    sayHello
     * Signature: ()V
     */
     JNIEXPORT void JNICALL Java_JNIHello_sayHello(JNIEnv *, jobject);
     #ifdef __cplusplus
        }
    #endif
#endif
```



IMPLEMENTACJA

```
#include “JNIHello.h”
#include <stdio.h>
JNIEXPORT void JNICALL Java_JNIHello_sayHello(JNIEnv *env, jobject obj){

   printf("Hello World!\n");

   return;

}
```



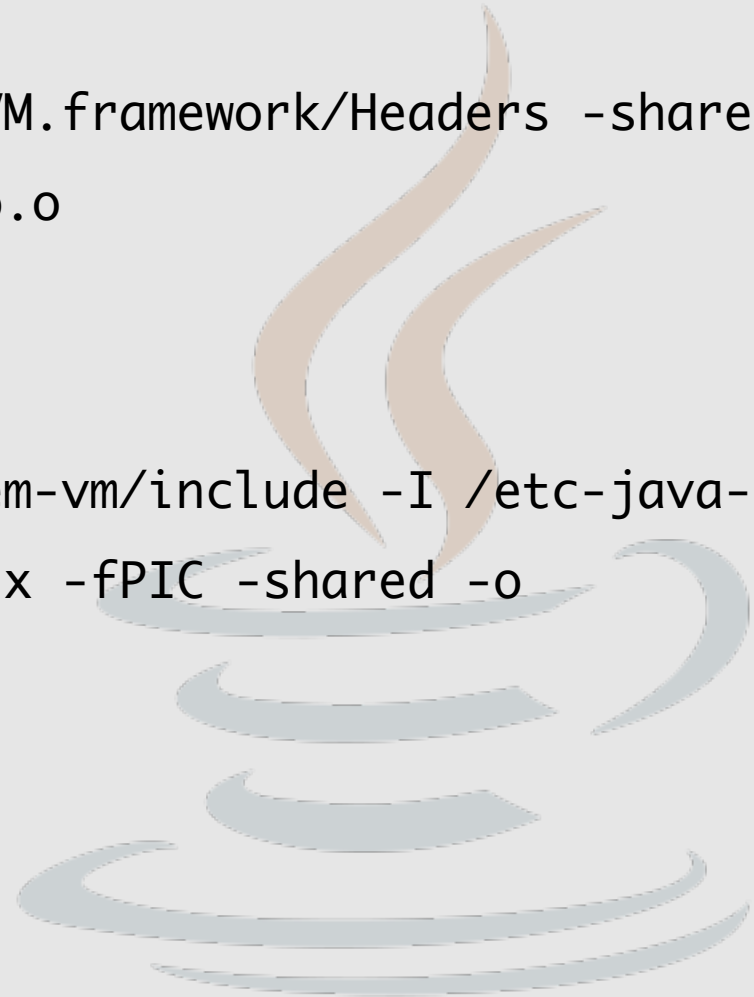
KOMPILACJA BIBLIOTEKI

OSX:

```
gcc -I /System/Library/Frameworks/JavaVM.framework/Headers -shared
-dynamiclib -o libhello.jnilib JNIHello.o
```

Linux:

```
gcc -I /etc-java-config-2/current-system-vm/include -I /etc-java-
config-2/current-system-vm/include/linux -fPIC -shared -o
libhello.so JNIHello.c
```



URUCHOMIENIE

Kompilacja biblioteki

OSX:

```
java JNIHello
```

Linux:

```
java -Djava.library.path=. JNIHello
```

PODSTAWY JNI

JNI definiuje następujące typy natywne:

- jint, jbyte, jshort, jlong, jfloat, jdouble, jchar, jboolean.
- jobject, jclass, jstring, jthrowable
- jarray (jintArray, jbyteArray, jshortArray, jlongArray, jfloatArray, jdoubleArray, jcharArray and jbooleanArray oraz jobjectArray).

Typy natywne mogą być przekazywane jako argumenty wywołania oraz mogą być zwracane. Jednak funkcje natywne operują na swoich natywnych typach (np int[], char*). Dlatego wymagana jest konwersja.

TYPY PRYMITYWNE

```
private native double add(double d1, double d2);

public static void main(String[] args){
    JNIExamples ex = new JNIExamples();
    System.out.println(ex.add(12.5, 5.3));
}
```

```
JNIEXPORT jdouble JNICALL Java_JNIExamples_add(JNIEnv *env, jobject obj,
jdouble d1, jdouble d2){
    printf("Otrzymade argumenty %f; %f \n",d1, d2);
    jdouble res = d1 + d2;
    return res;
}
```

STRINGI

```
private native String sayHello(String msg);
```

```
JNIEXPORT jstring JNICALL Java_JNIExamples_sayHello(JNIEnv *env, jobject
obj, jstring jin){
    // konwersja: (jstring) --> (char*)
    const char *in = (*env)->GetStringUTFChars(env, jin, NULL);
    if (NULL == in) return NULL;

    // dzialanie:
    printf("odebralem %s\n", in);
    // zwolnienie zasobow
    (*env)->ReleaseStringUTFChars(env, jin, in);  // release resources

    // przygotowanie odpowiedzi
    char out[] = "from C";
    // konwersja
    return (*env)->NewStringUTF(env, out);
}
```

STRINGI (UTF-8)

```
// UTF-8 String --> null-terminated char-array
const char * GetStringUTFChars(JNIEnv *env, jstring string, jboolean
*isCopy);

// Informije JVM ze kod natywny nie bedzie potrzebowal utf
void ReleaseStringUTFChars(JNIEnv *env, jstring string, const char *utf);

// Tworzy java.lang.String ze znakow
jstring NewStringUTF(JNIEnv *env, const char *bytes);

// zwraca długosc stringu
jsize GetStringUTFLength(JNIEnv *env, jstring string);

// w buforze zostanie zapisany wskazany fragment str
void GetStringUTFRegion(JNIEnv *env, jstring str, jsize start, jsize
length, char *buf);
```

STRINGI (UNICODE)

```
// Unicode String --> jchar*
const jchar * GetStringChars(JNIEnv *env, jstring string, jboolean
*isCopy);


void ReleaseStringChars(JNIEnv *env, jstring string, const jchar *chars);


jstring NewString(JNIEnv *env, const jchar *unicodeChars, jsize length);


jsize GetStringLength(JNIEnv *env, jstring string);


void GetStringRegion(JNIEnv *env, jstring str, jsize start, jsize length,
jchar *buf);
```

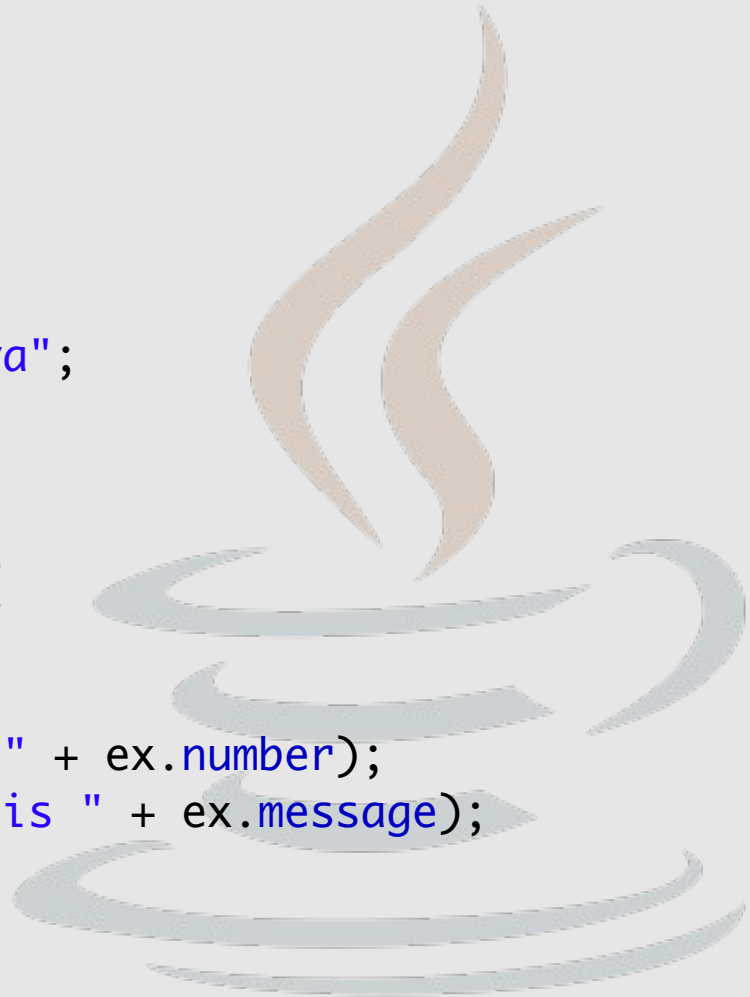
TABLICE

```
private native int[] swap(int[] numbers);
...
   int[] in = {1,2};
   int[] out = ex.swap(in);
   for(int i: out)
     System.out.println(i);
...
```

```
JNIEXPORT jintArray JNICALL Java_JNIExamples_swap(JNIEnv *env,
jobject obj, jintArray ia){
    jint *inArray = (*env)->GetIntArrayElements(env, ia, NULL);
    if (NULL == inArray) return NULL;
    jint outArray[] = {inArray[1], inArray[0]}; // dzialanie
    (*env)->ReleaseIntArrayElements(env, ia, inArray, 0); // zwolnienie
    jintArray out = (*env)->NewIntArray(env, 2);  // wynik
    if (NULL == out) return NULL;
    (*env)->SetIntArrayRegion(env, out, 0 , 2, outArray);  // kopiowanie
    return out;
}
```

DOSTĘP DO OBIEKTÓW

```
public class JNIExamples {
    static {
        System.loadLibrary("examples");
    }

    private int number = 88;
    private String message = "Hello from Java";

    private native void modifyVariables();

    public static void main(String[] args) {
        JNIExamples ex = new JNIExamples();
        ex.modifyVariables();
        System.out.println("In Java, int is " + ex.number);
        System.out.println("In Java, String is " + ex.message);
    }
}
```



DOSTĘP DO OBIEKTÓW

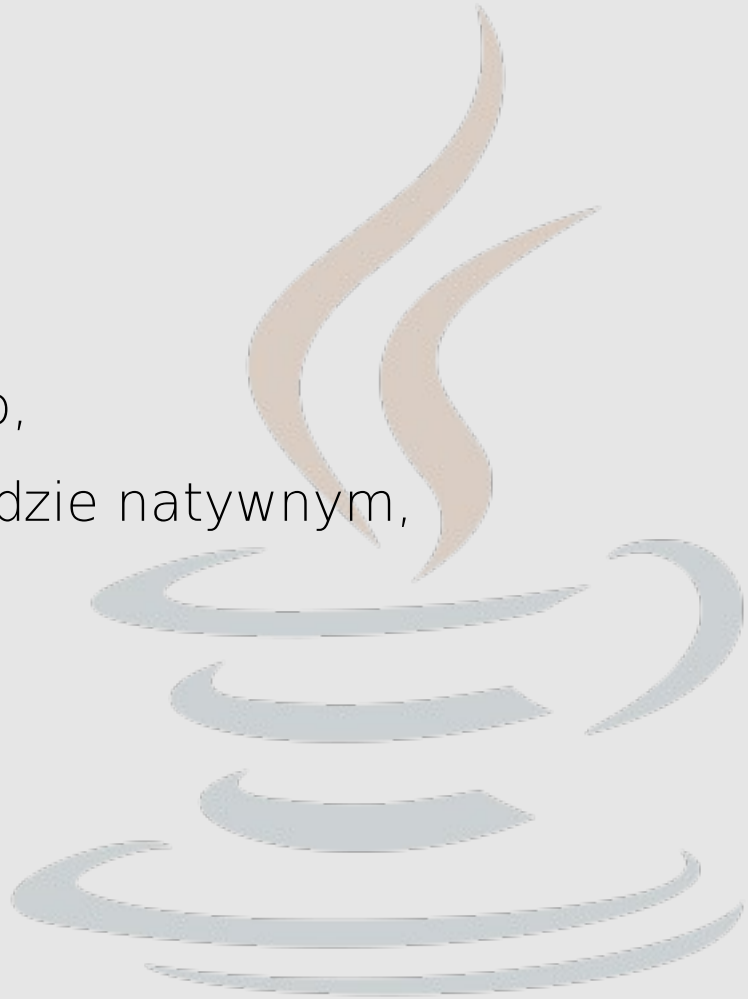
```
JNIEXPORT void JNICALL Java_JNIExamples_modifyVariables(JNIEnv *env,
jobject thisObj) {

    // referencja do obiektu
    jclass thisClass = (*env)->GetObjectClass(env, thisObj);

    // pobranie pola int
    jfieldID fidNumber = (*env)->GetFieldID(env, thisClass,"number","I");
    if (NULL == fidNumber) return;

    // pobranie wartosci
    jint number = (*env)->GetIntField(env, thisObj, fidNumber);
    printf("In C, the int is %d\n", number);

    // zmiana wartosci
    number = 99;
    (*env)->SetIntField(env, thisObj, fidNumber, number);
```

DOSTĘP DO OBIEKTÓW

```
    // pobranie pola String
    jfieldID fidMessage = (*env)->GetFieldID(env, thisClass, "message",
                                                    "Ljava/lang/String;");
    if (NULL == fidMessage) return;

    // pobranie zawartosci
    jstring message = (*env)->GetObjectField(env, thisObj, fidMessage);

    // konwersja
    const char *cStr = (*env)->GetStringUTFChars(env, message, NULL);
    if (NULL == cStr) return;
    printf("In C, the string is %s\n", cStr);
    (*env)->ReleaseStringUTFChars(env, message, cStr); // porzadki
    // nowa wartosc
    message = (*env)->NewStringUTF(env, "Hello from C");
    if (NULL == message) return;
    // modyfikacja - podstawienie nowej wartosci
    (*env)->SetObjectField(env, thisObj, fidMessage, message);
}
```

INNE MOŻLIWOŚCI

- zmiana zmiennych statycznych,
- wywoływanie metod z kodu natywnego,
- tworzenie obiektów Javy (i tablic) w kodzie natywnym,
- zarządzanie referencjami.



DZIĘKUJĘ ZA UWAGĘ

PROGRAMOWANIE DYNAMICZNE

ZAGADNIENIA:

- Plik .class
- ASM

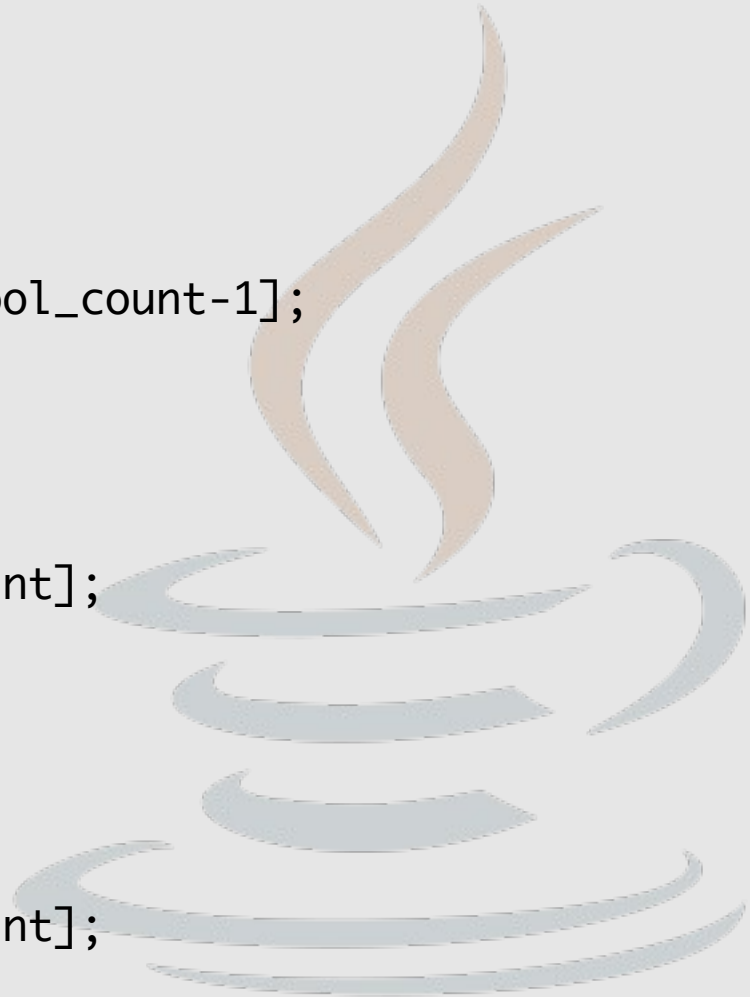
MATERIAŁY:

http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4

http://asm.ow2.org/index.html

PLIK .class

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```



PLIK .class

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

CA FE BA BE – "magic" - identyfikator formatu pliku

00 00 00 33 – numer wersji JVM: (51.0) – zgodność z Java 1.7

Ogólnie JVM 1.k (k>1) obsługuje klasy od 45.0 do 44+k.0 włącznie

00 22 – ilość deklarowanych elementów (*Constant Pool*). Po tej deklaracji następują kolejne, 33 elementy.

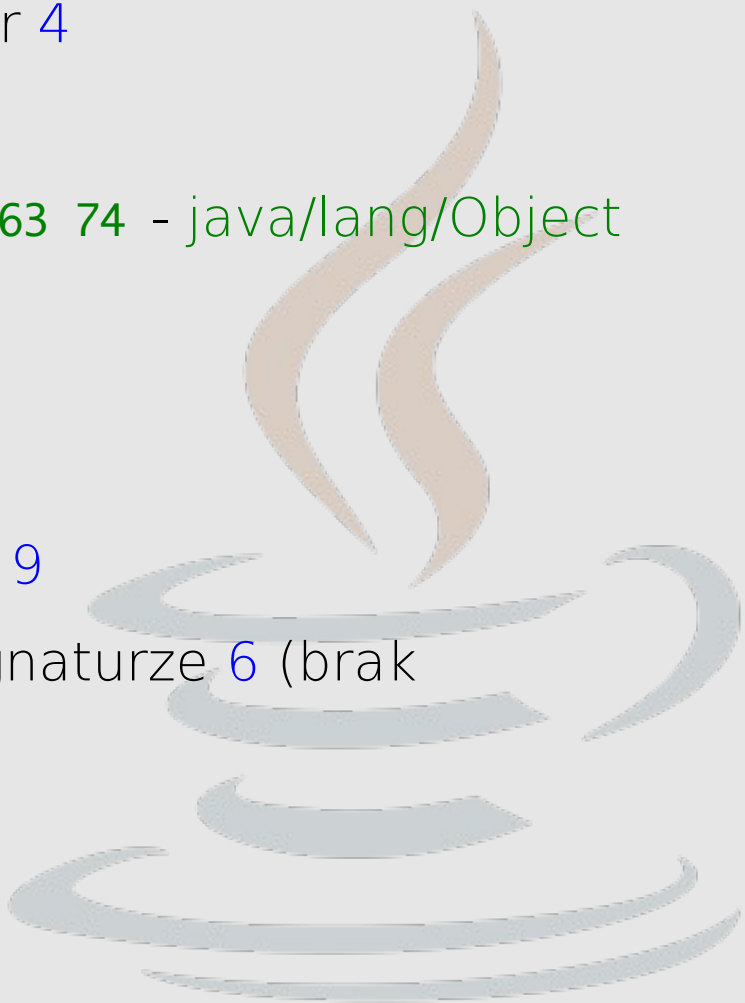
1. 07 00 02 – klasa o nazwie w elemencie nr 2
2. 01 00 04 – string UTF-8 o długości 4

   4D 61 69 6E - Main

PLIK .class

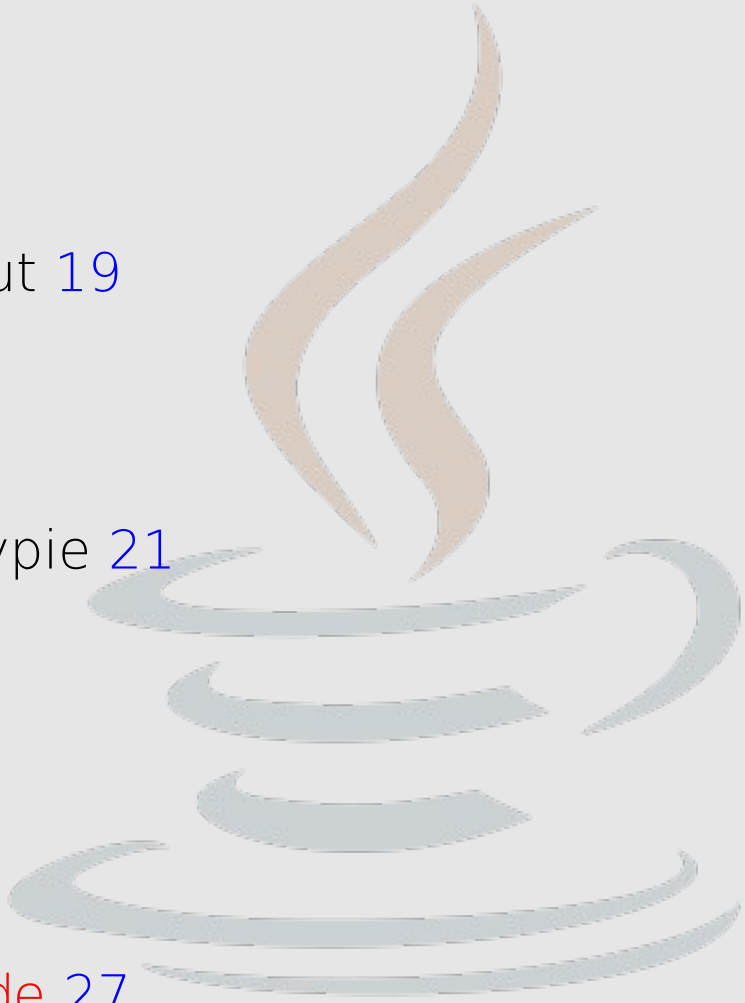
3. 07 00 04 – klasa o nazwie w elemencie nr 4
4. 01 00 10 – string UTF-8 o długości 16

   6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 - java/lang/Object
5. 01 00 06 <init>
6. 01 00 03 ()V
7. 01 00 04 Code
8. 0A 00 03 00 09 – klasa 3 posiada metodę 9
9. 0C 00 05 00 06 – metoda o nazwie 5 i sygnaturze 6 (brak argumentów (), zwraca void V)
10. 01 00 0F LineNumberTable
11. 01 00 12 LocalVariableTable
12. 01 00 04 this



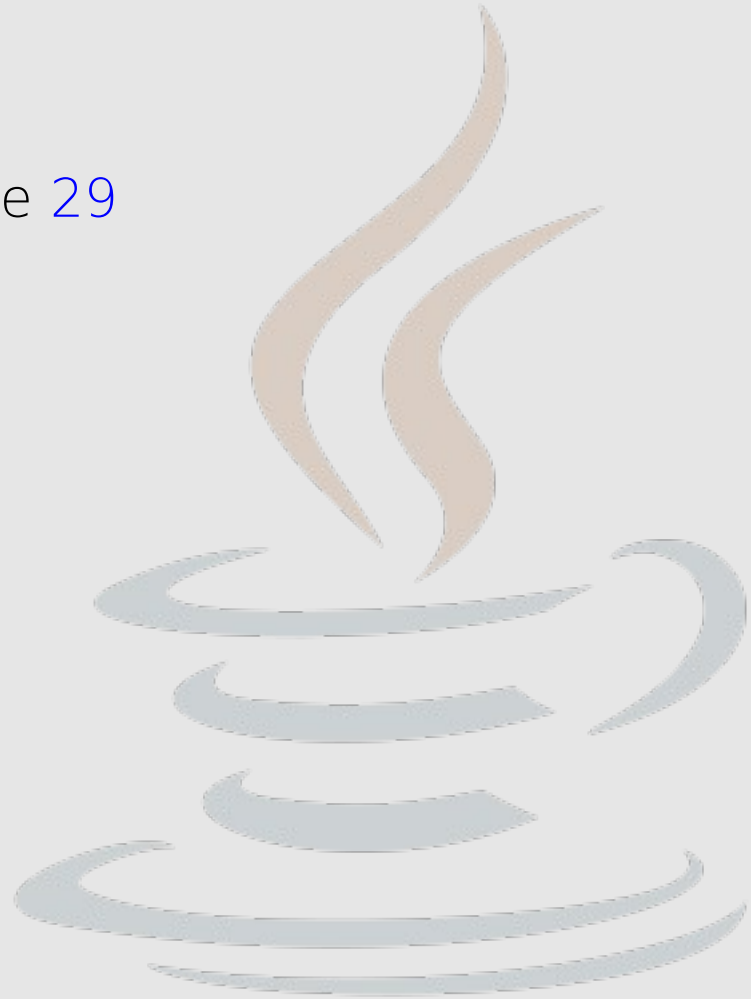
PLIK .class

13. 01 00 06 LMain;
14. 01 00 04 main
15. 01 00 16 ([Ljava/lang/String;)V
16. 09 00 11 00 13 – klasa 17 posiada atrybut 19
17. 07 00 12 – klasa o nazwie 18
18. 01 00 10 java/lang/System
19. 0C 00 14 00 15 – atrybut o nazwie 20 i typie 21
20. 01 00 03 out
21. 01 00 15 Ljava/io/PrintStream;
22. 08 00 17 – 23 element to stała tekstowa
23. 01 00 0C Hello world!
24. 0A 00 19 00 1B – klasa 25 posiada metodę 27



PLIK .class

25. 07 00 1A – klasa o nazwie 26
26. 01 00 13 java/io/PrintStream
27. 0C 00 1C 00 1D – metoda 28 o sygnaturze 29
28. 01 00 07 println
29. 01 00 15 (Ljava/lang /String;)V
30. 01 00 04 args
31. 01 00 13 [Ljava/lang/String;
32. 01 00 0A SourceFile
33. 01 00 09 Main.java



PLIK .class

00 21 - modyfikatory dostępu dla klasy: **ACC_PUBLIC (00 01) | ACC_SUPER (00 20)** – ze względu na kompatybilność ze starszymi JVM.

00 01 – numer elementu określający klasę definiowaną w tym pliku

00 03 – numer nadklasy

00 00 – liczba interfejsów

00 00 – liczba pól (atrybutów)

00 02 – liczba metod

PLIK .class – metody

```
method_info {
    u2             access_flags;
    u2             name_index;
    u2             descriptor_index;
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

00 01 – modyfikatory dostępu: **ACC_PUBLIC (00 01)**

00 05 – indeks elementu z nazwą metody (<init>)

00 06 – indeks elementu z sygnaturą metody

00 01 – liczba dodatkowych atrybutów

- **00 07** – indeks elementu z nazwą atrybutu (Code)
- **00 00 00 2F** – długość atrybutu
- **00 01 00 01** – rozmiar stosu I rozmiar tablicy zmiennych lokalnych

PLIK .class – <init>

00 00 00 05 – długość kodu

```
2A      : ALOAD_0 // zmienna lokalna o adresie 0
                  //  jest wstawiana na stos
B7 00 08: INVOKESPECIAL java/lang/Object.<init>()V;
                  // wywołuje metodę void Object.<init>();
B1      : RETURN  // zwraca typ void
```

00 00 – długość tablicy wyjątków

00 02 – liczba dodatkowych atrybutów

00 0A – tablica numerów linii

- 00 00 00 06 – długość atrybutu
- 00 01 – długość tabeli
  - 00 00 – indeks instrukcji w tabeli (Code)
  - 00 01 – nr lini w pliku źródłowym

PLIK .class – <init>

`00 0B` – tablica zmiennych lokalnych

`00 00 00 0C` – długość atrybutu

`00 01` – długość tabeli

`00 00` – początek zmiennej

`00 05` – długość zmiennej

`00 0C` – indeks nazwy zmiennej

`00 0D` – indeks nazwy typu zmiennej

`00 00` - indeks zmiennej w tabeli zmiennych lokalnych

PLIK .class – main

`00 09` – modyfikatory dostępu: **ACC_PUBLIC (00 01) | ACC_STATIC (00 08)**

`00 0E` – indeks elementu z nazwą metody (main)

`00 0F` – sygnatura ([Ljava/lang/String;)V

`00 01` – liczba dodatkowych atrybutów

- `00 07` – indeks elementu Code
- `00 00 00 37` – długość elementu
- `00 02` – rozmiar stosu
- `00 01` – rozmiar tablicy zmiennych lokalnych
- `00 00 00 09` – długość kodu

PLIK .class – main

```
B2 00 10 :GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
                          // inicjalizacja klasy/obiektu System.out
                          // i odłożenie go na stos
12 16    :LDC Hello world! // załadowanie na stos stałej tekstowej
B6 00 18 :INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang/String;)V
                          // wywołuje metodę System.out.println(String)
B1       :RETURN        // zwraca typ void
```

00 00 - długość tablicy wyjątków

00 02 – liczba dodatkowych atrybutów

00 0A – tablica numerów linii

```
  00 00 00 0A 00 02 00 00 00 04 00 08 00 05
```

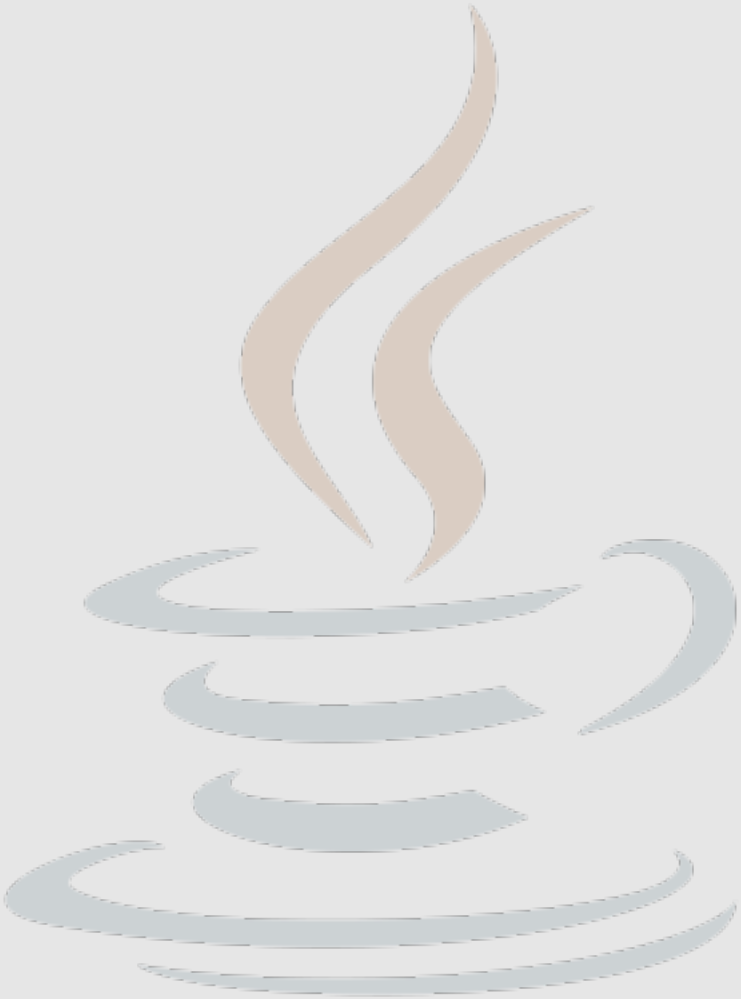
00 0B – tablica zmiennych lokalnych

```
  00 00 00 0C 00 01 00 00 00 09 00 1E 00 1F 00 00
```

PLIK .class – main

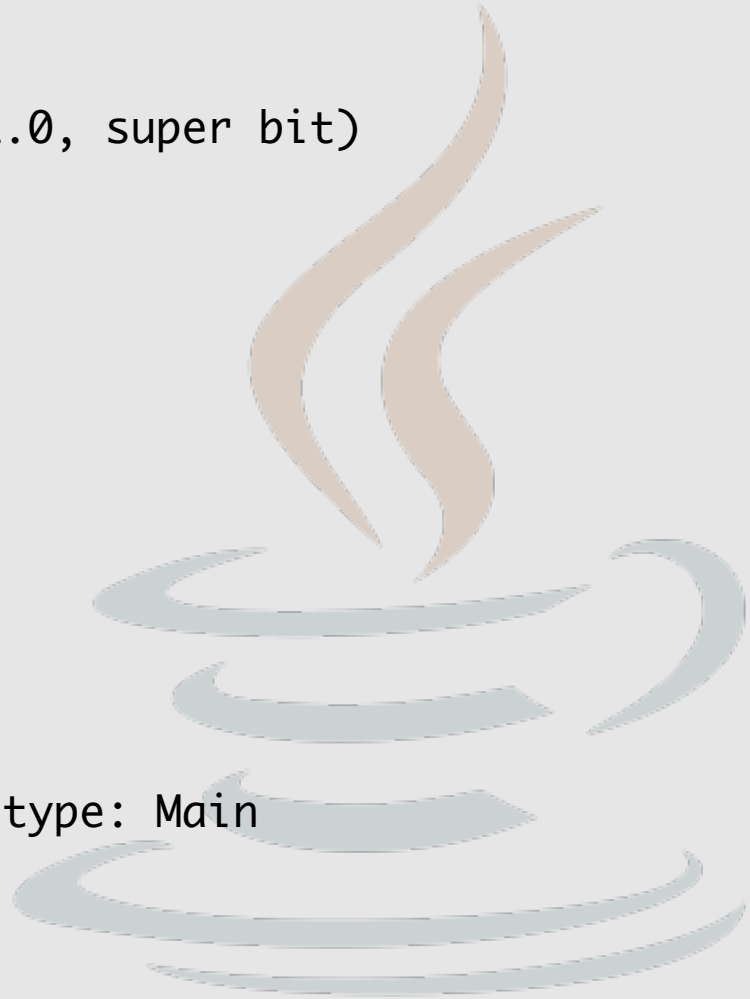
`00 01` – liczba atrybutów

- `00 20` – SourceFile
- `00 00 00 02` – długość atrybutu
- `00 21` - Main.java



PLIK .class – assembler

```
// Compiled from Main.java (version 1.7 : 51.0, super bit)
public class Main {

  // Method descriptor #6 ()V
  // Stack: 1, Locals: 1
  public Main();
    0  aload_0 [this]
    1  invokespecial java.lang.Object() [8]
    4  return
      Line numbers:
        [pc: 0, line: 1]
      Local variable table:
        [pc: 0, pc: 5] local: this index: 0 type: Main
```



PLIK .class – assembler

```
  // Method descriptor #15 ([Ljava/lang/String;)V
  // Stack: 2, Locals: 1
  public static void main(java.lang.String[] args);
    0  getstatic java.lang.System.out : java.io.PrintStream [16]
    3  ldc <String "Hello world!"> [22]
    5  invokevirtual java.io.PrintStream.println(java.lang.String) : void
[24]
    8  return
      Line numbers:
        [pc: 0, line: 4]
        [pc: 8, line: 5]
      Local variable table:
        [pc: 0, pc: 9] local: args index: 0 type: java.lang.String[]
}
```

ASM

ASM to biblioteka ułatwiająca manipulację bytecodem Javy.

Przykładowy kop generujący klasę Main wygląda następująco:

```
(java -cp  asm-4.1.jar:asm-util-4.1.jar:asm-commons-4.1.jar
 org.objectweb.asm.util.ASMifier  Main.class)
```

```
import org.objectweb.asm.*;

public class MainDump implements Opcodes {

    public static byte[] dump() throws Exception {

        ClassWriter cw = new ClassWriter(0);
        MethodVisitor mv;
        cw.visit(V1_7, ACC_PUBLIC + ACC_SUPER, "Main", null,
                                                "java/lang/Object", null);
```

ASM

```
mv = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
mv.visitCode();
mv.visitVarInsn(ALOAD, 0);
mv.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "<init>",
                                                        "()V");
mv.visitInsn(RETURN);
mv.visitMaxs(1, 1);
mv.visitEnd();

mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",
                              "([Ljava/lang/String;)V", null, null);
mv.visitCode();
mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
                                            "Ljava/io/PrintStream;");
mv.visitLdcInsn("Hello world!");
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
                             "println", "(Ljava/lang/String;)V");
mv.visitInsn(RETURN);
mv.visitMaxs(2, 1);
```

ASM

```
        mv.visitEnd();
        cw.visitEnd();

        return cw.toByteArray();
    }
}
```

Przykład użycia:

```
public class HelloWorldASM extends ClassLoader{
    public static void main(final String args[]) throws Exception {
        HelloWorldASM loader = new HelloWorldASM();
        byte[] code = MainDump.dump();
        Class cl = loader.defineClass("Main", code, 0, code.length);
        cl.getMethods()[0].invoke(null, new Object[] { null });
    }
}
```

DZIĘKUJĘ ZA UWAGĘ