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{
    ...
}
```

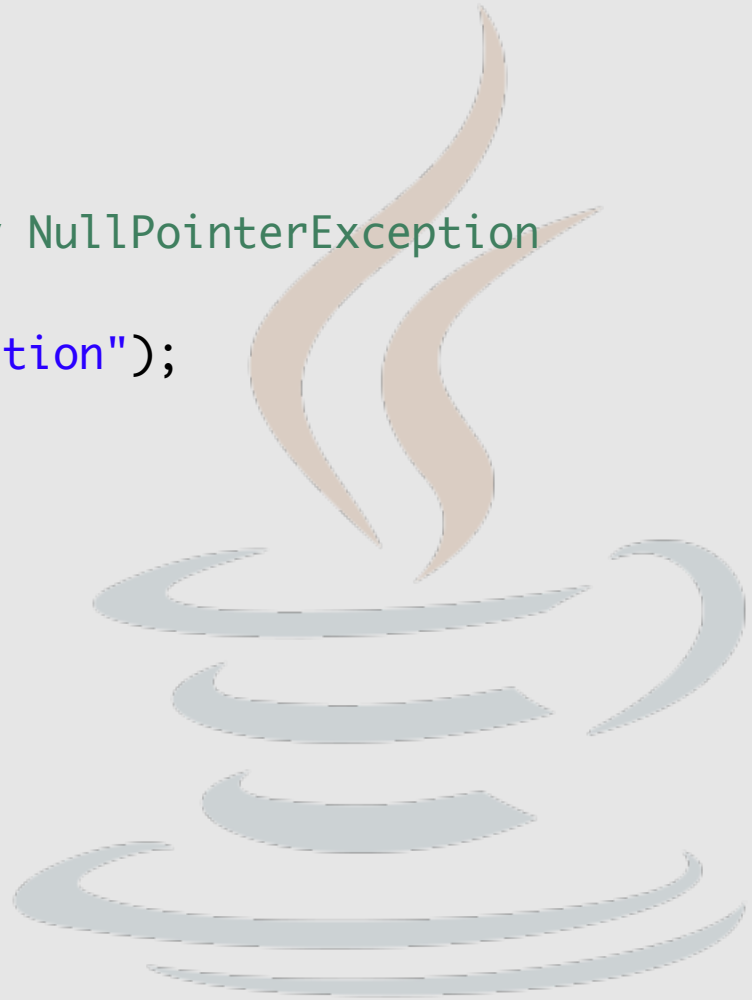
Od Javy 7 możliwe łączenie obsługi:

```
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");
}
```

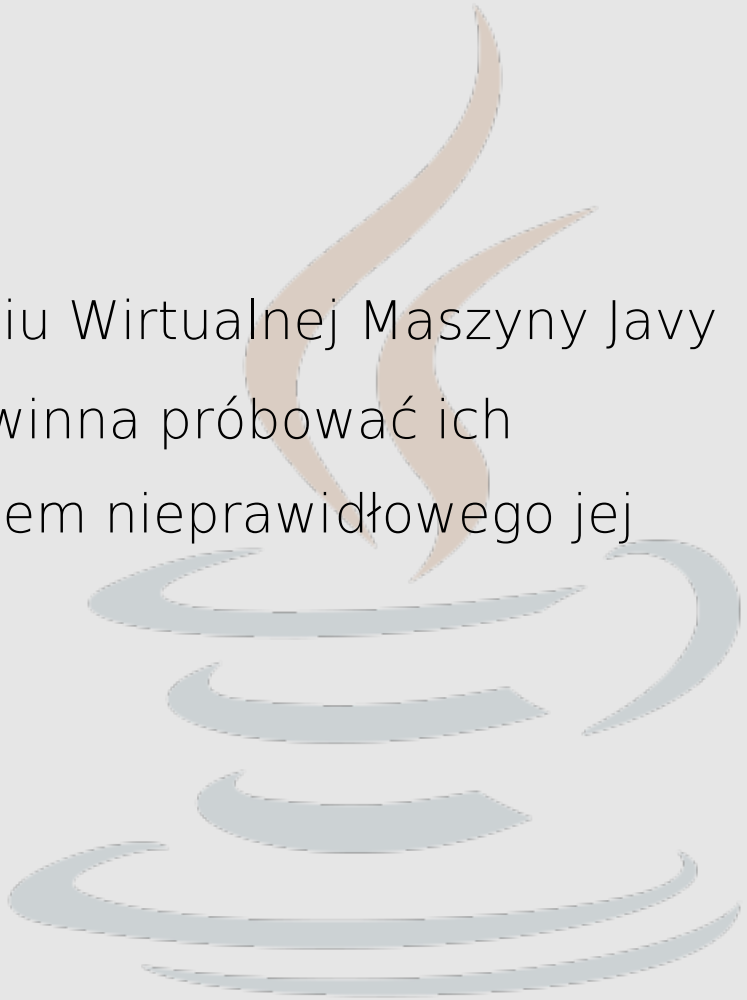


Co wypisze ten fragment kodu?

```
NullPointerException
Finally
```

BŁĘDY

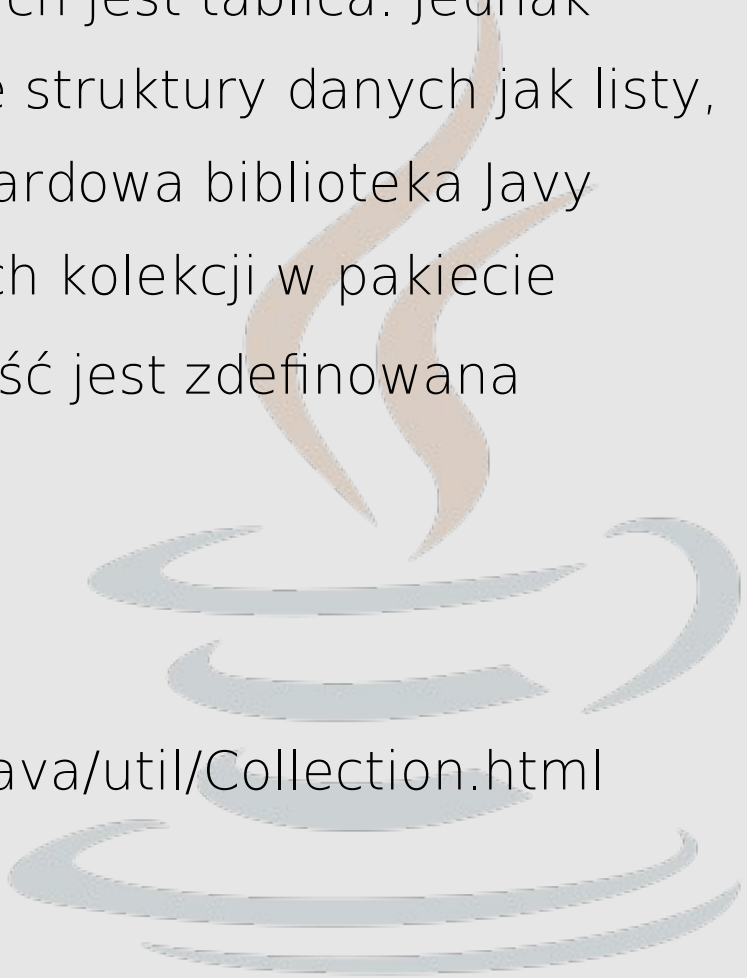
Błędy informują o nieprawidłowym działaniu Wirtualnej Maszyny Javy (np. `OutOfMemoryError`). Aplikacja nie powinna próbować ich obsługiwać, gdyż zwykle nie są one wynikiem nieprawidłowego jej działania.



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()
```

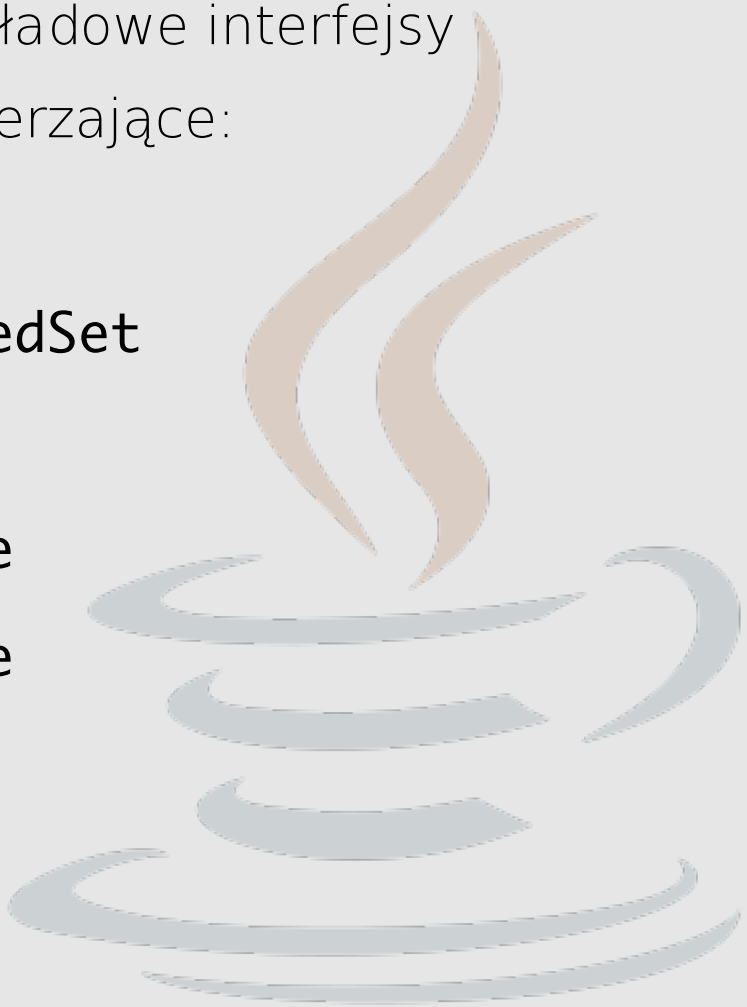
java.util.Collection

Przykładowe klasy rozszerzające:

- `ArrayList`
- `HashSet`
- `LinkedList`
- `Stack`
- `Vector`
- `PriorityQueue`
- `TreeSet`
- ...

Przykładowe interfejsy rozszerzające:

- `List`
- `SortedSet`
- `Set`
- `Queue`
- `Deque`
- ...



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{
}
```

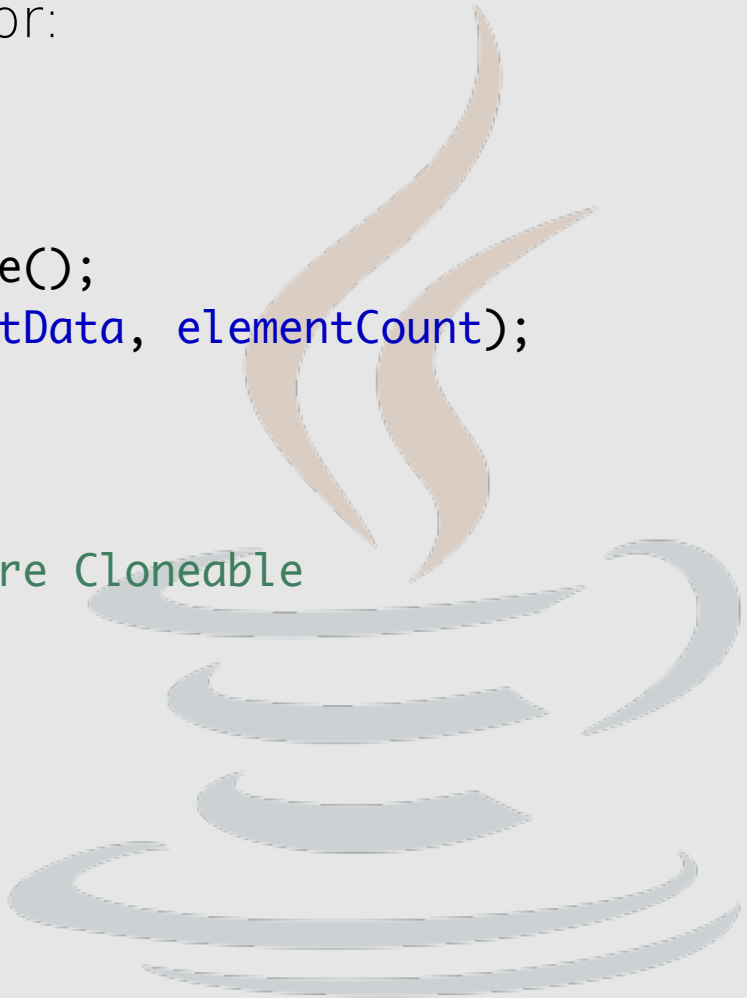
Implementacja interfejsu **Cloneable** informuje, że nasz obiekt wspiera klonowanie. Bazowa metoda clone jest zaimplementowana w klasie **Object**.

```
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.

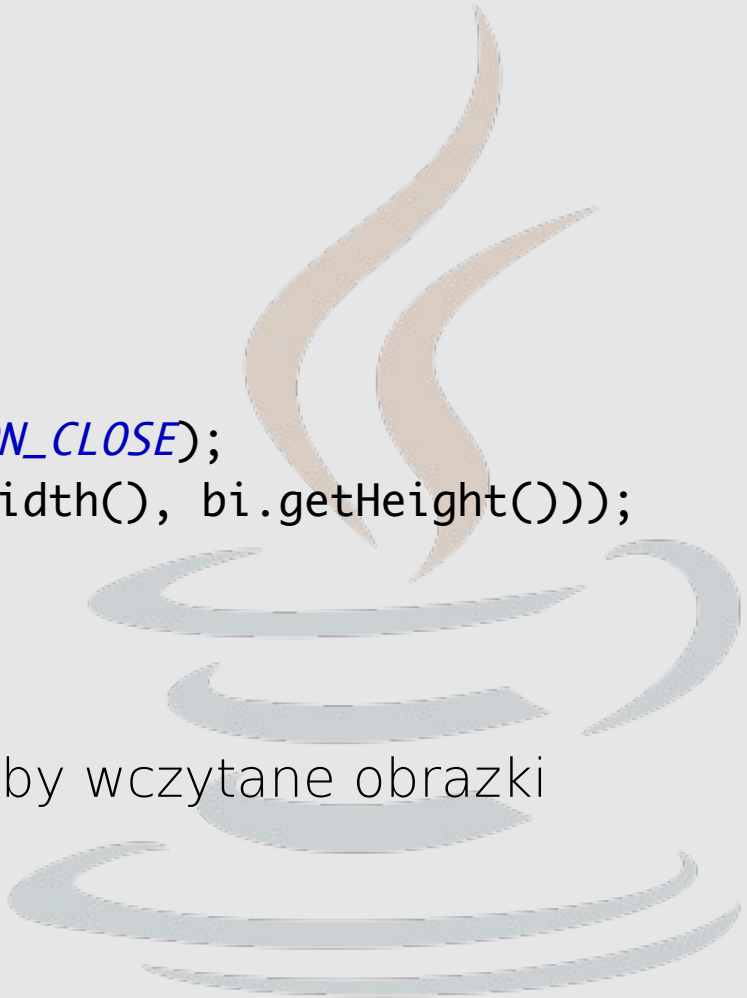
```
public class Hashtable<K,V> extends Dictionary<K,V>
             implements Map<K,V>, Cloneable, Serializable {
```

```
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);
```

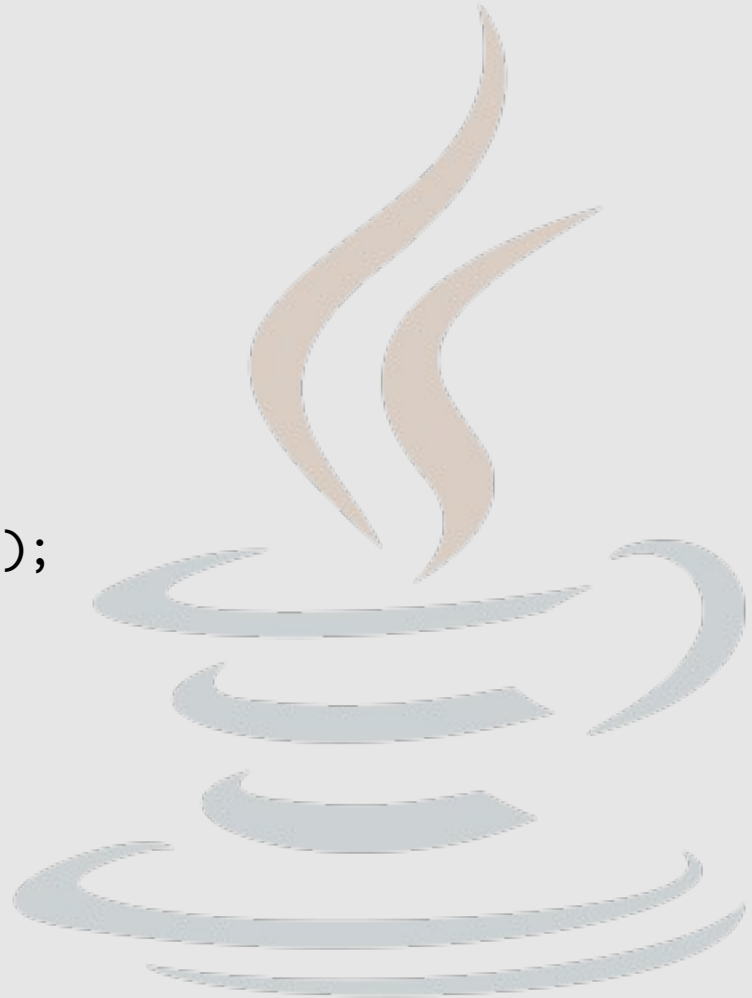
ĆWICZENIE: zmodyfikować program tak, aby wczytane obrazki zmieniały się.



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

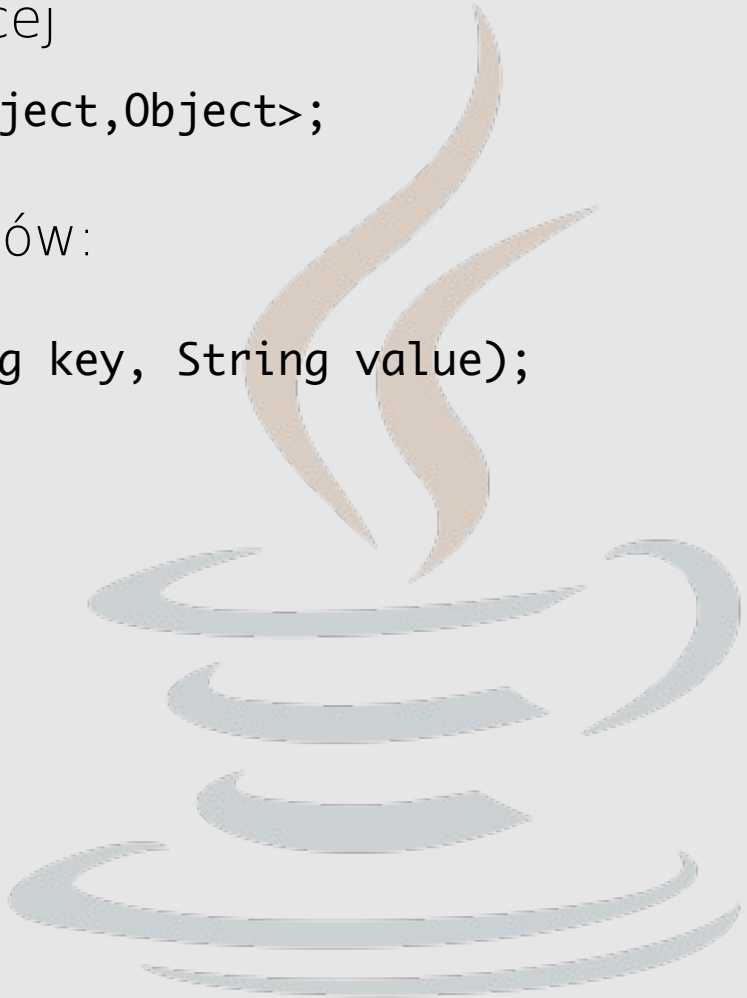
```
public class Properties extends Hashtable<Object,Object>;
```

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 plikami tekstowymi zapisanymi w określonym formacie:

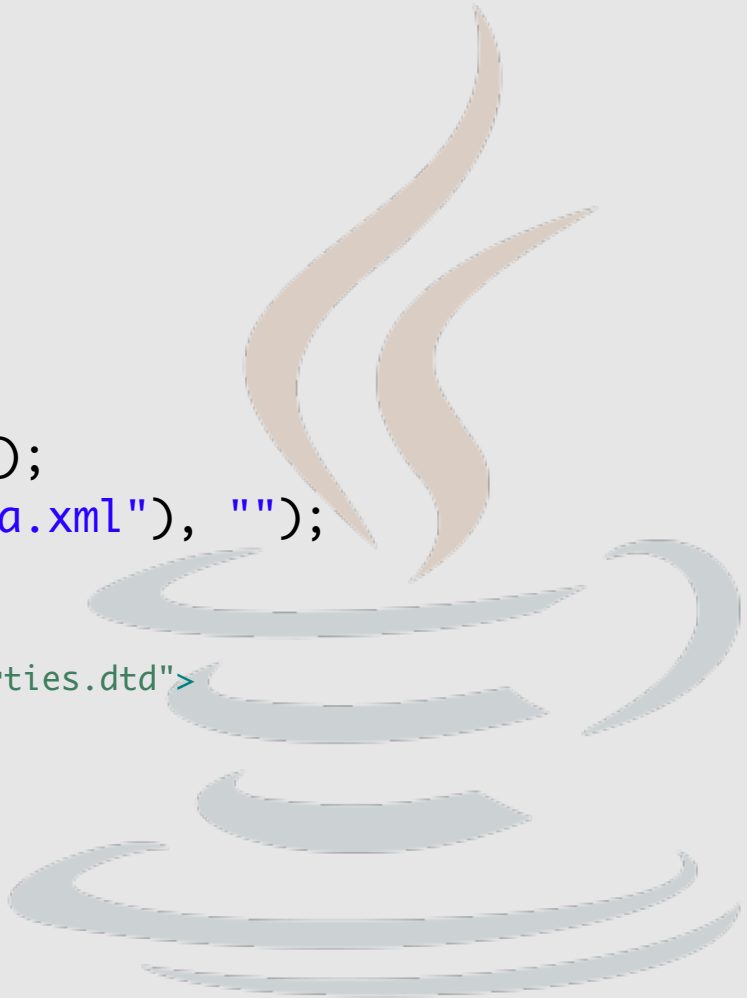
```
public synchronized void load(InputStream inStream) throws IOException

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

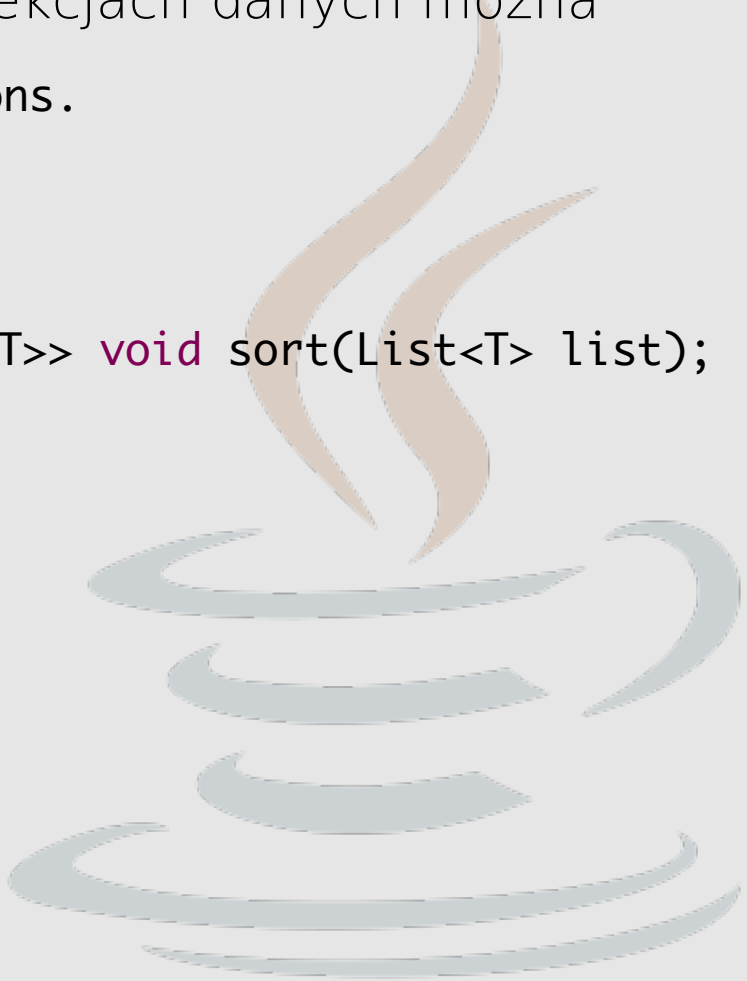
Podstawowe operacje na tablicach lub kolekcjach danych można wykonać za pomocą klas `Arrays` i `Collections.`

np sortowanie.

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

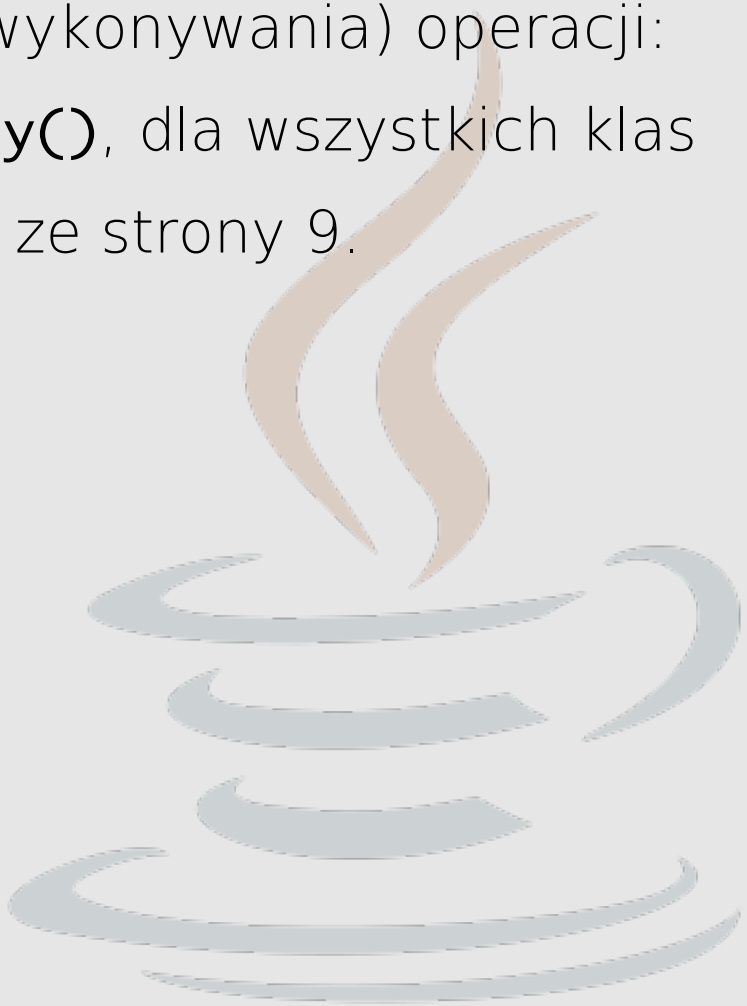
przy czym:

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



ĆWICZENIA

- Proszę porównać wydajność (czas wykonywania) operacji: `add()`, `remove()`, `contains()`, `toArray()`, dla wszystkich klas rozszerzających interfejs Collection ze strony 9.



DZIĘKUJĘ ZA UWAGĘ