Wprowadzenie do kolekcji

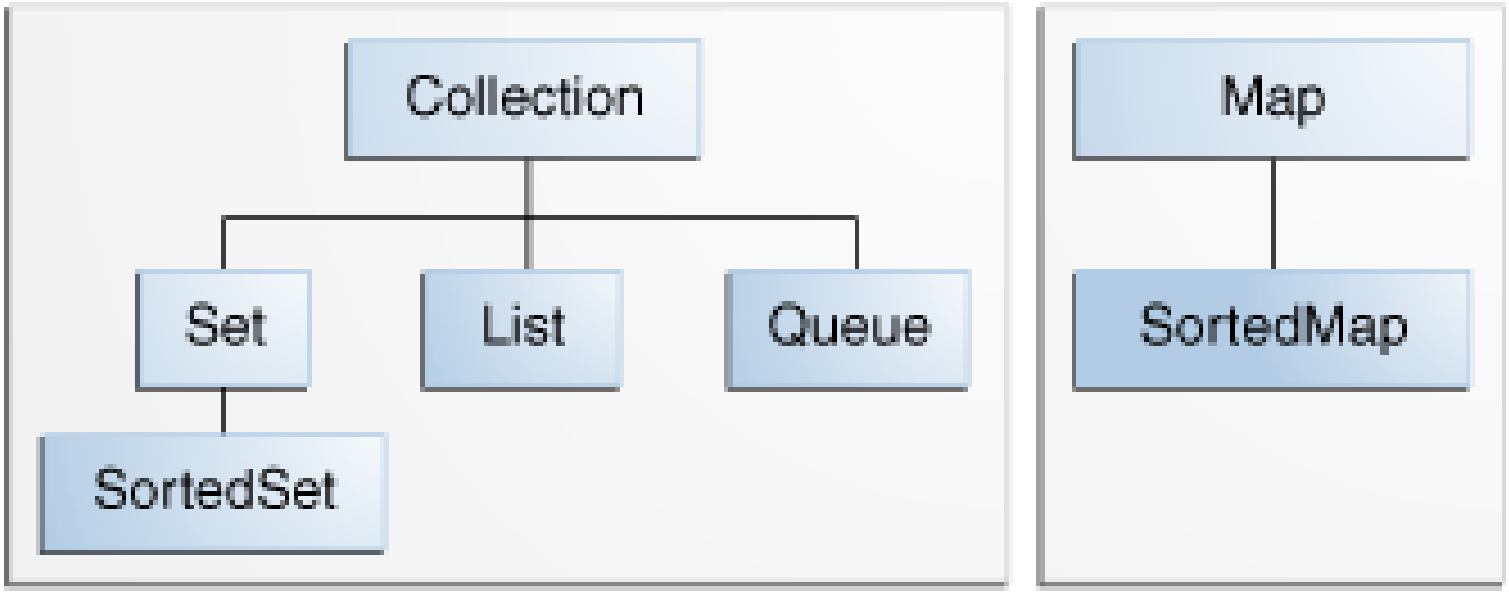
Dlaczego kolekcje?

- Przechowywanie obiektów w tablicach ma zasadniczą wadę: konieczność deklarowania rozmiaru tablicy w momencie jej tworzenia

(wyjątek ArrayIndexOutofBondException)

Kolekcje

- Java zapewnia zestaw interfejsów do efektywnego „przechowywania” obiektów



Przykład

Stwórzmy dwie proste klasy dziedziczące z innej:

```
class Zwierze {
        private static long counter;
        private final long id = counter++;
        public long id(){return id;}
}

class Kot extends Zwierze{
        public String miaucz(){return "Miauuu";}
}

class Pies extends Zwierze {
        public String szczekaj(){return „Hau!";}
}
```

Przykład Kolekcje1.java

```
ArrayList zwierzeta = new ArrayList();

for (int i =0; i<5; i ++) zwierzeta.add(new Kot());

for (int j =0; j<5; j ++) zwierzeta.add(new Pies());

for (int k = 0; k< zwierzeta.size(); k++){
        System.out.println( ((Zwierze)zwierzeta.get(k)).id());
}
// rzutowanie (Zwierze) pozwala na skorzystanie z metody id -
// jednak "efekt" rzutowania musi być ujęty w dodatkowy nawias:
// ( (Zwierze)zwierzeta.get(k) ).id()


// Rzutowanie na klasę Kot i próba skorzystania z metody miaucz
// Kompilator nie widzi bledu - wystąpi wyjątek ClassCastException:
for (int k = 0; k< zwierzeta.size(); k++){
        System.out.println( ((Kot)zwierzeta.get(k)).miaucz());
}
```

Przykład Kolekcje2.java

```
// Dla kolekcji można zdefiniować typ przechowywanych obiektów:
ArrayList<Zwierze> zwierzeta = new ArrayList<Zwierze>();

for (int i =0; i<5; i ++) zwierzeta.add(new Kot());
for (int j =0; j<5; j ++) zwierzeta.add(new Pies());

for (int k = 0; k< zwierzeta.size(); k++){
        System.out.println( zwierzeta.get(k).id());
}
//  w tym wypadku dostep do metody id nie wymaga rzutowania

//powyższą pętle można zapisać "bardziej elegancko":
for(Zwierze z : zwierzeta){
        System.out.println( z.id() );
}

// ale proba rzutowania na "nadklase" dalej moze spowodowac wyjatek:
for (Zwierze z : zwierzeta){
        System.out.println( ( (Kot)z ).miaucz());
}
```

Pętla **for each - ForEach.java**

służy do iteracji po kolejnych elementach tablicy lub kolekcji,
ogólna składnia:

```
for ({deklaracja zmiennej pętli} : {kolekcja lub tablica}) {
  {ciało pętli}
}
```

Np.:

```
String[] teksty = {"jeden", "dwa", "trzy" };
for (String s : teksty) System.out.println(s);

ArrayList<Kot> koty = new ArrayList<Kot>();
koty.add(new Kot() ); koty.add(new Kot() );
koty.add(new Kot() ); koty.add(new Kot() );

for(Kot k : koty) System.out.println("Kot id: " + k.id() );
```

Przykład Kolekcje3.java

```
ArrayList<Kot> koty = new ArrayList<Kot>();
// czasem deklarując zmienne stosuje sie bardziej ogólne
interfejsy
// co może m.in. ułatawiać ewentualne zmiany implementacji, np.
List<Pies> psy = new ArrayList<Pies>();
Collection<Zwierze> zwierzeta = new ArrayList<Zwierze>();

for (int i =0; i<5; i ++) {
      koty.add(new Kot());
      psy.add(new Pies());
}
```

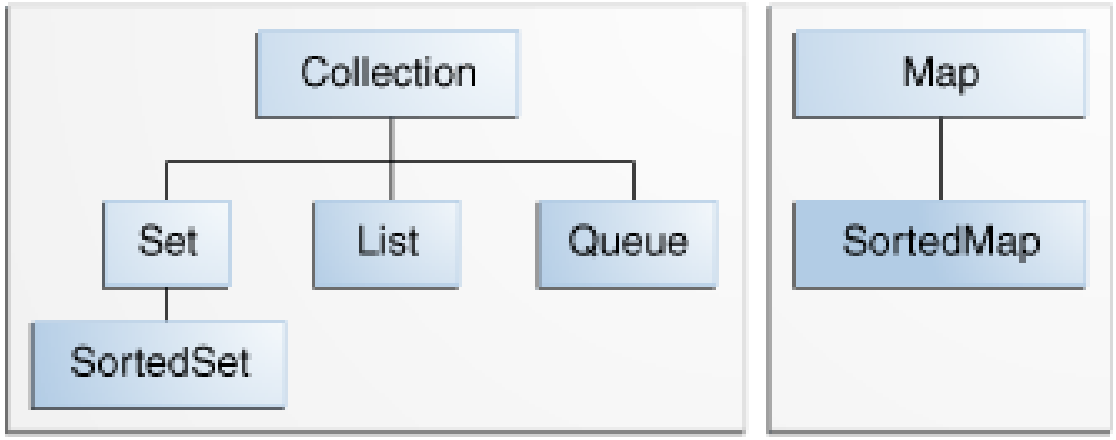
Przykład Kolekcje3.java

```
//kolekcje można łączyć lub dodawac do innej kolekcji, np:
  zwierzeta.addAll(koty);
  zwierzeta.addAll(psy);

for (Zwierze z : zwierzeta){

        if ( z.getClass() == Kot.class)
                System.out.println("Kot nr " + z.id() + ": " +
                        ((Kot)z).miaucz());

        if ( z.getClass() ==
        pl.edu.pw.fizyka.pojava.wyklad5.kolekcje.Pies.class )
                System.out.println("Pies nr " + z.id() + ": " +
                        ((Pies)z).szczekaj());
}
```

Konwersja tablic na kolekcję i odwrotnie – Kolekcje4.java

```
Integer[] tablicaLiczb = {1,2,3,4,5};

List<Integer> listaLiczb = new ArrayList<Integer>();
//List<Integer> listaLiczb2 = Arrays.asList(6,7,8);

listaLiczb.addAll( Arrays.asList(tablicaLiczb));

// Całą kolekcje można łatwo wypisać:
System.out.println(listaLiczb);

Integer[] innaTablica =
     listaLiczb.toArray(new Integer[listaLiczb.size()]);

//Dla tablic takie wyswietlanie nie dziala:
System.out.println(innaTablica);

for (int b : innaTablica) System.out.println(b);
```

Podstawowe interfejsy



- Set – nie może być duplikatów
- List – elementy w określonej kolejności
- Queue – uporządkowane zgodnie z dyscypliną kolejki
- Map – grupa par obiektów klucz-wartość

Metody „wspólne” – interfejs Collection

Modifier and Type	Method and Description
boolean	**add(E e)** Ensures that this collection contains the specified element (optional operation).
boolean	**addAll(Collection<? extends E> c)** Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear()** Removes all of the elements from this collection (optional operation).
boolean	**contains(Object o)** Returns `true` if this collection contains the specified element.
boolean	**containsAll(Collection<?> c)** Returns `true` if this collection contains all of the elements in the specified collection.
boolean	**equals(Object o)** Compares the specified object with this collection for equality.

Metody „wspólne” – interfejs Collection

int	**hashCode()** Returns the hash code value for this collection.
boolean	**isEmpty()** Returns `true` if this collection contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this collection.
boolean	**remove(Object o)** Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll(Collection<?> c)** Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll(Collection<?> c)** Retains only the elements in this collection that are contained in the specified collection (optional operation).

Metody „wspólne” – interfejs Collection

int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Więcej na:
http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html
Warto zwrócić uwagę, że część metod jest traktowanych jako operację „opcjonalne” – nie wszystkie są implementowane w interfejsach/klasach pochodnych

Interfejs Collection – interfejsy pochodne i implementujące klasy

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

Klasa Collections

- http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

zawiera szereg statycznych metod pozwalających na operacje na kolekcjach, np.: wyszukiwanie, sortowanie, dodawanie, odwracanie, wyszukiwanie maksymalnej/minimalnej wartości…

Posiada także metody statyczne, pozwalające zmieniać własności kolekcji, np. blokować metody modyfikujące kolekcję:

Collection immutableCollection(Collection kol)

List immutableList(List lista)

Set immutableSet(Set zbior)

Map immutableMap(Map mapa)

List – wybrane metody

dostęp pozycyjny do elementów

- Object get(int indeks)
- Object set(int indeks)
- Object add(int indeks)
- Object remove(int indeks)

wyszukiwanie

- int indexOf(Object obiekt)
- int lastIndexOf(Object obiekt)

rozszerzona iteracja

- ListIterator listIterator()

widok przedziałowy

- List subList(int poczatek, int koniec)

LinkedListDemo.java

Queue

Metody niezależne od interfejsu Collection:

dodawanie elementu

- boolean offer(Object obiekt)

usuwanie elementu

- Object remove()

- Object poll()

inspekcja

- Object element()

- Object peek()

QueueDemo.java

```
Queue<Integer> queue = new LinkedList<Integer>();

for (int i = 0 ; i< 100; i+=8)
 queue.offer(i);

System.out.println(queue);

while (!queue.isEmpty())
{
 System.out.println(queue.poll());
// System.out.println(queue.peek()); // pobiera wartosc bez
usuwania z kolejki
}
```

QueueDemo.java

```
List<Integer> listaLiczb = Arrays.asList(10,6,7,8,0);

PriorityQueue<Integer> priorQueue = new
                PriorityQueue<Integer>(listaLiczb);
System.out.println(priorQueue);

PriorityQueue<Integer> priorQueue2 =
        new PriorityQueue<Integer>(listaLiczb.size(),
                                   Collections.reverseOrder());
priorQueue2.addAll(listaLiczb);
System.out.println(priorQueue2);
```

Set

- Reprezentacja zbioru matematycznego
  - brak relacji porządku wewnątrz zbioru
  - brak duplikatów
- Brak nowych metod w stosunku do *Collection*
- najważniejsze implementacje: *HashSet*, *TreeSet*

SetDemo.java

```
Random r = new Random();

Set <Integer> intSet = new HashSet<Integer>();

for (int i = 1; i<10000; i++) intSet.add(r.nextInt(20));

System.out.println(intSet);

// prosty generator "lotto":
Set <Integer> lotto = new HashSet<Integer>();
    for (int i = 0; i<6; i++)
    if (! lotto.add(r.nextInt(48)+1)) i--;

    // lotto.add(r.nextInt(48)) zwraca false jesli proba dodania
liczby nieudana - liczba juz wczesniej wyloswoana....

System.out.println("Podaje szczesliwe liczby: " + lotto);
```

Map

- Kolekcja jednoznacznych odwzorowań *klucz-wartość*
- Klucze, wartości i pary *klucz-wartość* dostępne jako obiekty *Collection*
- Dwa obowiązkowe konstruktory (analogicznie do kolekcji)

Najważniejsze implementacje:
HashMap, *TreeMap*, *SortedMap*

MapDemo.java

```
Random r = new Random();

Map<String, Integer> mapa = new HashMap<String, Integer>();

for (int i = 0; i<10; i++ )
{
        int los = r.nextInt(100);
        String s = "\nLiczba nr " + i + ": ";
// s = "klucz stały"; // tylko jeden element bedzie dodany
       mapa.put(s, los);
}

System.out.println(mapa);
```

MapDemo.java

```
for (Map.Entry<String, Integer> e : mapa.entrySet()){
        String s = e.getKey();
        Integer i = e.getValue();
        System.out.println("Klucz: " + s + " Wartość: " + i );
}

mapa.remove("\nLiczba nr 5: ");
System.out.println(mapa);
```

Iteratory

Iteratory pozwalają przesuwać się po kontenerze. Zwykłe iteratory są wyposażone w następujące metody:
next() – zwraca następny obiekt w kolekcji,

hasNext() – zwraca czy jest jakiś następny element,
remove() – wymazuje ostatnio zwrócony obiekt.

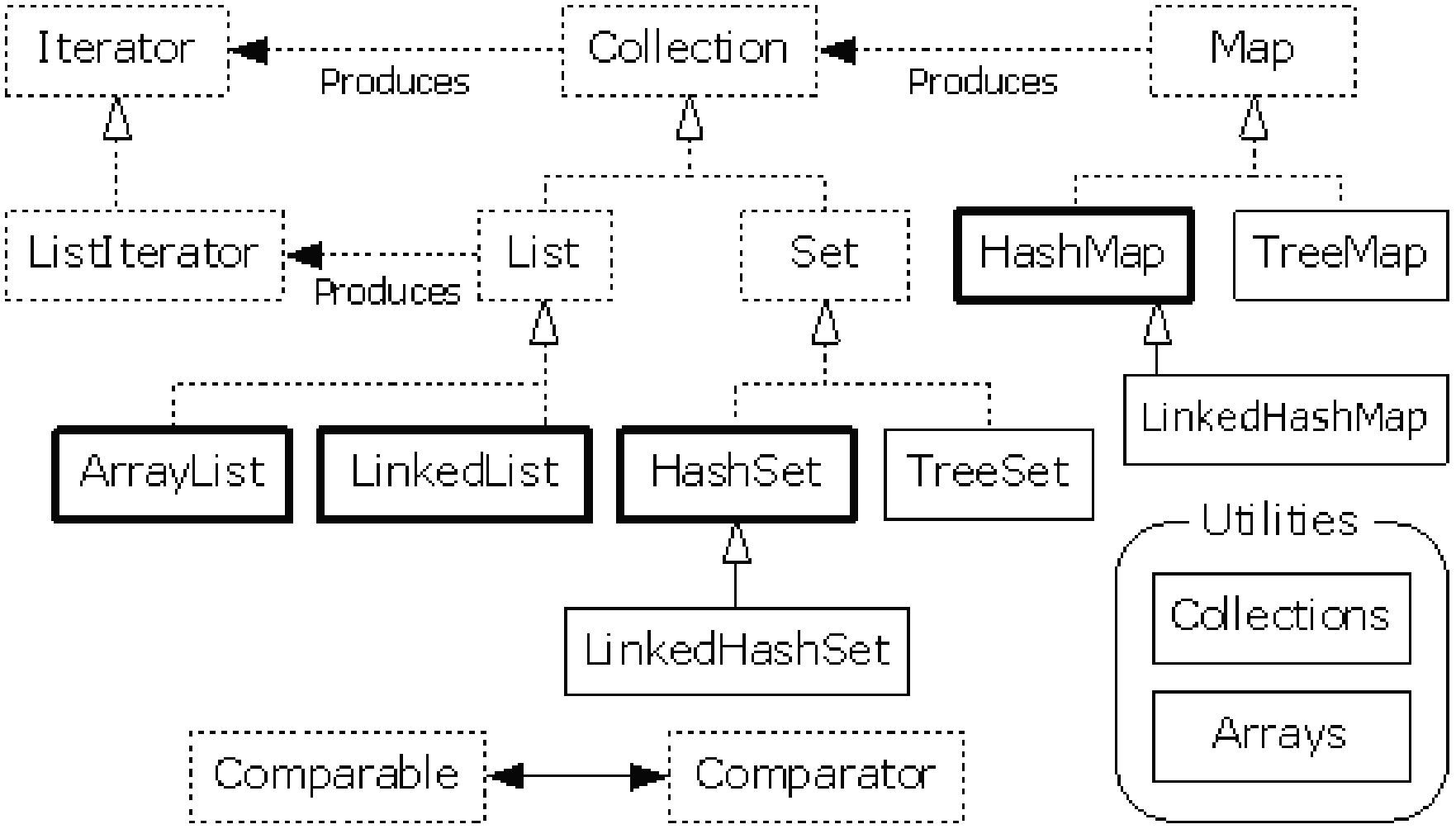
Dla kontenerów typu List istnieje nieco bardziej rozbudowany interator zwany ListIterator (m.in. jest „dwukierunkowy” – posiada metodę **previous()**…)

IteratorDemo.java

```
List<Integer> listaLiczb = Arrays.asList(1,2,3,4,5,6,7,8,9,0);

// Rownowazne takiemu zapisowi:
// Integer[] tablicaLiczb = {1,2,3,4,5,6,7,8,9,0};
// List<Integer> listaLiczb = new ArrayList<Integer>();
// listaLiczb.addAll( Arrays.asList(tablicaLiczb));

Iterator i = listaLiczb.iterator();

while (i.hasNext()){
       System.out.println(i.next());
        //i.remove();
}

System.out.println(listaLiczb);
```

IteratorDemo.java

```
ListIterator i2 = listaLiczb.listIterator();

while (i2.hasNext()){
	System.out.println(i2.next());
}

while (i2.hasPrevious()){
	System.out.println("Obiekt o indeksie " +
		i2.previousIndex() + " : " + i2.previous());
}

//możliwe przegladanie listy „w obie strony”
```


Iterator
Produces
Collection
Produces
Map
ListIterator
Produces
List
Set
HashMap
TreeMap
ArrayList
LinkedList
HashSet
TreeSet
LinkedHashMap
LinkedHashSet
Utilities
Collections
Arrays
Comparable
Comparator

Rysowanie z poprzedniego wykładu z wykorzystaniem kolekcji

```
List<Prostokat> prostakaty = new ArrayList<Prostokat>();
int liczbaelementow =0;
Prostokat p = null;

  public void mousePressed(MouseEvent e) {

        p = new Prostokat();
        p.setX( e.getX() );
        p.setY( e.getY() );
        p.setWidth(50);
        p.setHeight(50);
        p.setColor(new Color((float)Math.random(),
                  (float)Math.random(),(float)Math.random()));

        prostakaty.add( p);

         repaint();

   }
```

Rysowanie z poprzedniego wykładu z wykorzystaniem kolekcji

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);

    for(Prostokat pr : prostakaty){
              pr.paint(g);
    }

}
```

LookAndFeelChooser.java
LookAndFeelChooserCollection.java

```
static final Map<String, String> LF_MAP = createLfMap();


 static Map<String, String> createLfMap(){
     Map<String, String> map = new HashMap<String, String>();
     for (UIManager.LookAndFeelInfo info :
                     UIManager.getInstalledLookAndFeels()){
         map.put(info.getName(), info.getClassName());
         System.out.println(info.getClassName());
     }
     return Collections.unmodifiableMap(map);
 }


 final JComboBox chooseLFComboBox = new JComboBox();
        for (String name : LF_MAP.keySet()){
            chooseLFComboBox.addItem(name);
        }
```

Strumienie – przykładowe programy (więcej na następnym wykladzie…)

- BuforZnakowy.java
- Konsola.java
- Pliki_zapis.java
- Pliki_zapis2.java
- Pliki_odczyt.java
- Pliki_odczyt2.java
- Pliki_odczyt3.java
- Pliki_RAF.java
- Pliki_wybor.java
- LoadObjects.java
- SaveObjects.java