Programowanie obiektowe

Wykład 7: Kolekcje

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.4
4 marca 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny

Ograniczenia klasy Array

- Obiekt klasy `Array` ma z góry określony rozmiar. Jego zwiększanie jest uciążliwe

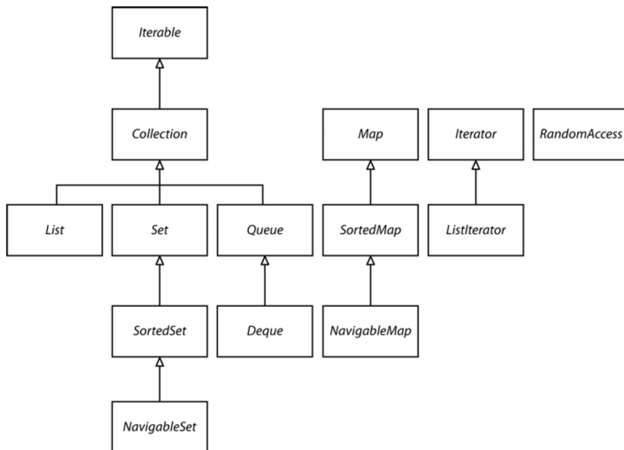
```
1       Integer[] array = {1,2,3,4,5};
2       Integer[] tmp = new Integer[6];
3       for(int i=0; i<array.length;i++)
4           tmp[i]=array[i];
5       tmp[5]=6;
6       array=tmp;
```

- Klasa `Array` nie ma kontroli typów elementów

```
1   String temp[] =  new String[2];
2   temp[0] = new Integer(12);
```

- Problemy te rozwiązuje wprowadzenie interfejsu `Collection`

Hierarchia Java Collections Framework



Rysunek 1: Hierarchia interfejsów kolekcji [Horstmann, 2016]

Podstawowe kolekcje

List Uporządkowana lista elementów,

Set Zbiór unikalnych elementów

Queue Struktura z wyróżnionym początkiem i końcem

Map Słownik par klucz wartość

Metody podstawowe

- Liczność kolekcji.
  - `int size();`
- Czy kolekcja jest pusta?.
  - `boolean isEmpty();`
- Czy kolekcja zawiera element?.
  - `boolean contains(Object element);`
- Dodanie elementu do kolekcji.
  - `boolean add(E element);`
- Usunięcie elementu z kolekcji.
  - `boolean remove(Object element);`

Kolekcje jako zbiory

- Zawieranie.
  - `boolean containsAll(Collection<?> c);`
- Unia
  - `boolean addAll(Collection<? extends E> c);`
- Różnica.
  - `boolean removeAll(Collection<?> c);`
- Iloczyn
  - `boolean retainAll(Collection<?> c);`
- Zbiór pusty.
  - `void clear();`

Przeglądanie kolekcji

- Interfejs kolekcji jest wspólny dla bardzo różnych struktur danych.
- Co więcej struktura kolekcji nie musi być zwarta jak w tablicy.
- Dostęp do elementów kolekcji określony jest więc abstrakcyjnie poprzez interfejs `Iterator`

Iterator

```
1  public interface Iterator<E>{
2      E next();
3      boolean hasNext();
4      void remove();
5      default void forEachRemaining(Consumer<? super E>
           action);
6  }
```

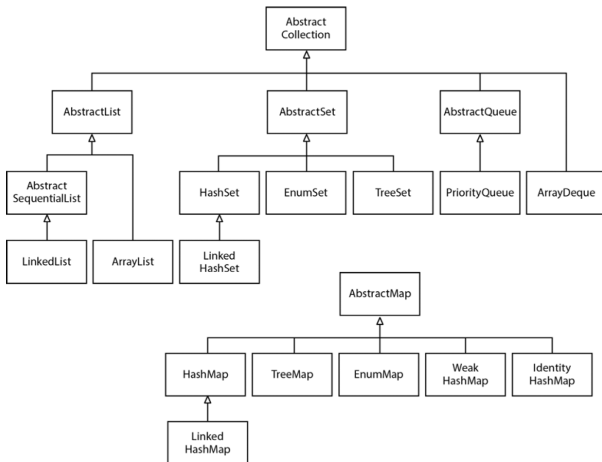
Używanie iteratora

- Iterator pobieramy z kolekcji metodą `iterator()`
- Iterator zwraca następny element kolekcji.
  - `E next();`
- Jednakże należy sprawdzić, czy taki element istnieje.
  - `boolean hasNext();`
- Iterator pozwala na usunięcie wskazywanego elementu
  - `void remove();`
- Iterator automatycznie integruje się z pętlą typu `for each`, a także pozwala na automatyczne wykonanie akcji na pozostałych elementach kolekcji
  - `default void forEachRemaining(Consumer<? super E> action); //Wiecej w przyszlym semestrze`

Konwersja do tablicy

- Kolekcje da się rzutować do klasy `Array`.
  - `Object[] toArray();`
- Jest to czasami niezbędne aby uzyskać odpowiedni format danych wejściowych
  - Sortowanie kolekcji odbywa się po ich konwersji do tablicy
- By uzyskać inny typ danych należy dokonać konwersji generycznej
  - `<T> T[] toArray(T[] a)`
- Konwersja odczytuje typ danych z tablicy a będącej parametrem metody.
- Wynik jest tablicą obiektów odczytanego typu o rozmiarze kolekcji.
- Jeżeli rozmiar tablicy będącej argumentem metody na to pozwala to zostanie ona także wypełniona elementami kolekcji

Implementacje interfejsów kolekcji



Rysunek 2: Klasy implementujące `Collection` [Horstmann, 2016]

Interfejs listy

- Listy implementują interfejs List, który umożliwia:
  - Dostęp do elementów i ich modyfikacje

```
1  E get (int index);
2  E set(int index, E element);
3  boolean add(E element);
4  void add(i index, E element);
5  E remove(int index);
6  boolean addAll(int index, Collection <? Extends E> c);
```

  - Wyszukiwanie elementów

```
1  int indexOf(Object o);
2  int lastIndexOf(Object o);
```

  - Uzyskanie zakresu z listy

```
1  List<E> subList(int from, int to);
```

Interfejs iteratora listy

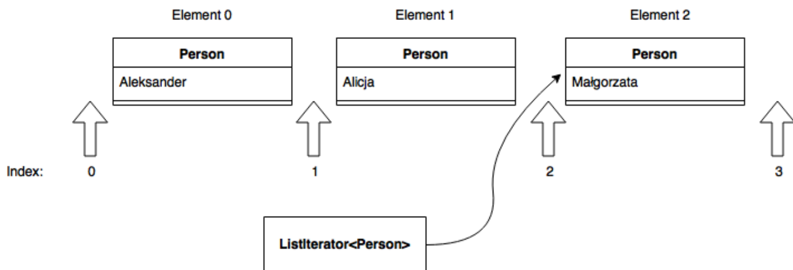
- Listy udostępniają dwukierunkowy iterator ListIterator

```
1  ListIterator<E> listIterator();
2  ListIterator<E> listIterator(int index);
```

- Funkcje iteratora

```
1  //Sprawdzenie obecnosci elementu
2  boolean hasNext();
3  boolean hasPrevious();
4  //Pobieranie elementu
5  E next();
6  E previous();
7  //Odczyt indeksu
8  int nextIndex();
9  int previousIndex();
10 //Akcja na wskazywanym elemencie
11 void remove();
12 void set(E e);
13 void add(E e);
```

Działanie iteratora



Rysunek 3: Lista i iterator wskazujący na element

- Iterator może się znajdować przed, pomiędzy i za elementami listy.
- Metody `hasNext()` i `hasPrevious()` sygnalizują to poprzez zwracane wartości

Iterowanie

- Iterator można wykorzystywać w sposób jawny.

Sortowanie kart

```
1   for(Iterator i = suits.iterator(); i.hasNext();){
2       Suit suit = (Suit) i.next();
3       for(Iterator j = ranks.iterator(); j.hasNext();){
4       Rank rank = (Rank) j.next();
5        sortedDeck.add(new Card(suit,rank));
6       }
7   }
```

- Może to jednak prowadzić do pewnych problemów.
- Bardziej eleganckie i czytelne jest stosowanie iteratora w sposób ukryty.

Sortowanie kart z ukrytym iteratorem

```
1   for(Suit suit:suits){
2       for(Rank rank:ranks){
3           sortedDeck.add(new Card(suit,rank));
4       }
5   }
```

Implementacje listy

- Lista jest implementowana przez dwie klasy.
  - `Vector`
  - `ArrayList`
- Implementują interfejs listy.
- Zapewniają bezpieczne zwiększanie rozmiaru kolekcji.

Zwiększanie rozmiaru

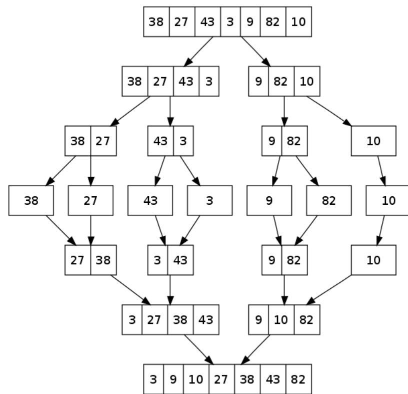
```
1   ArrayList<String> listaImion = new ArrayList<>(2); //Lista
        deklarowana jako dwuelementowa
2   listaImion.add("Aleksander");
3   listaImion.add("Alicja");
4   listaImion.add("Malgorzata"); //automatyczny wzrost rozmiaru}
```

Różnice między implementacjami

- `Vector`
  - W przypadku braku miejsca podwaja swój rozmiar.
  - Jest synchronizowany, czyli bezpiecznie można go używać przy programowaniu wielowątkowym.
- `ArrayList`
  - W przypadku braku miejsca zwiększa swój rozmiar o połowę.
  - Nie jest synchronizowana, więc działa szybciej w aplikacjach jednowątkowych.

Sortowanie

- Sortowanie kolekcji to porządkowane jej elementów względem zadanego klucza.
- Java oferuje sortowanie *stabilne*, czyli dwa elementy o tej samej wartości nie zmienią kolejności po sortowaniu.
- Używany jest zmodyfikowany algorytm sortowania przez scalanie.



Rysunek 4: Działanie algorytmu *mergesort* [Wikipedia, 2018]

Sortowanie kolekcji

- Kolekcje możemy sortować korzystając z metody statycznej klasy `Collections`
  - `void sort(Collection c);`
- Sortowanie wymaga, aby obiekty w kolekcji implementowały interfejs `Comparable`
- Typy wbudowane jak `String`, `Integer`, mają już zaimplementowane działanie interfejsu `Comparable`.

Interfejs Comparable<T>

- Interfejs jest zdefiniowany dla określonego typu `T`, czyli możemy porównywać tylko obiekty danego typu
- Zawiera tylko jedną metodę `int compareTo(T o)`
    - Zwraca 0 jeżeli obiekty są równe
    - Zwraca wartość ujemną, jeżeli dany obiekt jest mniejszy/wcześniejszy niż o
    - Zwraca wartość dodatnią, jeżeli dany obiekt jest większy/późniejszy niż o

Kolekcja monet

- Mamy kolekcję monet opisanaych przez klasę `Coin`
  - Nazwa
  - Wartość
  - Rok wybicia

Kolekcja

1. Double eagle (1933) - $7.4 million.
2. Liberty Head nickel (1913) - $3.7 million.
3. Flowing hair silver dollar (1794) - $10.0 million.
4. Edward III florin (1343) - $6.8 million.
5. Silver dollar (1804) - $3.8 million.
6. Brasher Doubloon (1787) - $7.4 million.
7. Saint-Gaudens double eagle (1907) - $7.4 million.

Sortowanie monet

- Niech klasa Coin implementuje interfejs Comparable, by porównywać wartość monet

```
1 @Override
2 public int compareTo(Coin c) {
3     if(this.value==c.value)
4         return 0;
5     return this.value<c.value?-1:1;
6 }
```

- Możemy posortować monety względem wartości:

Kolekcja myCoins

1. Double eagle (1933) - $7.4 million.
2. Liberty Head nickel (1913) - $3.7 million.
3. Flowing hair silver dollar (1794) - $10.0 million.
4. Edward III florin (1343) - $6.8 million.
5. Silver dollar (1804) - $3.8 million.
6. Brasher Doubloon (1787) - $7.4 million.
7. Saint-Gaudens double eagle (1907) - $7.4 million.

Collections.sort(myCoins)

1. Liberty Head nickel (1913) - $3.7 million.
2. Silver dollar (1804) - $3.8 million.
3. Edward III florin (1343) - $6.8 million.
4. Double eagle (1933) - $7.4 million.
5. Brasher Doubloon (1787) - $7.4 million.
6. Saint-Gaudens double eagle (1907) - $7.4 million.
7. Flowing hair silver dollar (1794) - $10.0 million.

- Jakie są ograniczenia tego podejścia?

Interfejs Comparator<T>

- Interfejs `Comparable` ogranicza porównanie obiektów do jednego, domyślnego aspektu.
- Oprócz niego możemy stworzyć szereg klas określających jak porównywać obiekty poprzez implementację interfejsu `Comparator<T>`.
- Zawiera tylko jedną metodę `int compare(T o1, T o2)`.
- Pozwala to na tworzenie wielu sposobów sortowania danych.

Porządkowanie monet

- Stwórzmy komparatory dla roku i nazwy

```
public class CoinDateComparator
     implements Comparator<Coin> {
 @Override
 public int compare(Coin o1, Coin o2) {
  return o1.year-o2.year;
 }
}
```

```
public class CoinNameComparator
     implements Comparator<Coin> {
 @Override
 public int compare(Coin o1, Coin o2) {
  return o1.name.compareTo(o2.name);
 }
}
```

- Możemy teraz użyć metody `Collections.sort` z dodatkowym argumentem, implementacją klasy `Comparator`.

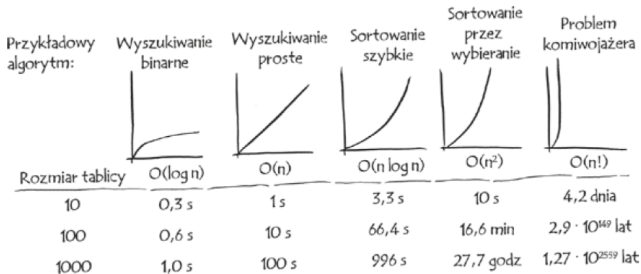
`sort(myCoins, new CoinDateComparator())`

1. Edward III florin (1343) - $6.8 million.
2. Brasher Doubloon (1787) - $7.4 million.
3. Flowing hair silver dollar (1794) - $10.0 million.
4. Silver dollar (1804) - $3.8 million.
5. Saint-Gaudens double eagle (1907) - $7.4 million.
6. Liberty Head nickel (1913) - $3.7 million.
7. Double eagle (1933) - $7.4 million.

`sort(myCoins, new CoinNameComparator())`

1. Brasher Doubloon (1787) - $7.4 million.
2. Double eagle (1933) - $7.4 million.
3. Edward III florin (1343) - $6.8 million.
4. Flowing hair silver dollar (1794) - $10.0 million.
5. Liberty Head nickel (1913) - $3.7 million.
6. Saint-Gaudens double eagle (1907) - $7.4 million.
7. Silver dollar (1804) - $3.8 million.

Koszty sortowania



Przykładowy algorytm:	Wyszukiwanie binarne	Wyszukiwanie proste	Sortowanie szybkie	Sortowanie przez wybieranie	Problem komiwojażera
Rozmiar tablicy	O(log n)	O(n)	O(n log n)	O(n²)	O(n!)
10	0,3 s	1 s	3,3 s	10 s	4,2 dnia
100	0,6 s	10 s	66,4 s	16,6 min	$2{,}9 \cdot 10^{149}$ lat
1000	1,0 s	100 s	996 s	27,7 godz	$1{,}27 \cdot 10^{2559}$ lat

Rysunek 5: Średni koszt obliczeniowy algorytmów. Czas szacowany dla 10 operacji na sekundę [Bhargava, 2017].

- Koszty sortowania wynoszą dla najlepszych algorytmów $O(n \log(n))$.
- Znacznie taniej jest na bieżąco pilnować wstawiania elementów w odpowiedniej kolejności.

Wstawianie elementów do posortowanego zbioru

- W klasie `Collections` jest statyczna metoda `binarySearch`

```
<T> int binarySearch(List<? extends Comparable<? super T>>
     list, T key)
<T> int binarySearch(List<? extends T> list, T key,
     Comparator<? super T> c)
```

- Pozwala ona na zlokalizowanie pozycji elementu `key` na posortowanej liście `list`
  - Jeżeli zwrócona wartość jest nieujemna to wskazuje szukany element
  - Jeżeli zwrócona wartość będzie ujemna to taki obiekt nie istnieje na liście i należy go wstawić na pozycji $(-(wartosc) - 1)$
  - Co jest przyczyną takiej komplikacji?

Dodanie nowej monety

Kolekcja posortowana względem roku wybicia

1. Edward III florin (1343) - $6.8 million.
2. Brasher Doubloon (1787) - $7.4 million.
3. Flowing hair silver dollar (1794) - $10.0 million.
4. Silver dollar (1804) - $3.8 million.
5. Saint-Gaudens double eagle (1907) - $7.4 million.
6. Liberty Head nickel (1913) - $3.7 million.
7. Double eagle (1933) - $7.4 million.

- Dodajemy nową monetę
  - Liberty Seated Dollar (1870) - $2.0 million.
- Jaką wartość zwróci `Collections.binarySearch` dla tej monety przy, jeżeli drugim argumentem będzie `CoinYearComparator`?
  - -6
- Na której pozycji powinniśmy dodać tę monetę?
  - $-(-6) - 1 = 5$

Dodanie tej samej monety

Kolekcja posortowana względem nazwy

1. Brasher Doubloon (1787) - $7.4 million.
2. Double eagle (1933) - $7.4 million.
3. Edward III florin (1343) - $6.8 million.
4. Flowing hair silver dollar (1794) - $10.0 million.
5. Liberty Head nickel (1913) - $3.7 million.
6. Liberty Seated Dollar (1870) - $2.0 million.
7. Saint-Gaudens double eagle (1907) - $7.4 million.
8. Silver dollar (1804) - $3.8 million.

- Ponownie dodajemy monetę
  - Liberty Seated Dollar (1870) - $2.0 million.
- Jaką wartość zwróci `Collections.binarySearch` dla tej monety przy, jeżeli drugim argumentem będzie `CoinNameComparator`?
  - 5

Mapowanie

- Mapowanie przekształca jeden typ w drugi.
- Jest używane do odnalezienia wartości odpowiadającej danemu kluczowi.
- Interfejs `AbstractMap` może być zaimplementowany jako drzewo `TreeMap` lub tablica haszująca `HashMap`

Wyliczanie mody stosując mapowanie

```
1        public static <T> T calculateMode(ArrayList<T> timeSeries) {
2            T modeValue = null;
3            int maxCount = 0;
4
5            HashMap<T,Integer> counter = new HashMap();
6            for (T value : timeSeries) {
7                Integer val = counter.getOrDefault(value,0);
8                val++;
9                counter.put(value,val); //constant-time performance
10               if (val > maxCount) {
11                   maxCount = val;
12                   modeValue = value;
13               }
14           }
15           return modeValue;
16       }
```

Najpopularniejsza moneta

- Moja kolekcja
  1. Double eagle (1933) - $7.1 million.
  2. Double eagle (1933) - $7.4 million.
  3. Silver dollar (1804) - $3.5 million.
  4. Silver dollar (1804) - $3.3 million.
  5. Silver dollar (1804) - $3.8 million.
  6. Brasher Doubloon (1787) - $7.4 million.
- Wynik

```
The most common coin is:
    Silver dollar (1804) - $3.8 million.
```

Dyskusja wyników

```
Silver dollar (1804) - $3.8 million.
```

- Otrzymaliśmy wynik będący jedną z trzech instancji klasy `Coin`.
- Nazwa i rok wybicia są wspólne dla całej grupy, ale wartość już nie.
- Dzieje się tak ponieważ klasa `HashMap` identyfikuje klucze korzystając z metod `equals` i `hashCode`.
  - Implementacja zastosowana w klasie `Coin` porównuje nazwę i rok wybicia monety.

Bibliografia

[Bhargava, 2017] Bhargava, A. Y. (2017).
Algorytmy ilustrowany przewodnik.
Helion.

[Horstmann, 2016] Horstmann, C. S. (2016).
Java. Podstawy.
Helion.

[Wikipedia, 2018] Wikipedia (2018).
Merge sort.