



Kolekcije

Programski jezici II

Kolekcije

- kolekcija – nekad se naziva i kontejner
- kolekcija – je objekat koji grupiše više elemenata u jednu jedinicu
- koriste se za pohranjivanje, dobijanje, manipulaciju i komunikaciju skupom podataka
- obično reprezentuju jedinice podataka koji čine prirodnu grupu – npr.:
 - mail folder – kolekcija poruka
 - telefonski imenik – mapiranje imena u telefonske brojeve
- ranije verzije Jave (prije 1.2) obuhvatale se Vector, Hashtable i niz – implementacije kolekcija
- ranije verzije Jave (prije 1.2) nisu posjedovale Collections framework

Collection Framework

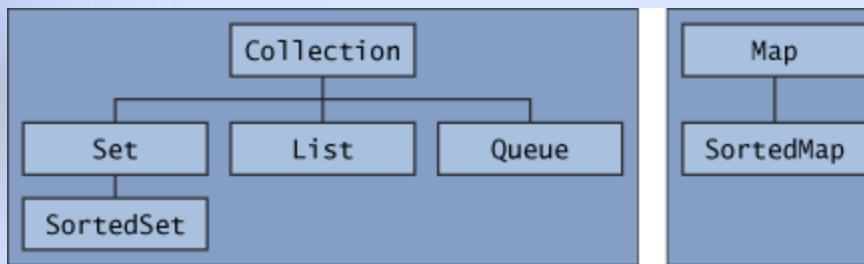
- Collections framework je unificirana arhitektura za reprezentaciju i manipulaciju kolekcijama
- posjeduje:
 - interfejse – koji omogućavaju manipulaciju kolekcijama nezavisno od njihove konkretne implementacije
 - implementacije – konkretne implementacije interfejsa
 - algoritme – metode koje vrše korisne operacije, kao što su pretraživanje i sortiranje, nad objektima klasa koje implementiraju interfejse kolekcija – algoritmi su polimorfni, tj. ista metoda se može koristiti nad mnogim različitim implementacijama odgovarajućeg interfejsa kolekcija

Java Collections Framework

- prednosti:
 - redukuje programerski napor
 - povećava brzinu i kvalitet programa
 - omogućava interoperabilnost između nepovezanih API-ja
 - redukuje napor neophodan za učenje i korištenje novih API-ja
 - redukuje napor neophodan za dizajniranje novih API-ja
 - ohrabruje reupotrebu koda

Java Collections Framework

- osnovni interfejsi kolekcija omogućavaju manipulaciju kolekcijama nezavisno od detalja njihove reprezentacije
- osnovni interfejsi kolekcija su temelj JCF-a
- osnovni interfejsi kolekcija kreiraju hijerarhiju



- postoje 2 stabla kolekcija
- osnovni interfejsi kolekcija su generički – npr. deklaracija Collection interfejsa

```
public interface Collection<E>
```

Java Collections Framework

- Collection – root hijerarhije kolekcija
- interfejs Collection je najmanji zajednički imenitelj kojeg implementiraju sve kolekcije i koristi se za prosljeđivanje i manipulaciju kolekcijama u slučaju kada je potrebna maksimalna generalizacija
- implementacije interfejsa iz JCF ne moraju podržavati sve operacije
- ako je pozvana operacija koja nije podržana, biće bačen UnsupportedOperationException izuzetak
- kolekcija predstavlja grupu objekata – elemenata, pri čemu:
 - neke kolekcije dozvoljavaju duple elemente, a neke ne
 - neke kolekcije su uređene, a neke nisu
- ne postoji direktna implementacija interfejsa Collection, ali postoje implementacije podinterfejsa, npr. Set i List

Java Collections Framework

- Set
 - predstavlja skup jedinstvenih elemenata – kolekcija koja ne može sadržavati duple elemente. Ovaj interfejs predstavlja apstrakciju matematičkog skupa.
- SortedSet
 - nasljeđuje Set interfejs dodajući mu mogućnost sortiranja elemenata na odgovarajući način
- NavigableSet
 - nasljeđuje SortedSet interfejs proširujući ga navigacionim metodama
- List
 - uređena kolekcija (nekad se naziva i sekvenca)
 - može sadržavati duple elemente
 - elementima se može pristupiti na bazi indeksa (pozicije)
 - sličnost sa Vector klasom
- Queue
 - kolekcija koja sadrži elemente koje se trebaju procesirati
 - obično, ali ne obavezno, uređuju elemente u FIFO red – izuzetak su prioritetni redovi
 - glavu reda čini element koji će biti uklonjen pozivom metode remove ili poll
 - novi elementi se dodaju na kraj reda, rep
 - drugi tipovi redova mogu koristiti drugačija pravila za smještanje elemenata
- Deque
 - nasljeđuje Queue interfejs na takav način da predstavlja red čiji se elementi mogu procesirati s oba kraja

Java Collections Framework

- Map
 - objekat koji mapira ključeve u vrijednosti
 - ne može sadržavati duple ključeve – svaki ključ se može mapirati u najviše jednu vrijednost
 - sličnost sa Hashtable klasom
- SortedMap
 - Map koji sadrži mapiranja na takav način da su ključevi u rastućem poretku
 - Map analogija SortedSet-a
- NavigableMap
 - nasljeđuje SortedMap interfejs proširujući ga navigacionim metodama

Collection interfejs

- Collection interfejs predstavlja grupu objekata - elemenata
- Collection interfejs se koristi za manipulaciju kolekcijama u slučaju kada je potrebna maksimalna generalizacija
- sve implementacije kolekcija, po konvenciji, imaju konstruktor koji kao argument ima Collection
- ovaj konstruktor omogućava konverziju tipova kolekcije – naziva se i *conversion* konstruktor
- primjer
 - Collection<String> c koji može biti List, Set, ili druga kolekcija (Collection tipa).
`List<String> list = new ArrayList<String>(c);`
 - kreirana je ArrayList (implementacija List interfejsa), koja inicijalno sadrži sve elemente iz c

Collection interfejs

- `boolean add(E e)`
 - obezbeđuje da kolekcija sadrži specificirani element – ako e nije u kolekciji biće dodati; ako kolekcija dozvoljava duple elemente e će biti dodat
 - vraća true ako je kolekcija izmijenjena
 - opcionalna operacija
- `boolean addAll(Collection<? extends E> c)`
 - dodaje sve elemente specificirane kolekcije u tekuću kolekciju
 - opcionalna operacija
- `void clear()`
 - uklanja sve elemente iz kolekcije
 - opcionalna operacija
- `boolean contains(Object o)`
 - vraća true ako kolekcija sadrži specificirani element
- `boolean containsAll(Collection<?> c)`
 - vraća true ako kolekcija sadrži sve elemente specificirane kolekcije
- `boolean equals(Object o)`
 - poređenje specificiranog objekta sa tekućom kolekcijom
- `int hashCode()`
 - vraća hash code tekuće kolekcije
- `boolean isEmpty()`
 - vraća true ako kolekcija ne sadrži elemente

Collection interfejs

- `Iterator<E> iterator()`
 - vraća iterator nad elementima tekuće kolekcije
 - `boolean remove(Object o)`
 - uklanja jednu instancu specificiranog elementa iz kolekcije, ako je prisutan
 - opcionala operacija
 - `boolean removeAll(Collection<?> c)`
 - uklanja sve elemente tekuće kolekcije koji su sadržani u specificiranoj kolekciji
 - opcionala operacija
 - `boolean retainAll(Collection<?> c)`
 - ostavlja sve elemente tekuće kolekcije koji su sadržani u specificiranoj kolekciji
 - opcionala operacija
 - `int size()`
 - vraća broj elemenata kolekcije
 - `Object[] toArray()`
 - vraća niz koji sadrži sve elemente kolekcije
 - `<T> T[] toArray(T[] a)`
 - vraća niz koji sadrži sve elemente kolekcije, runtime tip vraćenog niza je identičan tipu specificiranog niza
- ```
String[] y = x.toArray(new String[0]);
```

# Collection interfejs

```
public class CollectionsTest {
 public static void main(String[] args) {
 Collection<Integer> collection = new
ArrayList<Integer>();
 System.out.println(collection.isEmpty());
 collection.add(1);
 collection.add(5);
 collection.add(10);
 System.out.println(collection.contains(5));
 collection.remove(5);
 System.out.println(collection.contains(5));
 for(Integer i: collection)
 System.out.println(i);
 System.out.println(collection.isEmpty());
 }
}
```

# Collection interfejs

- obilazak kolekcija

- for-each konstrukcija

```
for (Object o : collection)
 System.out.println(o);
```

- iteratori

```
public interface Iterator<E> {
 boolean hasNext();
 E next();
 void remove(); //opciono
}
```

---

```
public interface Collection<E> extends Iterable<E>
```

- Iterator se koristi umjesto for-each konstrukcije kada je neophodno:
  - ukloniti tekući element – for-each konstrukcija sakriva iterator, tako da se ne može pozvati remove()
  - iterirati preko višestrukih kolekcija paralelno

# Collection interfejs

- obilazak kolekcija

```
public class CollectionsTest {
 public static void main(String[] args) {
 Collection<Integer> collection = new
ArrayList<Integer>();
 collection.add(1);
 collection.add(5);
 collection.add(10);
 for(Integer i: collection) // 1
 System.out.println(i);
 Iterator<Integer> iterator =
collection.iterator();
 while(iterator.hasNext()) // 2
 System.out.println(iterator.next());
 }
}
```

# Set interfejs

- Set je kolekcija koja ne može sadržavati duple elemente
- apstrakcija je matematičkog skupa
- Set interfejs sadrži samo metode nasljeđene iz Collection i dodaje restrikciju – zabranjuje duple elemente

`a.addAll(b)`

$a = a \cup b$

`a.removeAll(b)`

$a = a - b$

`a.retainAll(b)`

$a = a \cap b$

`a.containsAll(b)`

$a \subseteq b$

`a.clear()`

$a = \emptyset$

# Set interfejs

- Set uvodi "snažniji" ugovor po pitanju ponašanja equals i hashCode metoda – omogućava da se Set instance porede bez obzira na njihovu implementaciju – dvije Set instance su jednake, ako sadrže iste elemente

```
public interface Set<E> extends Collection<E> {
 int size();
 boolean isEmpty();
 boolean contains(Object element);
 boolean add(E element);
 boolean remove(Object element);
 Iterator<E> iterator();
 boolean containsAll(Collection<?> c);
 boolean addAll(Collection<? extends E> c);
 boolean removeAll(Collection<??> c);
 boolean retainAll(Collection<??> c);
 void clear(); // opcionalno
 Object[] toArray();
 <T> T[] toArray(T[] a);
 boolean equals(Object o);
 int hashCode();
}
```

# Set interfejs – implementacije

- HashSet i LinkedHashSet
- HashSet – čuva elemente u hash tabeli – implementacija sa najboljim performansama – nema garancija po pitanju redoslijeda iteracija
- LinkedHashSet – implementirana kao hash tabela sa ulančanom listom – uređuje elemente prema redoslijedu ubacivanja

# SortedSet interfejs

- nasljeđuje Set interfejs dodajući mu mogućnost sortiranja elemenata na odgovarajući način
- svi elementi SortedSet-a moraju implementirati Comparable interfejs (ili implementirati Comparator)
- metode

```
SortedSet<E> subSet(E fromElement, E toElement);
SortedSet<E> headSet(E toElement);
SortedSet<E> tailSet(E fromElement);
E first();
E last();
Comparator<? super E> comparator();
```

- obilazak elemenata korištenjem for petlje ili iteratora će biti obavljen na način kako su elementi sortirani

# NavigableSet interfejs

- interfejs NavigableSet nasljeđuje SortedSet interfejs proširujući ga navigacionim metodama
- navigacione metode služe za pronađak elemenata u kolekciji koja implementira interfejs NavigableSet
- preporuka je da se koristi umjesto SortedSet interfejsa

# NavigableSet interfejs

```
SortedSet<E> subSet(E fromElement, E toElement) // 1
NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
E toElement, boolean toInclusive) // 2
SortedSet<E> headSet(E toElement) // 3
NavigableSet<E> headSet(E toElement, boolean inclusive) // 4
SortedSet<E> tailSet(E fromElement) // 5
NavigableSet<E> tailSet(E fromElement, boolean inclusive)// 6
E ceiling(E e) // 7
E floor(E e) // 8
E higher(E e) // 9
E lower(E e) // 10
E pollFirst() // 11
E pollLast() // 12
Iterator<E> iterator() // 13
Iterator<E> descendingIterator() // 14
NavigableSet<E> descendingSet() // 15
```

# NavigableSet interfejs – implementacije

- klasa TreeSet implementira NavigableSet interfejs, a samim tim i SortedSet interfejs
- elementi ovog skupa, tj. elementi objekta ove klase, nalaze se u prirodnom poretku ili u poretku koji definiše komparator
  - koji od ova dva načina će biti primjenjen zavisi od korištenog konstruktora
- implementacija TreeSet klase bazirana je na balansiranom stablu, tako da je složenost osnovnih operacija (dodavanja elementa, uklanjanja elementa i pretraživanje) nad ovom strukturom  $O(\log_n)$
- HashSet ima bolje performanse od TreeSet-a kad je u pitanju pretraživanje elemenata, dok TreeSet ima bolje performanse kada elementi moraju biti sortirani, a kada se zahtjeva njihovo brzo pretraživanje i ubacivanje

# TreeSet – primjer

```
public class TreeSetTest {
 public static void main(String[] args) {
 NavigableSet<Integer> ns = new
 TreeSet<Integer>(); // 1
 ns.add(1); // 2
 ns.add(9); // 3
 ns.add(6); // 4
 ns.add(35); // 5
 ns.add(12); // 6
 for (Iterator<Integer> it = ns.iterator();
it.hasNext()); // 7
 Integer integer = (Integer) it.next();
 System.out.println(integer);
 }
 for (Iterator<Integer> it =
ns.descendingIterator(); it.hasNext()); { // 8
 Integer integer = (Integer) it.next();
 System.out.println(integer);
 }
 }
}
```

# HashSet vs TreeSet

- HashSet:
  - konstantno vrijeme za osnovne operacije (add, remove, contains i size)
  - ne garantuje da će poredak elemenata biti očuvan
  - vrijeme iteracije kroz ovaj skup je proporcionalno sumi broja elemenata (veličini HashSet instance) i kapaciteta pozadinske HashMap instance – zato je bitno da inicijalni kapacitet ne bude suviše veliki (ili da load factor ne bude suviše mali), ako su performanse iteracije bitne)
- TreeSet:
  - složenost osnovnih operacija (add, remove i contains) nad ovom strukturom  $O(\log_n)$
  - garantuje da će elementi skupa biti sortirani (rastuće u prirodnom poretku ili u poretku koji definiše komparator)
  - ne nudi parametre čijim podešavanjem se može uticati na performanse iteracije
  - nudi “zgodne” metode za rad sa skupovima, poput first(), last(), headSet(), tailSet() i sl.

# List interfejs

- List je uređena kolekcija – Collection (nekad se naziva i sekvenca)
- liste mogu sadržavati duple elemente
- pored metoda nasleđenih iz Collection interfejsa, List interfejs dodaje i metode za:
  - pozicioni pristup – manipuliše elementima na bazi njihove pozicije u listi
  - pretragu – pretraživanje objekta u listi i vraćanje njegove numeričke pozicije
  - iteraciju
  - range-view –range operacije nad listom, npr. subList
- sličnost sa klasom Vector – uklanja nekoliko manjih API nedostataka klase Vector – npr., odgovarajuće metode elementAt i setElementAt iz klase Vector imaju kraća imena – get i set

```
niz: x[3] = "abc";
Vector: x.setElementAt("abc", 3);
List: x.set(3, "abc");
```

```
niz: a[i] = a[j].method(a[k]);
Vector: v.setElementAt(v.elementAt(j).method(v.elementAt(k)), i);
List: v.set(i, v.get(j).method(v.get(k)));
```

# List interfejs

```
public interface List<E> extends Collection<E> {
 // pozicioni pristup
 E get(int index);
 E set(int index, E element); // opciona
 boolean add(E element); // opciona
 void add(int index, E element); // opciona
 E remove(int index); // opciona
 boolean addAll(int index, Collection<? extends
E> c); // opciona
 // pretraga
 int indexOf(Object o);
 int lastIndexOf(Object o);
 ListIterator<E> listIterator();
 ListIterator<E> listIterator(int index);
 // dio liste
 List<E> subList(int from, int to);
}
```

# List interfejs – implementacija

- klasa ArrayList implementira List interfejs
- riječ je o nizu promjenljive dužine
- ova klasa implementira sve opcione metode List interfejsa i dozvoljava smještaj elemenata bilo kojeg tipa, uključujući i null
- ova klasa je slična Vector klasi, osim što nije sinhronizovana
- složenost metoda size, isEmpty, get, set, iterator i listIterator je  $O(1)$
- metoda add se izvršava u amortizovanom konstantnom vremenu, tj. složenost dodavanja n elemenata iznosi  $O(n)$
- sve druge metode izvršavaju se u linearном vremenu, tj. njihova složenost iznosi  $O(n)$
- svaka instanca klase ArrayList ima kapacitet
  - kapacitet predstavlja veličinu niza koja je potrebna da se smjeste elementi liste
  - kapacitet je, minimalno, jednak veličini liste
  - kako se elementi dodaju u listu, tako se povećava i kapacitet
  - ensureCapacity – korisna metoda
- nije sinhronizovana – potrebna je eksterna sinhronizacija

# List interfejs – implementacija

- klasa `LinkedList` implementira `List` interfejs
- ova klasa implementira sve opcione metode `List` interfejsa i dozvoljava smještaj elemenata bilo kojeg tipa, uključujući i `null`
- ova klasa implementira i `Queue` i `Deque` interfejse, tako da se može koristiti za kreiranje steka i različitih vrsta redova

# List interfejs – implementacija

- klasa Vector implementira List interfejs
- ova klasa je slična ArrayList klasi, s tim što je i sinhronizovana
- objekat klase Vector efikasno može da upravlja prostorom za smještanje elemenata podešavanjem njegovog kapaciteta i veličine inkrementa u kojima se povećava kapacitet
- kapacitet je, najčešće, veći od veličine ove kolekcije, jer se kapacitet uvećava u inkrementima
- veličina inkrementa definisana je atributom capacityIncrement

# Queue interfejs

- interfejs Queue nasljeđuje Collection interfejs na takav način da predstavlja red
- obično, ali ne obavezno, elementi se uređuju prema FIFO algoritmu
- moguće je da elementi budu uređeni i prema LIFO (*Last In First Out*) algoritmu ili prema prioritetu
- kod prioritetnih redova, elementi mogu biti u prirodnom poretkom ili poretku koji je određen komparatorom

```
boolean add(E e)
E element()
boolean offer(E e)
E peek()
E poll()
E remove()
```

- Queue interfejs – metode kojim proširuje Collection

# Queue interfejs

- svaka Queue metoda postoji u dva oblika, i to:
  - metoda baca izuzetak ako operacija ne uspije
  - metoda vraća specijalnu vrijednost ako operacija ne uspije – null ili false, zavisno od operacije

|                | baca izuzetak | vraća specijalnu vrijednost |
|----------------|---------------|-----------------------------|
| <b>insert</b>  | add(e)        | offer(e)                    |
| <b>remove</b>  | remove()      | poll()                      |
| <b>examine</b> | element()     | peek()                      |

# Queue interfejs – implementacije

- klasa PriorityQueue implementira interfejs Queue
- ako se ne traži mogućnost bidirekcionog obilaska, onda je preporuka da se koristi ova implementacija, prije nego LinkedList
- klasa PriorityQueue predstavlja implementaciju reda sa prioritetnim poretkom elementata
- ova implementacija bazirana je na prioritetnom *heap-u*
- u slučaju kada postoji više elemenata sa istim prioritetom, poredek tih elemenata se određuje proizvoljno
- ova klasa nije sinhronizovana
- složenost metoda offer, poll, add i remove iznosi  $O(\log(n))$
- metode remove(Object) i contains(Object) izvršavaju se u linearном vremenu, tj. njihova složenost iznosi  $O(n)$
- metode peek, element i size izvršavaju se u konstantnom vremenu, tj. njihova složenost iznosi  $O(1)$

# Queue interfejs – implementacije

- klasa LinkedList implementira interfejs Queue

```
public class LinkedListAsQueueTest {
 public static void main(String[] args) {
 LinkedList<Integer> queue = new
 LinkedList<Integer>();
 queue.offer(3); // 1
 queue.offer(8); // 2
 queue.offer(6); // 3
 queue.offer(14); // 4
 queue.offer(1); // 5
 System.out.print("Uklanjanje: ");
 while(!queue.isEmpty())
 System.out.print(queue.poll() + " "); // 6
 }
}
```

# Interfejs Deque

- interfejs Deque nasljeđuje Queue interfejs na takav način da predstavlja linearu kolekciju koja podržava ubacivanje i uklanjanje elemenata sa oba kraja, tj. dozvoljava i operacije nad repom, a ne samo nad glavom reda
- većina implementacija ovog interfejsa nema ograničenje broja elemenata koje može sadržavati, mada interfejs obezbjeđuje i tu mogućnost
- klase koje implementiraju ovaj interfejs mogu se koristiti kao FIFO redovi, s tim da se elementi dodaju s repa, a uzimaju s glave reda
- pored toga, klase koje implementiraju ovaj interfejs mogu se koristiti kao stek, s tim da se elementi dodaju i uzimaju sa iste strane reda, prema LIFO algoritmu

# Interfejs Deque

```
boolean offerFirst(E element)
boolean offerLast(E element)
void push(E element)
void addFirst(E element)
void addLast(E element)
E pollFirst()
E pollLast()
E pop()
E removeFirst()
E removeLast()
boolean removeFirstOccurrence(Object obj)
boolean removeLastOccurrence(Object obj)
E peekFirst()
E peekLast()
E getFirst()
E getLast()
```

# Interfejs Deque

- klasa ArrayDeque implementira interfejs Deque
- ova implementacija nema ograničenje kapaciteta, tj. objekat ove klase dinamički raste koliko je to potrebno
- ova klase nije sinhronizovana, tj. bez eksterne sinhronizacije, ona ne podržava konkurentni pristup od strane više niti
- u ovu kolekciju nije dozvoljeno dodavati null elemente
- elementi ove strukture se mogu obilaziti od glave prema repu ili obrnuto, od repa prema glavi
- pozicioni pristup elementima nije moguć, niti ih je moguće sortirati
- većina metoda ArrayDeque klase izvršava se u amortizovanom konstantnom vremenu. Tako, na primjer, složenost dodavanja n elemenata iznosi  $O(n)$ .
- izuzetak su metode remove, removeFirstOccurrence, removeLastOccurrence, contains, iterator.remove() i bulk metode, koje se izvršavaju u linearном vremenu, tj. njihova složenost iznosi  $O(n)$

# Interfejs Deque

```
public class ArrayDequeTest {
 public static void main(String[] args) {
 ArrayDeque<Integer> stack = new
ArrayDeque<Integer>();
 stack.push(3); // 1
 stack.push(9); // 2
 stack.push(6); // 3
 System.out.print("Skidanje sa steka: ");
 while(!stack.isEmpty())
 System.out.print(" " + stack.pop()); // 4

 ArrayDeque<Integer> fifoQueue = new
ArrayDeque<Integer>();
 fifoQueue.offerLast(3); // 5
 fifoQueue.offerLast(9); // 6
 fifoQueue.offerLast(6); // 7
 System.out.print("\nUzimanje iz FIFO reda: ");
 while(!fifoQueue.isEmpty())
 System.out.print(" "
+ fifoQueue.pollFirst()); // 8
 }
}
```

# Interfejs Deque

- klasa LinkedList implementira i interfejs Deque

```
public class LinkedListAsStack {
 public static void main(String[] args) {
 LinkedList<Integer> stack = new
LinkedList<Integer>();
 stack.push(3);
 stack.push(9);
 stack.push(6);
 System.out.print("Skidanje sa steka: ");
 while(!stack.isEmpty())
 System.out.print(" " + stack.pop());
 }
}
```

# Map interfejs

- interfejs Map predstavlja korijen drugog stabla hijerarhije kolekcija
- ovaj interfejs definiše metode za rad sa mapiranjima (parovima) ključ-vrijednost
- ova struktura ne može da sadrži duple ključeve, a svaki ključ se može mapirati u najviše jednu vrijednost
- ne postoji veza više-prema-jedan između ključeva i vrijednosti
- i ključevi i vrijednosti moraju biti objekti.
- interfejs Map specificira ugovor koji sve mape treba da implementiraju – neke od metoda ovog interfejsa su opcione
- mapa koja ne implementira neku od metoda ovog interfejsa obavezna je da obezbijedi da data metoda baci izuzetak UnsupportedOperationException, u slučaju njenog poziva
- sličnost sa HashTable klasom

# Map interfejs

```
public interface Map<K,V> {
 // osnovne operacije
 V put(K key, V value);
 V get(Object key);
 V remove(Object key);
 boolean containsKey(Object key);
 boolean containsValue(Object value);
 int size();
 boolean isEmpty();
 // grupne operacije
 void putAll(Map<? extends K, ? extends V> m);
 void clear();
 // konverzije
 public Set<K> keySet();
 public Collection<V> values();
 public Set<Map.Entry<K,V>> entrySet();
 // interfejs za entrySet elemente
 public interface Entry {
 K getKey();
 V getValue();
 V setValue(V value);
 }
}
```

# Map interfejs – implementacije

- klasa HashMap implementira Map interfejs
- ova klasa predstavlja neuređenu mapu (nema garancije poretku elemenata), implementira sve opcione metode Map interfejsa i dozvoljava dodavanje jednog null ključa u ovu strukturu
- ova klasa nije sinhronizovana, tj. bez eksterne sinhronizacije, ona ne podržava konkurentni pristup od strane više niti
- osnovne metode za uzimanje elemenata iz, i dodavanje elemenata u ovu strukturu izvršavaju se u konstantnom vremenu, tj. njihova složenost iznosi  $O(1)$
- iteracija kroz elemente kolekcije nastale pozivom metoda za konverziju u kolekcije zahtijeva vrijeme proporcionalno kapacitetu objekta klase HashMap i njegovoj veličini (broju ključ-vrijednost mapiranja)
  - iz ovog razloga bitno je da inicijalni kapacitet ne bude suviše veliki, ako su performanse iteracije kroz elemente važne

# Map interfejs – implementacije

- instanca HashMap klase ima dva parametra koji utiču na performanse: inicijalni kapacitet i faktor opterećenja
- faktor opterećenja predstavlja mjeru maksimalno dozvoljene popunjenošći hash tabele prije automatskog povećanja njenog kapaciteta – kada broj elemenata (mapiranja) u hash tabeli postane veći od proizvoda faktora opterećenja i trenutnog kapaciteta, izvršiće se izmjena interne strukture hash tabele, na takav način da će se njen kapacitet udvostručiti – podrazumijevani faktor opterećenja iznosi 0.75 i predstavlja dobar izbor u većini slučajeva – veća vrijednost faktora opterećenja smanjuje gubitak prostora, ali povećava vrijeme izvršavanja većine metoda HashMap klase – ako je potrebno dodati veliki broj mapiranja u objekat klase hashMap, preporuka je da se ovaj objekat kreira sa dovoljno velikim kapacitetom - ovo će omogućiti efikasnije dodavanje velikog broja mapiranja, nego u slučaju kada se kapacitet interne hash tabele automatski povećava

# Map interfejs – implementacije

```
public class HashMapTest {
 public static void main(String[] args) {
 HashMap<Integer, String> hm = new
 HashMap<Integer, String>(); // 1
 hm.put(1, "1"); // 2
 hm.put(2, "2"); // 3
 hm.put(4, "4"); // 4
 hm.put(3, "3"); // 5
 System.out.println("Velicina: " + hm.size()); // 6
 System.out.println("Kljuc 3: " +
 hm.containsKey(3)); // 7
 System.out.println("Vrijednost 2: " +
 hm.containsValue("2")); // 8
 hm.remove(3); // 9
 System.out.println("Velicina: " + hm.size()); // 10
 }
}
```

# Map interfejs – implementacije

- klasa LinkedHashMap nasljeđuje klasu HashMap
- veza između LinkedHashMap i HashMap klasa analogna je vezi između LinkedHashSet i HashSet klasa
- elementi (mapiranja) HashMap i HashSet objekata su neuređeni, a elementi LinkedHashMap i LinkedHashSet objekata su uređeni
- ova implementacija razlikuje se od HashMap implementacije u tom što održava dvostruko-ulančanu listu svih mapiranja
- ovom ulančanom listom definiše se poredak elemenata
- podrazumijevani poredak je poredak u kojem su ključevi dodavani u ovu strukturu
  - poredak se ne mijenja ponovnim dodavanjem istog ključa, jer mapiranje sa ključem koji već postoji u ovoj strukturi neće biti dodato
- elementi se mogu čuvati i u poretku nastalom pristupom elementima
  - ovaj poredak se može specificirati odgovarajućim konstruktorom

```
LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)
```

# Map interfejs – implementacije

- klasa `Hashtable` implementira Map interfejs
- ova klasa implementira neuređenu mapu, sinhronizovana je i ne dozvoljava dodavanje null ključeva i null vrijednosti
- instanca `Hashtable` klase ima dva parametra koji utiču na performanse: inicijalni kapacitet i faktor opterećenja – podrazumijevano 11 i 0.75
  - podrazumijevani faktor opterećenja iznosi 0.75 i obično predstavlja dobar izbor – veća vrijednost faktora opterećenja smanjuje gubitak prostora, ali povećava vrijeme izvršavanja većine metoda `Hashtable` klase
  - ako je potrebno dodati veliki broj mapiranja u objekat klase `Hashtable`, preporuka je da se ovaj objekat kreira sa dovoljno velikim kapacitetom – ovo će omogućiti efikasnije dodavanje velikog broja mapiranja, nego u slučaju kada se kapacitet hash tabele automatski povećava

# Interfejs SortedMap

- interfejs SortedMap nasljeđuje interfejs Map i analogan je interfejsu SortedSet
- objekti klasa koje implementiraju interfejs SortedMap obezbjeđuju poredak elemenata (mapiranja) prema prirodnom poretku ključeva ili prema komparatoru
- svi ključevi sortirane mape moraju implementirati Comparable interfejs i moraju biti uzajamno komparabilni, što znači da se pri poređenju bilo koja dva ključa sortirane mape neće desiti izuzetak ClassCastException

# Interfejs NavigableMap

- interfejs NavigableMap nasljeđuje interfejs SortedMap i analogan je interfejsu NavigableSet
- preporuka je da se koristi umjesto SortedMap interfejsa

# Interfejs NavigableMap - implementacije

- klasa TreeMap implementira NavigableMap interfejs, a samim tim i SortedMap interfejs
- riječ je o crveno-crnom stablu
- mapiranja u ovoj mapi nalaze se u prirodnom poretku ili u poretku koji definiše komparator
  - koji od ova dva načina će biti primjenjen zavisi od korištenog konstruktora
- ova implementacija garantuje  $\log(n)$  složenost metoda containsKey, get, put i remove
- ova klasa nije sinhronizovana

# Klase Collections

- klasa Collections obezbjeđuje različite metoda za rad sa kolekcijama, poput sortiranja, pretraživanja i zamjene elemenata u kolekciji
- sve metode ove klase su deklarisane kao static i public
- u slučaju kada se kao neki od argumenata neke od metoda ove klase proslijedi null referenca, umjesto reference na kolekciju, desiće se izuzetak NullPointerException
- veliki broj metoda
  - u Java verziji 8, ukupno 66 metode ove klase

# Klase Collections

- metode za sortiranje

```
<T extends Comparable<? super T>> void sort(List<T> list) //1
<T> void sort(List<T> list, Comparator<? super T> c) // 2
void reverse(List<?> list) // 3
<T> Comparator<T> reverseOrder() // 4
<T> Comparator<T> reverseOrder(Comparator<T> comparator) // 5
static void rotate(List<?> list, int distance) // 6
static void shuffle(List<?> list) // 7
static void shuffle(List<?> list, Random rnd) // 8
static void swap(List<?> list, int i, int j) // 9
```

# Klase Collections

- List I može se sortirati pomoću  
`Collections.sort(l);`
- ako se ova lista sastoji od String elemenata, biće sortiran u alfabetском poretku
- ako se ova lista sastoji od Date elemenata, biće sortiran u hronološkom poretku
- i String i Date implementiraju Comparable interfejs, kao i mnoge druge klase (Integer, Float, Double, Character, Byte, Boolean,...)
- implementacije Comparable interfejsa obezbjeđuju prirodni poređak objekata, što omogućava da objekti ove klase budu automatski sortirani

# Klasa Collections

```
public class CollectionsSortTest {
 public static void main(String[] args) {
 LinkedList<Integer> ll = new
 LinkedList<Integer>(); // 1
 ll.add(1);ll.add(3);ll.add(7); // 2
 ll.add(9);ll.add(6);ll.add(7); // 3
 iterate(ll);
 Collections.sort(ll); // 4
 iterate(ll);
 Collections.reverse(ll); // 5
 iterate(ll);
 Collections.rotate(ll, 3); // 6
 iterate(ll);
 Collections.shuffle(ll); // 7
 iterate(ll);
 }
 private static void iterate(LinkedList<Integer> ll){
 for (Iterator<Integer> it = ll.iterator();
it.hasNext();) {
 System.out.print(" " + it.next());
 System.out.println("\n =====");
 }
 }
}
```

# Klase Collections

- pretraživanje kolekcija

```
<T> int binarySearch(List<? extends Comparable<? super T>>
list, T key) // 1
<T> int binarySearch(List<? extends T> list, T key,
Comparator<? super T> c)) // 2
int indexOfSubList(List<?> source, List<?> target) // 3
int lastIndexOfSubList(List<?> source, List<?> target) // 4
<T extends Object & Comparable<? super T>> T
max(Collection<? extends T> c) // 5
<T> T max(Collection<? extends T> c, Comparator<? super T>
comp) // 6
<T extends Object & Comparable<? super T>> T
min(Collection<? extends T> c) // 7
<T> T min(Collection<? extends T> cl, Comparator<? super T>
comp) // 8
```

# Klasa Collections

```
public class CollectionsSearchTest {
 public static void main(String[] args) {
 LinkedList<Integer> ll = new
LinkedList<Integer>();
 ll.add(1);ll.add(3);ll.add(7);
 ll.add(9);ll.add(6);ll.add(7);
 Collections.sort(ll);
 for (Iterator<Integer> it = ll.iterator();
it.hasNext();) {
 System.out.print(it.next() + " ");
 }
 System.out.println("\nPretraga 6: " +
Collections.binarySearch(ll, 6));
 System.out.println("Pretraga 2: " +
Collections.binarySearch(ll, 2));
 System.out.println("Max: " +
Collections.max(ll));
 System.out.println("Min: " +
Collections.min(ll));
 }
}
```

# Klase Collections

- zamjena elemenata u kolekcijama

```
<E> boolean addAll(Collection<? super E> collection, E...
elements)
<E> void copy(List<? super E> destination, List<? extends E>
source)
<E> void fill(List<? super E> list, E element)
<E> boolean replaceAll(List<E> list, E oldVal, E newVal)
<E> List<E> nCopies(int n, E element)
```

# Klase Collections

- sinhronizacija kolekcija

```
Collection c = Collections.synchronizedCollection(myCollection);
...
synchronized(c) {
 Iterator i = c.iterator(); //mora biti u sinhronizovanom bloku
 while (i.hasNext())
 func(i.next());
}

synchronizedSet
synchronizedSortedSet
synchronizedList
synchronizedMap
synchronizedSortedMap
```

# Klase Arrays

- klasa Arrays obezbjeđuje različite metoda za rad sa nizovima, poput sortiranja, pretraživanja i zamjene elemenata u nizovima
- sve metode ove klase su deklarisane kao static i public
- u slučaju kada se kao neki od argumenata neke od metoda ove klase proslijedi null referenca, umjesto reference na kolekciju, desiće se izuzetak NullPointerException
- veliki broja metoda
  - u Java verziji 7, ukupno 155 metoda

# Klase Arrays

- sortiranje nizova

```
void sort(byte[] a)
void sort(byte[] a, int fromIndex, int toIndex)
<T> void sort(T[] a, Comparator<? super T> c)
<T> void sort(T[] a, int fromIndex, int toIndex,
Comparator<? super T> c)
```

```
public class ArraysSortTest {
 public static void main(String[] args) {
 int array[] = {4, 7, 2, 3, 9, 7, 6};
 Arrays.sort(array);
 for (int i = 0; i < array.length; i++) {
 System.out.print(array[i] + " ");
 }
 }
}
```

# Klase Arrays

- pretraživanje nizova

```
int binarySearch(byte[] a, byte key) // 1
int binarySearch(byte[] a, int fromIndex, int toIndex, byte
key) // 2
<T> int binarySearch(T[] a, T key, Comparator<? super T> c)
 // 3
<T> int binarySearch(T[] a, int fromIndex, int toIndex, T
key, Comparator<? super T> c) // 4

public class ArraysSearchTest {
 public static void main(String[] args) {
 int array[] = {4, 7, 2, 3, 9, 7, 6};
 Arrays.sort(array);
 for (int i = 0; i < array.length; i++)
 System.out.print(array[i] + " ");
 System.out.println();
 System.out.println("Pretraga 3: " +
Arrays.binarySearch(array, 3));
 System.out.println("Pretraga 5: " +
Arrays.binarySearch(array, 5));
 }
}
```

# Poredak objekata

- Comparable interfejs

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

- compareTo metoda poredi primljeni objekat sa tekućim i vraća negativan integer, 0 ili pozitivan integer, zavisno od toga da li je primljeni objekat manji, jednak ili veći od tekućeg objekta
- ako se objekti ne mogu porebiti metoda će baciti ClassCastException izuzetak

# Poredak objekata

- Comparator interfejs

```
public interface Comparator<T> {
 int compare(T o1, T o2);
}
```

- compare metoda poređi dva argumenta i vraća negativan integer, 0 ili pozitivan integer, zavisno od toga da li je prvi objekat manji, jednak ili veći od drugog objekta
- ako bilo koji od argumenata ima neodgovarajući tip compare metoda bacava ClassCastException

# Bitno

- za Set interfejs – HashSet je najčešće korišćena implementacija
- za List interfejs – ArrayList je najčešće korišćena implementacija
- za Map interfejs – HashMap je najčešće korišćena implementacija
- za Queue interfejs – LinkedList je najčešće korišćena implementacija
- svaka od implementacija obezbjeđuje sve opcione metode iz odgovarajućeg interfejsa