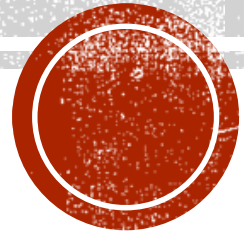
ОБЈЕКТНО ОРИЈЕНТИСАНО ПРОГРАМИРАЊЕ
ПРОГРАМСКИ ЈЕЗИК ЈАВА – 1

Колекције и речници



КОЛЕКЦИЈЕ

- Колекција представља набројиву групу објеката над којом је могуће вршити операције писања, читања, измене или сумаризације (агрегирања).
- На пример, може бити: колекција слова у неком језику, бројева телефона, боја и слично.
- Колекција може имати додатне карактеристике попут уређености, непостојања дупликата итд.
- Јава је на почетку испоручивана са свега неколико класа за рад са колекцијама података попут **Vector**, **Stack**, **Hashtable**, **BitSet** и интерфејсом **Enumeration**.
- У даљем развоју Јаве је требало помирити супротстављене захтеве:
  - библиотека за колекције и речнике треба да буде мала и да се лако може користити;
  - да не буде сложена као што је STL код C++, али да омогући рад са генеричким алгоритмима на начин сличан оном који се користи у STL-у;
  - било је потребно да се старе, већ испоручене класе, природно уклопе у нови оквир.

ИНТЕРФЕЈС И ИМПЛЕМЕНТАЦИЈА

- Библиотека за Јава колекције и речнике стриктно раздваја интерфејсе од имплементација.
- Начин раздвајања ће бити детаљније приказан на колекцији ред (енг. **queue**) која је корисна када елементе треба обрађивати по редоследу приспећа (енг. **first in, first out — FIFO**).

ПРИМЕР 1

- Дефинисати генерички интерфејс за ред који чине методи за додавање елемента на крај ред, узимање са почетка и добијање информације о величини реда.
- Након тога, два пута имплементирати уведени интерфејс за генеричке типове података: употребом кружног низа и повезане листе, као унутрашњих структура података.

ПРИМЕР 1 (2)

```
package rs.math.oop.g14.p01.kolekcijeRed;
/**
*  Поједностављена верзија генеричког реда из стандардне библиотеке
* @param <T> - параметар типа без ограничења
*/
interface Red<T>{
    void dodaj(T element);
    T ukloni();
    int velicina();
}

public class RedPrekoKruznogNiza<T> implements Red<T> {
    private T[] redNiz;
    private int kapacitet;
    private int velicina;
    private int pocetakIndeks;
    private int naredniIndeks;

    …
```

- Остатак програма погледати у књизи.

КОЛЕКЦИЈЕ И ИТЕРАТОРИ

- Најважније могућности у раду са колекцијама су:
  - додавање елемента;
  - уклањање елемента;
  - испитивање да ли колекција садржи елемент;
  - испитивање величине колекције и
  - набрајање елемената.
- За потребе набрајања елемената, дизајнери колекцијске библиотеке су морали да осмисле општи и проширив механизам.
- Приметимо да набројивост не говори ништа о уређењу, тј. не прецизира редослед елемената у оквиру колекције.

ИНТЕРФЕЙС COLLECTION

```
public interface Collection<T>{
  boolean add(T element);
  Iterator<T> iterator();
  boolean contains(Object o);
  boolean remove(Object o);
  int size();
  ...
}
```

ИНТЕРФЕЈСИ ITERABLE И ITERATOR

- Концепт употребе итератора, односно набројивости, није карактеристичан само за колекције. Стога је та могућност издвојена у интерфејс под називом **Iterable.**
- Најзначајнији метод у интерфејсу **Iterable** је **iterator().**

```
public interface Iterable<T>{
  Iterator<T> iterator();
  ...
}
```

- Интерфејс Iterator садржи четири метода,
  при чему су нам за даље разумевање рада итератора значајна прва три.

```
public interface Iterator<T>{
  T next();
  boolean hasNext();
  void remove();
  void forEachRemaining(Consumer<? super T> action);
}
```

ПРИМЕР 2

- Демонстрирати употребу итератора над произвољном генеричком колекцијом.
- Урадити исто и помоћу колекцијске наредбе **for**.

ПРИМЕР 2 (2)

```
public class IteratorPovezanaLista {

    public static void main(String[] args) {
        // уграђена повезана листа, већ смо видели како се може реализовати
        Collection<String> kolekcija = new LinkedList<String>();
        kolekcija.add("Марко");
        kolekcija.add("Марија");
        kolekcija.add("Ивана");
        kolekcija.add("Дарко");
        Iterator<String> iterator = kolekcija.iterator();
        while(iterator.hasNext()) {
            String element = iterator.next();
            System.out.println(element);
        }
        for(String element : kolekcija)
            System.out.println(element);
    }

}
```

ОПЕРАЦИЈЕ НАД КОЛЕКЦИЈОМ КОРИШЋЕЊЕМ ИТЕРАТОРА

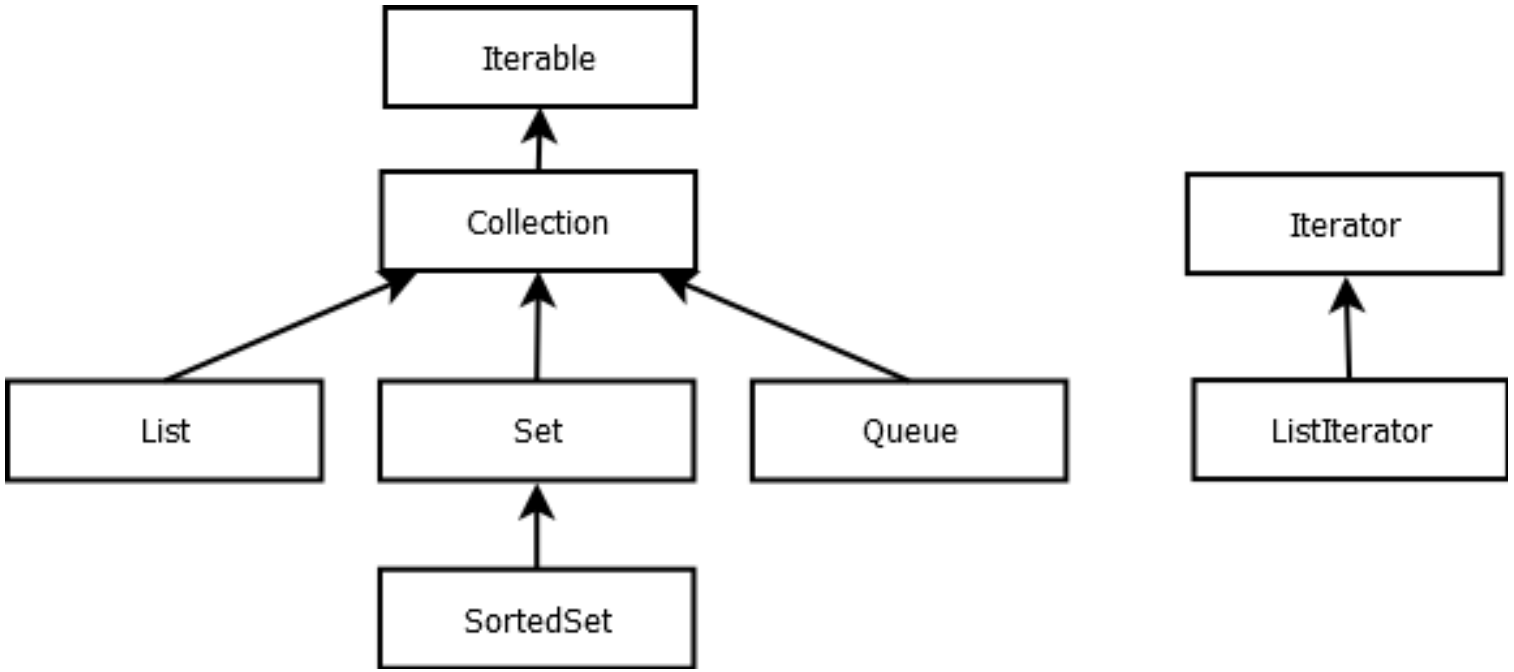
- Позиција итератора је у сваком моменту “између” елемената колекције.

```
                Елемент(0)   Елемент(1)   Елемент(2)   ...  Елемент(n-1)
позиције     ^            ^            ^                 ^                ^
```

- За n елемената постоји n+1 позиција итератора.
- При позиву метода **next()**, итератор прескаче следећи елемент и враћа референцу на елемент који је управо прескочио.
- Метод **remove()** уклања елемент враћен последњим позивом метода **next()**.

```
Iterator String it = c.iterator();
it.next(); // прескочи се први елемент
it.remove(); // сада се уклони
```

КОЛЕКЦИЈСКИ ИНТЕРФЕЈСИ



ИНТЕРФЕЈС LIST

- Листа је уређена колекција, позната и под називом секвенца.
- За разлику од општијег интерфејса **Collection**, корисник има контролу над позицијом сваког елемента.

```
public interface List<T>{
  boolean add(T element);
  void add(int index, T element);
  T get(int index);
  ListIterator<T> listIterator();
  ...
}
```

- За разлику од интерфејса **Collection**, **List** интерфејс обезбеђује и приступ специфичном типу итератора **ListIterator.**

ИНТЕРФЕЈС LISTITERATOR

- Овај итератор наслеђује интерфејс **Iterator.**
  - И притом захтева од класе која га имплементира да буде свесна не само које елементе садржи, већ и њихових позиција.

```
public interface ListIterator<T> extends Iterator<T>{
  void add(T element); // додавање испред позиције итератора
  boolean hasPrevious(); // true ако постоји још неки елемент до краја
  int nextIndex(); // враћа индекс који ће бити враћен наредним next()
  T previous(); // враћа претходни елемент и помера итератор уназад
  int previousIndex(); // попут nextIndex() методе
  void set(T element); // мења последњи елемент добијен next() или previous()
}
```

- Методи листе за приступање и додавање елемената (**get()** и **add()**) засновани су на **ListIterator**-у.
  - Он омогућава позиционирање,
    након чега следи додавање или читање вредности елемента на датој позицији.

ИНТЕРФЕЈСИ QUEUE И DEQUEUE

- Редови омогућавају додавање елемената на крај и уклањање елемената са почетка.
  - Додавање елемената у средину није подржано.
- Овде се мисли на редове који се понашају по фер принципима, тзв. **FIFO** редови (енг. **first in, first out).**

```
public interface Queue<T>{
  boolean add(T element);
  boolean offer(T element);
  T remove();
  T poll();
  T element();
  T peek();
}
```

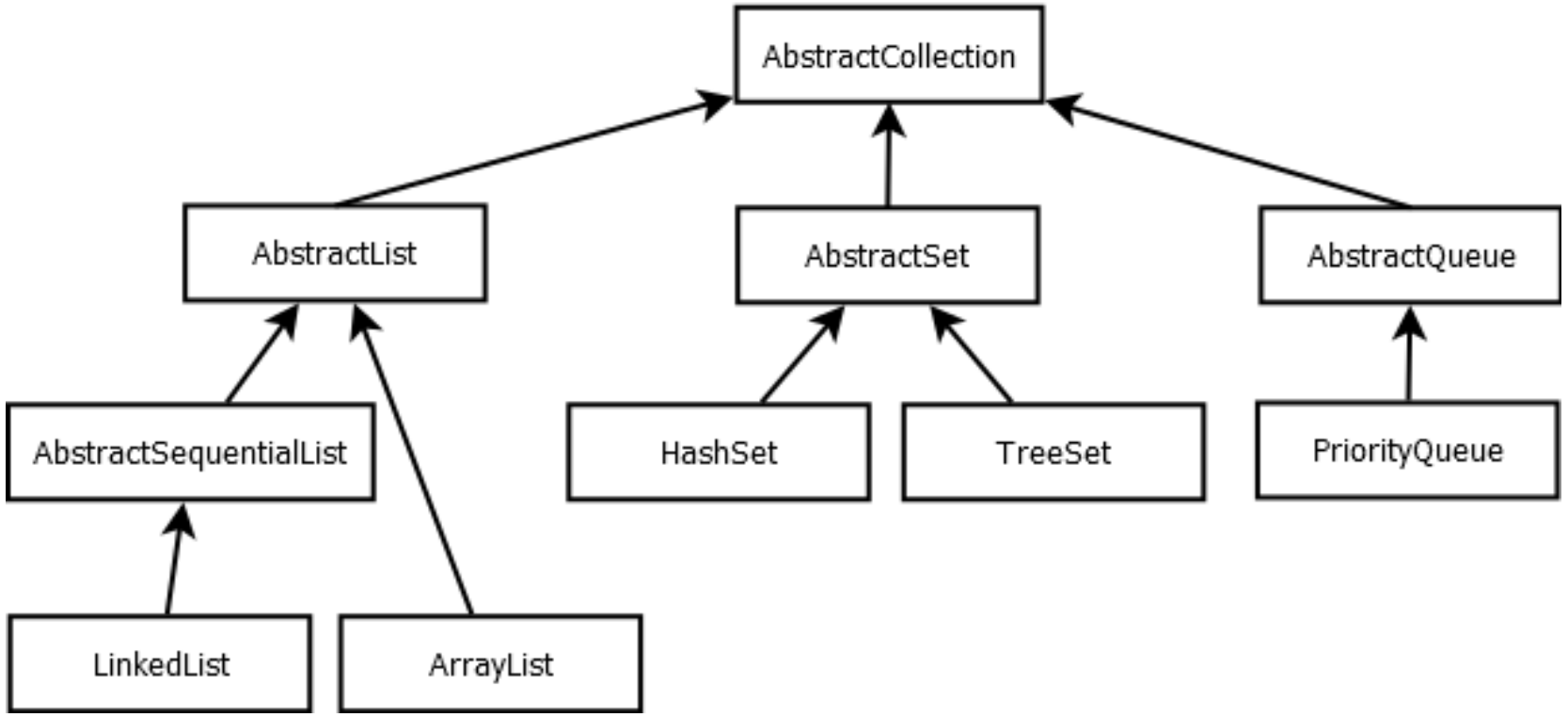
- Ред са два краја, (енг. **deque**), омогућава ефикасно додавање и уклањање елемената и са почетка и са краја
  - Неки методи из **Queue** сада имају и суфикс **First** и **Last.**

ИНТЕФЕЈС SET

- Јава скупови, попут оних у математици, не смеју да садрже дупликате.
  - Методи наслеђени из интерфејса **Collection**, овде прописују специфичније понашање.
- Методи за додавање елемента у скуп и проверу да ли елемент постоји у истом, морају бити свесни забране постојања дупликата.
- На пример, покушај убацивања елемента у скуп који га већ садржи резултира непромењеним скупом.
- У наставку је приказан списак неких релевантнијих метода интерфејса **Set**.

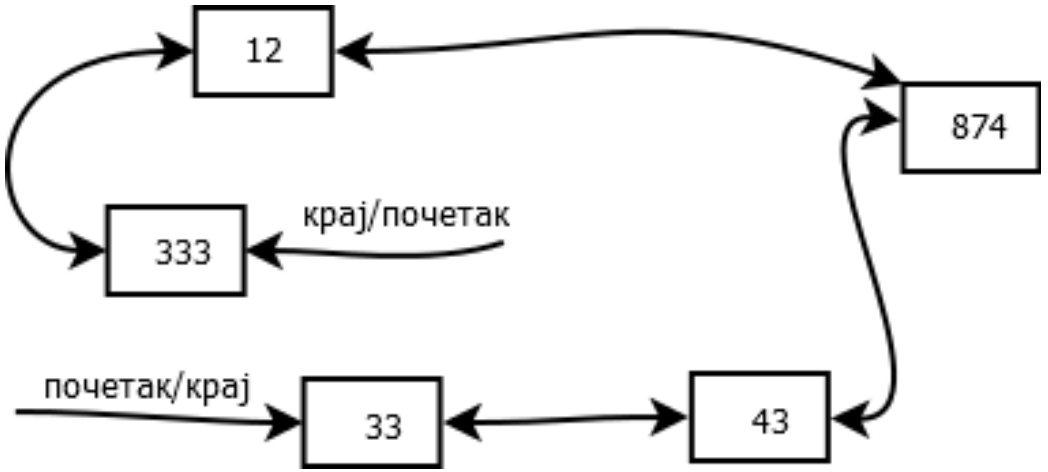
```
public interface Set<T>{
  boolean add(T element); // додавање, ако елемент већ није унутра
  boolean addAll(Collection<? extends T>c); // попут уније, с не мора бити скуп
  void clear(); // формира празан скуп
  boolean equals(); // поређење скупова, небитан редослед
  boolean remove(Object o); // уклањање, ако equals за неки елемент врати true
  boolean removeAll(Collection<?> c);
  boolean retainAll(Collection<?> c);
}
```

КОЛЕКЦИЈСКЕ КЛАСЕ



ДВОСТРУКО ПОВЕЗАНА ЛИСТА LINKEDLIST

- Елементи повезане листе могу бити распоређени на произвољним меморијским локацијама.
- Сваки елемент, поред податка који енкапсулира (нпр. целог броја), има и референце према следећем и претходном елементу.
  - Изузетак су елементи на почетку и крају листе који имају само долазне референце.
- Приступ елементу листе на случајној позицији (енг. random access) има сложеност O(n).
- Сложеност уклањања или додавања са почетка или краја је константна.



ПРИМЕР 3

- Демонстрирати рад са класом **LinkedList** — додавање и уклањање елемената, приступ елементима на случајној позицији и набрајање свих елемената листе у оба смера.

ПРИМЕР 3 (2)

```
LinkedList<Integer> povezanaLista = new LinkedList<Integer>();
povezanaLista.addFirst(33); // [33]
povezanaLista.addLast(12); // [33, 12]
povezanaLista.add(333); // [33, 12, 333] - додаје на крај
povezanaLista.add(1, 43); // [33, 43, 12, 33] - убацује на позицију 1
ListIterator<Integer> iterator = povezanaLista.listIterator();
iterator.next(); // итератор између 33 и 43
iterator.next(); // итератор између 43 и 12
iterator.add(874); //[33, 43, 874, 12, 333] - итератор између 874 и 12
iterator.previous(); // итератор између 43 и 874
iterator.previous(); // итератор између 33 и 43
iterator.remove(); // уклања се следећи тј. 43

System.out.println("Елементи од почетка ка крају:");
for(Integer e : povezanaLista)
    System.out.println(e);

System.out.println("Елементи од краја ка почетку:");
// позиција итератора после последње позиције povezanaLista.size()-1
iterator = povezanaLista.listIterator(povezanaLista.size());
while(iterator.hasPrevious()) // читање уназад
    System.out.println(iterator.previous());

System.out.println("Елементи од почетка ка крају помоћу индекса:");
for(int i=0; i<povezanaLista.size(); i++)
    System.out.println(povezanaLista.get(i)); // неефикасно O(n)
```

НИЗОВНА ЛИСТА ARRAYLIST

- Низовна листа **ArrayList** енкапсулира низ објеката и омогућава манипулацију над њим.
- Највећа погодност за програмера у поређењу са обичним низом је аутоматско проширивање или смањивање капацитета низа (реалокација).
- Временска сложеност стандардних операција је иста као код низа објеката.
  - Због тога је низовна листа бољи избор од повезане листе у ситуацијама када је потребно учестало приступати или мењати вредности елемената на случајним позицијама.
- Карактеристика брзог случајног приступа се обећава имплементацијом интерфејса **RandomAccess**.
  - Овај интерфејс не садржи декларације метода.
  - Његова улога је да укаже на то да класа имплементира брз случајни приступ.
  - Ово надаље може бити корисно генеричким алгоритмима у одлучивању коју класу да примене.
- Са друге стране, ако је потребно често додавати (уметати) елементе, повезана листа је погоднија, јер не захтева померање елемената.
  - Слично важи и за уклањање елемента са неке позиције.

ПРИМЕР 4

- Демонстрирати рад са класом **ArrayList** — додавање и уклањање елемената, приступ елементима на задатој позицији и набрајање свих елемената листе у оба смера.

ПРИМЕР 4 (2)

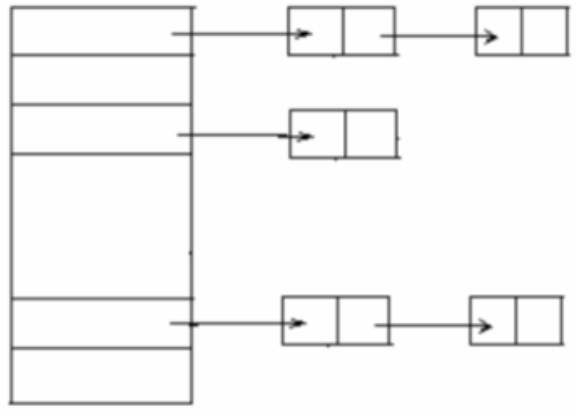
```
ArrayList<Integer> nizovnaLista = new ArrayList<Integer>();
nizovnaLista.add(33); // [33]
nizovnaLista.add(43); // [33, 43]
nizovnaLista.add(12); // [33, 43, 12]
nizovnaLista.add(2, 871); // [33, 43, 871, 12]
nizovnaLista.set(2, 874); // [33, 43, 874, 12]
nizovnaLista.add(333); // [33, 43, 874, 12, 333]
nizovnaLista.remove(1); // [33, 874, 12, 333]

System.out.println("Елементи од почетка ка крају:");
for (Integer e : nizovnaLista)
    System.out.println(e);

System.out.println("Елементи од краја ка почетку:");
// позиција итератора после последње позиције nizovnaLista.size()-1
ListIterator<Integer> iterator;
iterator = nizovnaLista.listIterator(nizovnaLista.size());
while (iterator.hasPrevious()) // читање уназад
    System.out.println(iterator.previous());

System.out.println("Елементи од почетка ка крају помоћу индекса:");
for (int i = 0; i < nizovnaLista.size(); i++)
    System.out.println(nizovnaLista.get(i)); // ефикасно O(1)
```

ХЕШ-СКУП HASHSET

- Ако је потребно пронаћи неки елемент у оквиру низа или листе, потребно је претражити (у најгорем случају) све елементе колекције.
- Уколико је редослед елемената небитан, може се користити колекција која обезбеђује много бржи приступ елементима.
  - Једино је незгодно што се тада не зна ништа о редоследу елемената, већ их та структура организује на произвољан начин.
- Добро позната структура за брзо проналажење елемената је хеш табела.
- За разлику од низова, код којих су индекси цели бројеви, овде је позиција у структури одређена произвољним објектом.
  - Ипак, имплицитно се захтева да сваком објекту буде придружен цели број тзв. хеш-код.

ОТВОРЕНА ХЕШ ТАБЕЛА

- Отворена хеш табела је обично реализована као низ повезаних листи.
- За проналажење места објекта у хеш табели, израчунава се хеш-код и дели се по модулу са бројем елемената тог низа (повезаних листи).
  - Резултат је индекс члана у низу тј. индекс повезане листе која садржи дати елеменат.
- Неизбежно је да се понекад деси да има више елемената којима одговара исти индекс листе и тада долази до тзв. колизије.
  - У том случају нови објекат се пореди са свим објектима из листе (метод equals()) како би се видело да ли је већ присутан у листи.
- Дакле, хеш-код апроксимира идентификацију објекта.
  - Што значи да ће у већини случајева (ако је **hashCode**()добро дефинисан) хеш-код омогућити лоцирање траженог објекта.
  - У преосталим случајевима, када хеш-код није у стању да идентификује тражени објекат, идентификација ће бити извршена применом скупље операције **equals**().



ПРИМЕР 6

- Реализовати генеричку класу за уређени пар уз редефинисање метода **equals**() и **hashCode**().
- Након тога демонстрирати поређење уређених парова и приказати добијене хеш-кодове.
- Као основ за класу користити раније уведену генеричку класу **UredjeniPar** из поглавља 13.

ПРИМЕР 6 (2)

```
public class UredjeniParUporediv<T, S> extends UredjeniPar<T, S>{


    public UredjeniParUporediv(T vrednost1, S vrednost2) { super(vrednost1, vrednost2);}

    @Override
    public boolean equals(Object obj) {
      if(this==obj)
          return true;
      if(obj==null)
          return false;
      if(!(obj instanceof UredjeniParUporediv))
          return false;
      UredjeniParUporediv par =
              (UredjeniParUporediv)obj;
      return getVrednost1().equals(par.getVrednost1()) &&
              getVrednost2().equals(par.getVrednost2());
    }

    @Override
    public int hashCode() {
      int hash = 7;
        hash = 31 * hash
              + (getVrednost1() == null ? 0 : getVrednost1().hashCode());
        hash = 31 * hash
              + (getVrednost2() == null ? 0 : getVrednost2().hashCode());
        return hash;
    }
```

ПРИМЕР 6 (3)

```
    public static void main(String[] args) {
      UredjeniParUporediv<Integer, Double> par1 =
              new UredjeniParUporediv<Integer, Double>(30, 6.74);
      UredjeniParUporediv<Integer, Double> par2 =
              new UredjeniParUporediv<Integer, Double>(30, 6.74);
      UredjeniParUporediv<String, Double> par3 =
              new UredjeniParUporediv<String, Double>("30", 6.74);
      System.out.println("Поредим "+par1+" и "+par1);
      System.out.println(par1.equals(par1));
      System.out.println("Хеш кодови: "+par1.hashCode()
          +" и "+par1.hashCode());
      System.out.println("Поредим "+par1+" и "+par2);
      System.out.println(par1.equals(par2));
      System.out.println("Хеш кодови: "+par1.hashCode()
          +" и "+par2.hashCode());
      System.out.println("Поредим "+par1+" и "+par3);
      System.out.println(par1.equals(par3));
      System.out.println("Хеш кодови: "+par1.hashCode()
          +" и "+par3.hashCode());
    }
}
```

ДРВО-СКУП TREESET

- Дрво-скуп (класа **TreeSet**), за разлику од хеш-скупа, представља уређену колекцију.
- То значи да се елементи у дрво-скуп убацују произвољним редоследом, а да се при пролазу кроз скуп елементи обилазе према неком критеријуму уређења.
- Претпоставимо да се ниске додају у дрво-скуп у следећем редоследу.

```
SortedSet<String> sortiranSkup= new TreeSet<String>();
sortiranSkup.add("Марко");
sortiranSkup.add("Ана");
sortiranSkup.add("Кристина");
for (String s : sortiranSkup)
    System.out.println(s);
```

- Без обзира на редослед уноса, приликом исписа, биће поштовано подразумевано лексикографско растуће уређење над нискама.
  - То значи да ће бити исписана имена у редоследу: Ана, Кристина, Марко.

ОСТАЛЕ ИМПЛЕМЕНТАЦИЈЕ СКУПОВА

- Постоји још неколико уграђених имплементација скупа у Јави: **LinkedHashSet, BitSet, EnumSet** и друге.
- Класа **LinkedHashSet** је уређена варијанта класе **HashSet.**
  - За реализацију ове класе се користи структура хеш-скупа уз додатак двоструко повезане листе која повезује све елементе у редоследу убацивања елемената у скуп.
  - То значи да је предност **LinkedHashSet** могућност набрајања елемената у истом (или обрнутом) редоследу убацивања.
  - Мана је додатни меморијски трошак потребан за одржавање двоструко повезане листе.
- **BitSet** је класа која омогућава рад са скупом ненегативних целих бројева.
  - Ова класа не имплементира интерфејс Set, али је овде описујемо, јер суштински јесте скуп.
  - Присуство елемента (i) у скупу подразумева постојање бита 1 на позицији (i).
  - С обзиром да је **BitSet** реализован помоћу низа, приступ елементу је могућ у времену O(1).

ОСТАЛЕ ИМПЛЕМЕНТАЦИЈЕ СКУПОВА (2)

- Унутрашња реализација класе **BitSet** је меморијски штедљива.
  - Она је заснована на низу речи, при чему је реч подразумевано типа **long** односно дужине 64 бита.
  - Појединачним битовима се приступа помоћу битовских оператора.
    На пример, за представљање скупа од 750 елемената користи се низ од 12 речи.
- **EnumSet** је класа слична **BitSet** с тим што се уместо рада са ненегативним целим бројевима овде ради са енумерисаним типовима.
  - Ограничење је да инстанца **EnumSet** може садржати искључиво вредности јединственог енумерисаног типа.
  - За разлику од **BitSet** класа **EnumSet** је генеричка и имплементира интерфејс **Set**.

НИЗОВНИ РЕД СА ДВА КРАЈА ARRAYDEQUE

- **ArrayDeque** имплементира интерфејс **Deque** употребом низа и помоћних индекса који у сваком моменту одржавају информацију о почетку и крају реда.
- Приликом додавања на крај, индекс краја се увећава, слично се приликом уклањања са краја тај индекс умањује.
  - По истом принципу функционише и одржавање индекса за почетак реда.

ПРИМЕР 12

- Демонстрирати употребу класе **ArrayDeque** у **FIFO** и **LIFO** (енг. **last in - first out**) режиму.

ПРИМЕР 12 (2)

```
Deque<String> red = new ArrayDeque<>();
red.addLast("Петар");
red.addLast("Ана");
red.add("Марко"); // add исто што и addLast
System.out.println(red);
red.removeFirst();
red.remove(); // remove исто што и removeFirst
System.out.println(red);
red.addLast("Милана");
red.addLast("Драган");
System.out.println(red);

Deque<String> stek = new ArrayDeque<>();
stek.add("main");
stek.addLast("fakt(4)");
stek.addLast("fakt(3)");
stek.addLast("fakt(2)");
stek.addLast("fakt(1)");
stek.addLast("fakt(0)");
System.out.println(stek);
stek.removeLast();
stek.removeLast();
System.out.println(stek);
```

РЕД СА ПРИОРИТЕТОМ PRIORITYQUEUE

- Редови са приоритетом враћају елементе у сортираном поретку иако су претходно унесени у произвољном поретку.
  - Прецизније, кад год се позове метод **remove**(), уклања се најмањи елемент реда.
- Редови са приоритетом користе елегантну и ефикасну структуру података, која се зове гомила (енг. heap).
- (Напомена: ову структуру података не треба поистовећивати са хип меморијом, немају никакве сличности, што на пример није случај када о стеку причамо из перспективе структуре података и сегмента меморије, ту постоји велика сличност.)
- Гомила је самоорганизовано бинарно дрво у којем су операције додавања и уклањања реализоване тако да се након сваке измене у корену опет нађе најмањи.
  - Притом су померања елемената унутар дрвета, потребна да би се ово својство одржавало, ефикасна и изискују O(nlog(n)) корака.
- Редови са приоритетом могу да чувају елементе класа које су упоредиве.

ПРИМЕР 13

- Демонстрирати употребу класе **PriorityQueue** у уређивању редоследа рада процеса.
- Процес је описан називом и приоритетом.

ПРИМЕР 13 (2)

```
Queue<Proces> procesi =new PriorityQueue<>();
procesi.add(new Proces("chrome", 10));
procesi.add(new Proces("cmd", 4));
procesi.add(new Proces("taskmgr", 1));
procesi.add(new Proces("explorer", 3));
System.out.println(procesi);
System.out.println(procesi.peek()); // на врху је taskmgr
System.out.println(procesi.poll()); // уклања се taskmgr
System.out.println(procesi);
procesi.add(new Proces("Network service", 2)); // нови најбитнији
System.out.println(procesi);
```

РЕЧНИЦИ

- Често је потребно на основу тзв. кључне информације неког објекта приступити осталим информацијама тог објекта.
- На пример, програмер може учитавати информације о особама из датотеке или са конзоле, и чувати их у меморији.
  - Касније, када је потребно приступити информацијама конкретне особе, може се користити идентификатор попут јединственог матичног броја грађанина (ЈМБГ).
  - Остале информације су одређене ЈМБГ-ом особе.
- У Јави и уопштено ООП терминологији може се говорити о пресликавању скупа кључева (скуп ЈМБГ вредности) у скуп вредности (нпр. име, презиме, година рођења итд.).
- Структуре података које омогућавају оваква пресликавања се називају речници.

РЕЧНИЦИ (2)

- Речник се састоји из више погледа (подструктура). Постоје три погледа:
  1. скуп кључева,
  2. колекција вредности;
  3. скуп парова кључ/вредност.
- Ова три претходно побројана погледа обезбеђују следећи методи.

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K,V>> entrySet();
```

- Колекције кључева и кључ/вредност су увек скупови, јер су кључеви јединствени.
- Интерфејс **Map.Entry<K, V>** описује уређени пар кључ/вредност и обезбеђује методе који враћају кључ и вредност.

ИНТЕРФЕЈС MAP

- Рад са речницима је дефинисан интерфејсом **Map.**
  - Постоји и подинтерфејс **SortedMap** чији је скуп кључева сортиран **SortedSet.**

```
public interface Map<K, V> {
    V get(K kljuc); // враћа вредност кључа или null ако кључ не постоји
    V put(K kljuc, V vrednost); // убацује пар кључ/вредност или ажурира
    V remove(Object kljuc); // уклања кључ/вредност или враћа null

    boolean containsKey(Object kljuc); // да ли постоји кључ
    boolean containsValue(Object value); // да ли постоји бар једна вредност
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
    ...
}
```

ИНТЕРФЕЈС SORTEDMAP

- Интерфејс **SortedMap** наслеђује **Map** и притом додаје још неколико метода.

```
public interface SortedMap<K, V> extends Map<K, V> {
    Comparator<? super K> comparator();
      /* враћа Comparator објекат који се користи за поређење или null ако се
           користи природно поређење (Comparable) */
    SortedMap<K, V> subMap(K kljucOd, K kljucDo); // подмапа за опсег кључева
    K firstKey(); // први кључ по уређењу
    K lastKey(); // последњи кључ по уређењу
    ...
}
```

КЛАСЕ ЗА РЕЧНИКЕ

- Јава библиотека подржава две главне имплементације речника: **HashMap** и **TreeMap.**
- Обе класе индиректно имплементирају **Map** интерфејс тако што наслеђују класу **AbstractMap.**
- Хеш речник не сортира кључеве, за разлику од дрвоидног речника.
- Скуп кључева хеш-речника је заснован на хеш-скупу.
  - Будући да је приступ елементима речника заснован на кључевима, временске сложености убацивања, проверавања постојања и уклањања елемената из хеш-речника су исто O(1).
- Скуп кључева дрво-речника је заснован на дрво-скупу.
  - Слично као и код хеш-речника, и овде су временске сложености свих операција зависне од имплементације скупа кључева. То значи да се операције убацивања, проверавања постојања и уклањања врше у O(log(n)).
- Да ли користити хеш-речник или дрвоидни речник?
  - Као и са скуповима, хеширање је нешто брже, и то је бољи избор уколико кључеви не морају бити сортирани.

ПРИМЕР 14

- Демонстрирати употребу класе **HashMap** у одржавању информација о особама.
- Особа се описује ЈМБГ-ом, именом, презименом и годином рођења.
- Потребно је омогућити претрагу скупа особа на основу ЈМБГ-а.

ПРИМЕР 14 (2)

```
 Osoba[] osobe = new Osoba[] {
    // ЈМБГ очигледно нису валидним, већ измишљени
    new Osoba("1009987567890", "Марко", "Петровић", 1987),
    new Osoba("2001967567890", "Ана", "Ковачевић", 1967),
    new Osoba("1009997567890", "Марија", "Мирковић", 1997),
    new Osoba("0302000567890", "Јована", "Драшковић", 2000),
    new Osoba("1504981567890", "Петар", "Марковић", 1981),
    // исти ЈМБГ - замениће претходни унос (Петар Марковић)
    new Osoba("1504981567890", "Марко", "Марковић", 1981),
};
Map<String, Osoba> osobeKatalog = new HashMap<String, Osoba>();
for(Osoba o : osobe)
    osobeKatalog.put(o.getJMBG(), o);


// редослед при испису није исти као при уносу
for(String jmbg: osobeKatalog.keySet())
    System.out.println(String.format("%s\t->\t%s",
            jmbg, osobeKatalog.get(jmbg)));
```

ПРИМЕР 15

- Пребројати појављивања речи у задатом тексту.
- Након тога исписати на конзоли првих 10 речи и њихових појављивања у складу са лексикографским уређењем речи.
- За реализацију користити уграђену класу **TreeMap**.
- Игнорисати знаке интерпункције и величину слова.

ПРИМЕР 15 (2)

```
String tekst = "Електрични аутомобили у свету нису више реткост "
        + "већ редовна појава на улицама. Тај тренд би требало "
        + "да расте, што показује и недавна најава великог Форда "
        + "да ће у Европи сви његови аутомобили бити електрични "
        + ...;
char[] interpunkcija = new char[] { '.', ',', ';', ':', '?', '!' };
for (char c : interpunkcija)
    tekst = tekst.replace(c, ' ');
String[] reciNeprecisceno = tekst.toLowerCase().split("\s+");
Map<String, Integer> reciPojavljivanja = new TreeMap<>();
for (String r : reciNeprecisceno)
    if (reciPojavljivanja.containsKey(r))
        reciPojavljivanja.put(r, reciPojavljivanja.get(r) + 1);
    else
        reciPojavljivanja.put(r, 1); // прво појављивање
int k = 1;
for (String r : reciPojavljivanja.keySet())
    if (k > 10)
        break;
    else {
        System.out.println(k + ".\t" + r + "\t"
                           + reciPojavljivanja.get(r));
        k++;
    }
```

ГЕНЕРИЦИ И КОЛЕКЦИЈЕ

- Генерички типови су до сада коришћени у комбинацији са низовима.
- Примена и начин рада генеричких типова у контексту колекција је слична.
  - Разлика је у томе што постоји велики број различитих колекција па су генерички колекцијски интерфејси формирани на начин који омогућава уопштавање.
- Нпр. налажење максималног елемента листе може изгледати овако:

```
if(v.size() == 0)
      throw new NoSuchElementException();
T maks = v.get(0);
for (int i = 1; i < v.size(); i++)
      if(maks.compareTo(v.get(i)) < 0)
            maks = v.get(i);
```

За низовну листу.

```
if (l.isEmpty())
      throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T maks = iter.next();
while(iter.hasNext()) {
T sled = iter.next();
      if(maks.compareTo(sled) < 0)
          maks = sled;
}
```

За повезану листу.

ГЕНЕРИЦИ И КОЛЕКЦИЈЕ (2)

- Пожељно је избећи постојање вишеструких дефиниција метода који примењују исти алгоритам над различитим колекцијама:

```
static <T extends Comparable> T max(ArrayList<T> lista);
static <T extends Comparable> T max(LinkedList<T> lista);
```

- Ту на сцену ступају колекцијски интерфејси. Варијанта која ради за било коју колекцију:

```
public static <T extends Comparable> T max(Collection<T> c){
      if (c.isEmpty())
              throw new NoSuchElementException();
      Iterator<T> iter = c.iterator();
      T maks = iter.next();
      while (iter.hasNext()) {
          T sled = iter.next();
          if (maks.compareTo(sled) < 0)
              maks = sled;
      }
      return maks;
}
```

ЏОКЕР ТИП

- Знак питања (?) је у Јава генеричком програмирању познат под називом џокер тип (енг. **wildcard).** Постоје два џокер типа:
  1. џокер тип горње границе (енг. **upper bounded wildcard**)
  2. и џокер тип доње границе (енг. **lower bounded wildcard**).
- Џокер тип горње границе се користи када је потребно релаксирати ограничење над генеричким типом.
  - На пример, ако је потребно дефинисати метод за сумирање који ради над генеричким листама **List<Integer>** и **List<Double>,** декларација метода могла да изгледа овако:

```
public static double suma(List<? extends Number> lista);
```

  - Дакле, џокер се користи у комбинацији са кључном речи **extends** како би се ограничио са горње стране класом **Number**.
- Џокер тип доње границе се користи у ситуацијама када би употреба конкретног генеричког типа била превише рестриктивна.
  - Синтакса је слична као код горње границе, с тим што се, уместо **extends**, користи реч **super**.

ГЕНЕРИЧКИ КОЛЕКЦИЈСКИ МЕТОДИ У ЈДК

- У поглављу посвећеном низовима било је речи о алгоритмима над низовима.
  - Алгоритми над колекцијама су врло слично реализовани, некад и директно сведени на алгоритам који ради над низом.
- Над низовима је било најсмисленије радити све на генерички начин.
- Таква је ситуација и код колекција па се говори о следећим генеричким методима над колекцијама:
  - сортирању,
  - мешању колекције,
  - бинарној претрази
  - итд.

СОРТИРАЊЕ КОЛЕКЦИЈЕ

- Како метод **Collections.sort()** сортира произвољну колекцију?
- Алгоритми за сортирање се у литератури обично дефинишу за низове.
- Кључна погодност низова је брз случајни приступ елементима.
- Међутим, колекција, у општем случају, нема случајан приступ.
- Имплементација у Јави једноставно:
  1. копира све елементе колекције у низ (применом итератора),
  2. сортира га применом метода **Arrays.sort()**,
  3. а затим копира сортирану секвенцу натраг у колекцију.

МЕШАЊЕ КОЛЕКЦИЈЕ

- Класа **Collections** има метод **shuffle()** који случајно пермутује (меша) елементе.
- На пример, следећи код омогућава генерисање случајне пермутације величине 10.

```
List<Integer> permutacija = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
Collections.shuffle(permutacija, new Random(12345));
System.out.println(permutacija); // [4, 3, 1, 6, 9, 10, 7, 8, 5, 2]
```

- Мешање је могуће извршити над било којим објектима, а не само над бројевима.

БИНАРНА ПРЕТРАГА

- Метод **Collections.binarySearch()** имплементира бинарну претрагу.
- Идеја алгоритма је иста као и у случају низова.
- Дакле, колекција мора претходно бити сортирана,
  у супротном ће алгоритам потенцијално вратити погрешан резултат.
- Следе примери позивања овог метода:

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, komparator);
```

- Са **c** је означена колекција у којој се претражује елемент **element.**
- У другом позиву се, уместо природног уређења дефинисаног имплементацијом интерфејса **Comparable**, користи **ad-hoc** уређење дефинисано **komparator** објектом.
- Бинарна претрага захтева од колекције могућност брзог случајног приступа.
  - Стога метод **binarySearch()** проверава да ли колекција имплементира **RandomAccess.**
  - Ако да, онда врши бинарну претрагу, у супротном врши линеарну претрагу.

АПСТРАКТНЕ КЛАСЕ КАО ОСНОВА ЗА КОЛЕКЦИЈЕ

- Када се прегледа **API** документација, уочава се да постоји још један скуп класа, чије име почиње са **Abstract** (на пример класа **AbstractSet**).
- Хијерархија апстрактних колекцијских класа је усаглашена са хијерархијом интерфејса, који су претходно уведени.
- На пример, интерфејс **Collection** има свој пандан **AbstractCollection**, за **Set** постоји **AbstractSet** итд.
- Свака од апстрактних класа имплементира одговарајући интерфејс, па се намеће питање зашто оне уопште постоје?
- Одговор је да ове апстрактне класе поседују имплементације за неке од метода који су прописани интерфејсима.
  - Што надаље омогућава програмеру да, приликом писања својих конкретних класа, не мора да имплементира све методе прописане интерфејсима (којих је обично доста).

АПСТРАКТНЕ КЛАСЕ КАО ОСНОВА ЗА КОЛЕКЦИЈЕ (2)

- Наредни пример демонстрира како се метод **containsAll**(), који је прописан интерфејсом **Collection** реализује позивањем метода **contains**() и итератором.

```
public abstract class AbstractCollection<T>{
  ...
  boolean containsAll(Collection<?> c){
      for (Object o: c)
        if (!contains(o))
            return false;
      return true;
  }
  ...
}
```

- Ако програмер жели директно да имплементира интерфејс **Collection**, потребно је да имплементира 13 метода.
- Ако би се одлучио да, уместо тога, наследи класу **AbstractCollection**, овај број се смањује на свега 2 метода: **iterator**() и **size**().

ПИТАЊА И ЗАДАЦИ

- Које су разлике, а које су сличности између колекције и речника? Како се речник може представити употребом колекција?
- Описати интерфејс **Collection** и навести примере употребе.
- Објаснити везу између интерфејса **Iterable** и интерфејса **Iterator.**
- Написати Јава програм који из колекције уклања све елементе који задовољавају дати услов. На пример, ако је у питању колекција целих бројева уклонити све парне бројеве или све бројеве који се завршавају датом цифром.
- Примерима илустровати ситуације када је корисно користити колекцију:
  - листа;
  - ред;
  - скуп.

ПИТАЊА И ЗАДАЦИ (2)

- Шта су апстрактне колекцијске класе и која је њихова предност у односу на одговарајући интерфејс?
- Објаснити хијерархију колекцијских класа у Јави.
- Упоредити две имплементације **List** интерфејса – **LinkedList** и **ArrayList.** Примером илустровати ситуације када је погодније користи једну, а када другу имплементацију.
- Како је имплементирана и како се користи класа **HashSet?** Зашто је битно да методи **hashCode()** и **equals()** буду конзистентно (ре)дефинисани?
- Како је имплементирана и како се користи класа **TreeSet?** Примером илустровати ситуације када је погодније користи **HashSet** имплементацију, а када **TreeSet** имплементацију скупа.

ПИТАЊА И ЗАДАЦИ (3)

- Које још уграђене имплементације скупа постоје у програмском језику Јава? Да ли све имплементирају интерфејс **Set**?
- Користећи класу **PriorityQueue** имплементирати алгоритам за сортирање који ради по принципу **heap sort**-а.
- Шта је џокер тип и када се користи? Примерима илустровати употребу џокер типа горње границе и џокер типа доње границе.
- Објаснити како се врши сортирање колекције употребом метода **Collections.sort()**.
- Карта је описана знаком (пик, каро, херц, треф) и вредношћу (ас, краљ, дама, жандар и бројчане вредности од 2 до 10). Написати Јава програм који генерише, а затим меша стандардни шпил од 52 и исписује их на екрану.
- Примерима илустровати употребу неких значајнијих метода класе **Collections**.