



Kolekcije

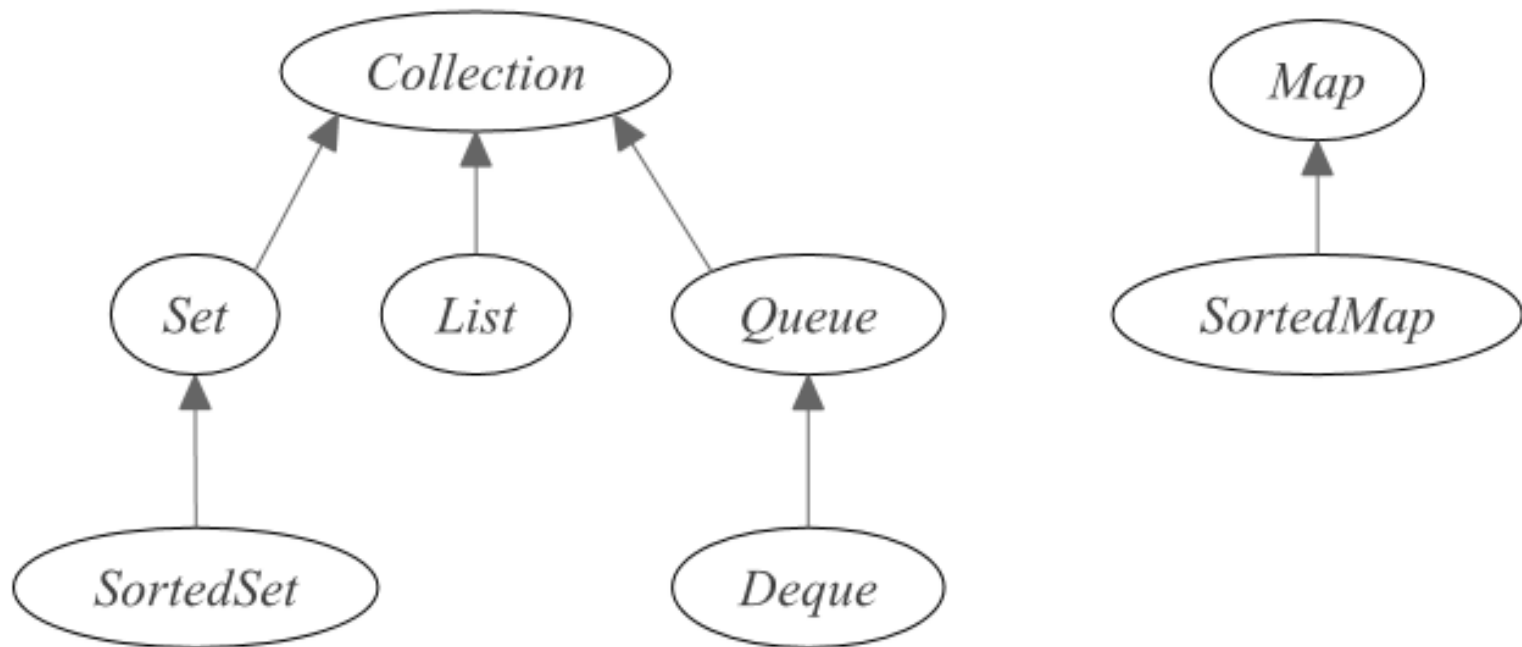


Kolekcije

- Kolekcija je objekat koji sadrži druge objekte.
- U standardnoj Java biblioteci, to mogu biti:
 - **liste,**
 - **skupovi,**
 - **redovi opsluživanja**
 - **i mape.**
- Kolekcije olakšavaju pisanje (naročito velikih) programa u značajnoj meri, jer pružaju gotove, efikasne implementacije bitnih struktura podataka.
- Sve kolekcije su sadržane u paketu **java.util.**

Kolekcije

Svaka od kategorija je predstavljena posebnim interfejsom, čije su hijerarhije date na slici:





Kolekcije

- Da bi se uspešno radilo sa kolekcijama, neophodno je poznavanje datih hijerarhija, kao i generalni opis svakog interfejsa:
 - **List**: Dinamička lista elemenata, može proizvoljno rasti i smanjivati se, shodno broju elemenata.
 - **Set**: Predstavlja skupove. Postoji još jedan interfejs za skupove, **SortedSet**, koji nasleđuje **Set** i sortira elemente po određenom kriterijumu.
 - **Map**: Mapa koja sadrži parove elemenata u obliku ključ → vrednost. Slično kao kod skupova, iz ovog interfejsa je izveden interfejs **SortedMap** koji predstavlja mape sa sortiranim ključevima.
 - **Queue**: Red opsluživanja. Interfejs koji ga nasleđuje, **Deque**, predstavlja redove opsluživanja koji mogu prihvatati i izbacivati elemente sa obe strane.



Kolekcije

- Sve kolekcije sem mapa nasleđuju interfejs **Collection** koji definiše skup zajedničkih metoda.
- Pored ovog interfejsa, postoji i klasa **Collections** koja sadrži veliki broj algoritama za rad sa kolekcijama, u obliku statičkih metoda.
- Na primer, pomoću ove klase moguće je ručno sortirati liste, pronaći najveći/najmanji element u kolekciji, obrnuti ili izmešati sadržaj liste, itd.



Kolekcije

- Metod **add** dodaje element u kolekciju.
- Ovaj metod vraća **true** ako je dodavanje metoda zaista promenilo kolekciju, a vraće **false** ako je kolekcija nepromenjena.
- Na primer, ako se dodaje u skup element koji se već nalazi u tom skupu, tada zahtev za dodavanje nema efekta jer skup odbija duplikate, pa metod **add** vraće false .
- Detaljan spisak i opis metoda je dostupan u zvaničnoj Java dokumentaciji.



Interfejs Collection

- Sledeći primer demonstrira upotrebu osnovnih metoda interfejsa Collection.
- Pošto je u pitanju interfejs, u programu je korišćen objekat jedne konkretne klase koja ga (indirektno) implementira, **ArrayList**.
- Umesto prazne implementacije, standardan pristup u Java API kolekcijama je generisanje izuzetka tipa **UnsupportedOperationException**.

Primer 10.1: Osnovne operacije za rad sa kolekcijama

```
import java.util.ArrayList;
import java.util.Collection;

public class CollectionBasic {
    public static void main(String[] args) {
        // inicijalizacija kolekcije (liste) brojeva
        Collection<Integer> c = new ArrayList<>();
        // dodajemo nekoliko elemenata
        c.add(5);
        c.add(-1);
        c.add(null); // vecina kolekcija moze sadrzati 'null' elemente
        // informacije o kolekciji
        System.out.println("Velicina: " + c.size());
        System.out.println("Da li je prazna? " + c.isEmpty());
        System.out.println("Da li sadrzi broj 6? " + c.contains(6));
        // izbacujemo element
        c.remove(-1);
        System.out.println("Kolekcija je: " + c);
    }
}
```

Velicina: 3
Da li je prazna? false
Da li sadrzi broj 6? false
Kolekcija je: [5, null]

- Kada radimo sa kolekcijama, nakon navođenja tipa kolekcije, a unutar simbola < > navodimo tip elemenata.
- Počev od Java 7 se sa desne strane izraza ne mora ponovo navesti tip, već je dovoljno ostaviti <> (eng. diamond syntax).
- Ako kao tip elemenata navedemo klasu A, tada kolekcija može sadržati objekte klase A, kao i objekte svih klasa koje su (direktno ili indirektno) izvedene iz A.
- Kolekcije mogu raditi samo sa referencijalnim tipovima podataka. Program će ispisati sledeće:



Interfejs Collection

- Generalno, za svaku konkretnu kolekciju postoje tri tipa konstruktora:
 - Bez parametara, koji inicijalizuje praznu kolekciju neke unapred zadate veličine;
 - Konstruktor koji prihvata početnu veličinu kolekcije;
 - Konstruktor koji prihvata kolekciju kao parametar, i kopira njene elemente u kolekciju koja se konstruiše.
- Ukoliko znamo koliko će (otprilike) elemenata biti u kolekciji, najbolje je koristiti drugi konstruktor. Tako će celokupna memorija za kolekciju biti obezbeđena na početku, i program će imati najbolje performanse.

Primer 10.2: Metodi koje prihvataju kolekcije kao parametre

```
public class CollectionBulk {
    public static void main(String[] args) {
        // znamo da cemo ubaciti 4 elementa
        Collection<String> a = new ArrayList<>(4);
        a.add("pera");
        a.add("pera"); // duplikati su dozvoljeni u listama
        a.add("aca");
        a.add("mika");
        // 'b' ce imati iste elemente kao 'a'
        Collection<String> b = new ArrayList<>(a);
        // jos jedna kolekcija
        Collection<String> c = new ArrayList<>();
        c.add("pera");
        c.add("mika");
        // ukloni sve elemente kolekcije 'c' iz 'a'
        a.removeAll(c);
        // u kolekciji 'b' zadrzi samo one elemente koji postoje u 'c'
        b.retainAll(c);
        // da li 'b' sadrzi sve elemente iz 'c'?
        System.out.println("Da li 'b' sadrzi sve iz 'c'? " +
            b.containsAll(c));
        // dodaj celu kolekciju 'c' u 'a', pa obrisi 'c'
        a.addAll(c);
        c.clear();
        // ispis
        System.out.println("Kolekcija a: " + a);
        System.out.println("Kolekcija b: " + b);
        System.out.println("Kolekcija c: " + c);
    }
}
```

Program će na ekranu ispisati sledeće informacije:

```
Da li 'b' sadrzi sve iz 'c'? true
Kolekcija a: [aca, pera, mika]
Kolekcija b: [pera, pera, mika]
Kolekcija c: []
```



```
a = [pera, pera, aca, mika]
b = [pera, pera, aca, mika]
c = [pera, mika]
```

(nakon inicijalizacije)

a.removeAll(c)

```
a = [aca]
b = [pera, pera, aca, mika]
c = [pera, mika]
```

b.retainAll(c)

```
a = [aca, pera, mika]
b = [pera, pera, mika]
c = [pera, mika]
```

a.addAll(c)

```
a = [aca]
b = [pera, pera, mika]
c = [pera, mika]
```

Slika 10.2: Efekat primene metoda removeAll, retainAll i addAll



Kolekcije i nizovi

- Ponekad je potrebno pretvoriti niz u kolekciju, ili kolekciju u niz.
- Prva operacija se obavlja pozivanjem statičkog metoda `asList` klase `java.util.Arrays`, koji prihvata niz i vraća odgovarajuću listu.
- Ovu listu je, na primer, kasnije moguće proslediti konstruktoru skupa.
- Za pretvaranje kolekcije u niz, koristi se generički metod interfejsa `Collection`:

```
<T> T[] toArray(T[] a)
```



Kolekcije i nizovi

Metod funkcioniše po sledećim pravilima:

- Ako cela kolekcija može da stane u niz **a** (tj. niz je dovoljno velik), elementi kolekcije se ubacuju u njega. Ako je dužina niza **n**, a veličina kolekcije **m < n**, elementi niza počev od **a[m]** će biti **null**. Povratna vrednost metoda će biti referenca na **a**.
- Ako je dužina niza **a** manja od veličine kolekcije, metod će alocirati novi niz, elemente kolekcije će ubaciti u njega, i vratiti referencu na taj novi niz. Niz **a** će ostati nepromenjen.
- Ako se tip elemenata kolekcije ne može pretvoriti u tip elemenata niza (npr. kolekcija sadrži brojeve, a niz stringove), biće generisan izuzetak tipa **ArrayStoreException**.

Primer 10.3: Pretvaranje kolekcije u niz

```
public class KolekcijaUNiz {  
    public static void main(String[] args) {  
        Collection<Integer> c = new ArrayList<>();  
        c.add(1);  
        c.add(2);  
        c.add(3);  
        // pretvaranje u niz  
        Integer[] niz = new Integer[c.size()];  
        c.toArray(niz);  
        System.out.println("Niz: " + Arrays.toString(niz));  
    }  
}
```

- Primer upotrebe metoda toArray.
- Primer poziva statički metod Arrays.toString(niz) kako bi pretvorio niz u string.



Prolazak kroz kolekcije

- Postoje dva univerzalna načina za prolazak kroz kolekcije: **for-each** petlja i **iterator**.
- Upotreba **for-each** petlje je identična onoj kod nizova.
- **Iterator** je predstavljen istoimenim interfejsom sa sledećim metodima:
 - **next()**: vraća sledeći element kolekcije;
 - **hasNext()**: vraća true ukoliko postoji još elemenata kolekcije; i
 - **remove()**: uklanja trenutni element iz kolekcije.
- U sledećem primeru je dat program koji ispisuje sve neparne brojeve iz kolekcije, upotrebom for-each petlje i iteratora.

Primer 10.4: Prolasci kroz kolekcije

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class Prolazak {
    public static void main(String[] args) {
        Collection<Integer> c = new ArrayList<>();
        for (int i = 0; i < 10; c.add(++i));
        // for-each petlja
        for (int i : c)
            if (i % 2 != 0)
                System.out.println(i);
        // iterator - inicijalizacija
        Iterator<Integer> i = c.iterator();
        while (i.hasNext()) { // dok ima elemenata...
            int j = i.next(); // uzmi sledeci element
            if (j % 2 != 0)
                System.out.println(j);
        }
    }
}
```

Svaki poziv metode **next()** vraća sledeći element. Ukoliko bismo telo while petlje zapisali u sledećem obliku:

```
if (i.next() != 0)
    System.out.println(i.next());
```

program ne bi funkcionisao korektno, jer bi ispitivao n-ti, a ispisivao (n+1)-vi element.

Štaviše, ukoliko pozovemo metod **next()**, a kolekcija nema više elemenata, biće generisan izuzetak tipa **NoSuchElementException**.



Prolazak kroz kolekcije

- Iteratore je zgodno koristiti kada želimo da izbacujemo elemente dok prolazimo kroz kolekciju.
- Zapravo, ovo je jedini dozvoljeni način.
- Ukoliko pokušamo da izbacimo elemente dok prolazimo kroz for-each petlju, biće generisan izuzetak.
- U narednom primeru je data aplikacija koja iz kolekcije izbacuje sve neparne brojeve.
- Aplikacija vrši tzv. filtriranje podataka, odnosno eliminisanje elemenata koji ne zadovoljavaju određene uslove.

Primer 10.5: Izbacivanje elemenata iz kolekcije prilikom prolaska

```
public class Izbacivanje {  
    public static void main(String[] args) {  
        Collection<Integer> c = new ArrayList<>();  
        for (int i = 0; i < 10; c.add(++i));  
        // iteratore mozemo koristiti i u for petlji  
        for (Iterator<Integer> i = c.iterator(); i.hasNext(); )  
            if (i.next() % 2 != 0)  
                i.remove(); // izbaci trenutni element  
        System.out.println(c);  
    }  
}
```



Rad sa listama

- Liste elemenata su predstavljene interfejsom **List**. Kao što je rečeno, liste su konceptualno slične nizovima, ali nude veću fleksibilnost
- Najvažnije, veličina liste se automatski menja kako dodajemo ili uklanjamo elemente, za razliku od niza čija se veličina nakon inicijalizacije ne može menjati.
- Pored toga, postoji značajniji veći broj metoda za rad sa listama nego nizovima.
- Postoje dve konkretne implementacije listi, koje su predstavljene klasama **ArrayList** i **LinkedList**. Prva implementacija se zasniva na nizu, a druga na dvostruko povezanoj dinamičkoj listi.



Rad sa listama

- `ArrayList` nudi bolje performanse prilikom pristupanja elementima preko indeksa, zatim ako elemente uglavnom dodajemo na kraj i uklanjamo s kraja, a pored toga koristi i manje radne memorije.
- Upotreba klase `LinkedList` se preporučuje samo kada dodajemo ili uklanjamo puno elemenata sa početka liste. Pored toga, ova klasa implementira i interfejs `Deque`, te se može upotrebiti za simulaciju reda opsluživanja i steka.
- Osnovna prednost listi u odnosu na sve druge kolekcije je rad sa celobrojnim indeksima. Tačnije, elementu liste je, slično kao kod nizova, moguće pristupiti preko indeksa, moguće je element ubaciti na tačno određenu poziciju, itd.
- Interfejs `List` tako proširuje Collection metodama koji se zasnivaju na indeksima.

Primer 10.6: Rad sa listama

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Liste {
    public static void main(String[] args) {
        List<Integer> a = new ArrayList<>();
        // dodajemo nekoliko elemenata
        a.add(1); a.add(2); a.add(3); a.add(4);
        // '5' dodajemo na poziciju 1
        a.add(1, 5);
        System.out.println(a);
        // umesto elementa na poziciji 0 ('1') stavimo '3'
        a.set(0, 3);
        System.out.println(a);
        // izbacujemo poslednji element
        a.remove(a.size() - 1);
        // pretrazivanje
        int i = a.indexOf(4);
        if (i == -1)
            System.out.println("Broj 4 ne postoji u listi");
        else
            System.out.println("Prvo pojavljivanje broja 4 je " +
                               "na poziciji " + i);
        i = a.lastIndexOf(3);
        if (i == -1)
            System.out.println("Broj 3 ne postoji u listi");
        else
            System.out.println("Poslednje pojavljivanje broja 3 je " +
                               "na poziciji " + i);
        // sortiranje liste
        Collections.sort(a);
        // izbaci sve sem prvog i poslednjeg elementa
        List<Integer> sub = a.subList(1, a.size() - 1);
        sub.clear();
        System.out.println(a);
    }
}
```

Program će ispisati sledeće:

[1, 5, 2, 3, 4]

[3, 5, 2, 3, 4]

Broj 4 ne postoji u listi

Poslednje pojavljivanje broja 3 je
na poziciji 3

[2, 5]



Rad sa listama

- Prethodni program će ispisati sledeće:

```
Program će ispisati sledeće:
```

```
[1, 5, 2, 3, 4]
```

```
[3, 5, 2, 3, 4]
```

```
Broj 4 ne postoji u listi
```

```
Poslednje pojavljivanje broja 3 je na poziciji 3
```

```
[2, 5]
```

- Dakle, kao i kod nizova, indeksi liste počinju od 0, dok poslednji element liste dužine n ima indeks n-1.
- Metodi `indexOf` i `lastIndexOf` vraćaju indekse, redom, prvog i poslednjeg pojavljivanja prosleđenog elementa, odnosno -1 ukoliko element ne postoji u listi.



Rad sa listama

- Metod `subList` prihvata dva parametra, `from` i `to`, i vraća pod-listu koja sadrži elemente osnovne liste u intervalu `[from, to)`.
- Ovde je bitno napomenuti da je povratna pod-lista samo referenca na osnovnu listu: ako menjamo nju, automatski ćemo menjati i osnovnu listu.
- Program koristi ovu osobinu da bi iz osnovne liste `a` izbacio sve sem prvog i poslednjeg elementa.
- Prilikom rada sa listama neophodno je voditi računa da indeksi koji se prosleđuju metodama budu u odgovarajućem intervalu. Ako je, na primer, indeks prosleđen metodi set van intervala `[0, size() - 1]`, biće generisan izuzetak.



Rad sa listama

- Pored for-each petlje i iteratora, kroz listu možemo proći i upotrebom objekta tipa `ListIterator`.
- Ovaj tip proširuje običan iterator opisan ranije, nudeći dodatne funkcionalnosti: kretanje unazad, dobijanje indeksa sledećeg ili prethodnog elementa, zamenu trenutnog elementa novim, itd.
- Detaljniji opis ovih funkcionalnosti je moguće pronaći u zvaničnoj dokumentaciji interfejsa `ListIterator`.



Rad sa skupovima

- Skupovi su strukture podataka koje ne mogu sadržati duplikate, i predstavljeni su interfejsima `Set` i `SortedSet`.
- `Set` sadrži samo metode iz interfejsa `Collection`, dok `SortedSet` sadrži nekoliko novih metoda za rad sa sortiranim skupovima.
- Postoje tri implementacije skupova, koje su predstavljene klasama `HashSet`, `LinkedHashSet` i `TreeSet`.
- Prve dve klase implementiraju interfejs `Set`, dok treća implementira `SortedSet`. Generalno govoreći, `HashSet` nudi najbolje performanse u toku izvršavanja, ali elemente čuva u proizvoljnom redosledu.
- `LinkedHashSet` je nešto sporija implementacija, ali elemente čuva u redosledu u kom su ubačeni u skup.
- Konačno, `TreeSet` je najsporija, ali sortira elemente po nekom kriterijumu. Razlika funkcionalnosti ove tri implementacije je data u sledećem primeru.

Primer 10.7: Ponašanje različitih implementacija skupova

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class Skupovi {
    private static void dodajIspisi(Set<String> skup) {
        skup.add("jedan");
        skup.add("dva");
        skup.add("tri");
        skup.add("cetiri");
        System.out.println(skup.getClass().getSimpleName() + ":\t" + skup);
    }

    public static void main(String[] args) {
        dodajIspisi(new HashSet<String>());
        dodajIspisi(new LinkedHashSet<String>());
        dodajIspisi(new TreeSet<String>());
    }
}
```

- Svaki skup će imati tačno četiri elementa: drugo ubacivanje stringa „tri“ neće uspeti, jer on već postoji u skupu. Metod add vraća true ako je ubacivanje uspelo, inače false. Rezultat izvršavanja programa je:

```
HashSet:      [jedan, tri, cetiri, dva]
LinkedHashSet: [jedan, dva, tri, cetiri]
TreeSet:      [cetiri, dva, jedan, tri]
```

- Dakle, ako nam redosled elemenata u skupu nije bitan, najefikasnije je koristiti klasu HashSet. U osnovnim podešavanjima TreeSet sortira stringove po rastućem redosledu.

Mape

- Mape se koriste za čuvanje parova elemenata, koji se nazivaju ključ i vrednost.
- Pri tome, ne mogu postojati dva ista ključa, dok se vrednosti ignorišu. Tačnije, vrednosti mogu biti null i mogu se ponavljati, dok se velika većina operacija, poput sortiranja i pretraživanja, vrši samo preko ključa. Takođe, vrednost je iz mape moguće uzeti jedino preko ključa.
- Elementi mape su zapravo predstavljeni unutrašnjim interfejsom Map.Entry, čija konkretna implementacija sadrži ključ i vrednost.
- Na primer, neka je data mapa koja sadrži cene voća na pijaci. Ključevi su nazivi voća (npr. jabuka, kruška i banana), dok su vrednosti cene (npr. 50, 60 i 80 din/kg. redom).

<i>ključevi</i> →	jabuka	kruska	banana
<i>vrednosti</i> →	50	60	80

Entry



Mape

- Postoje tri implementacije mapa, koje su predstavljene klasama:
 - `HashMap`,
 - `LinkedHashMap` i
 - `TreeMap`.
- Diskusija o performansama i funkcionalnostima je slična onoj kod skupova, s tim što, ponovo, mape posmatraju samo ključeve.
- Ne postoji ugrađeni način da, na primer, sortiramo cene voća koristeći `TreeMap`.
- U nastavku je dat primer koji demonstrira najvažnije operacije za rad sa mapama.

Primer 10.8: Najvažnije operacije za rad sa mapama

```
public class Pijaca {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        // cene voca, kljuc (naziv voca) je tipa String,
        // vrednost (cena po kg) je Integer
        Map<String, Integer> cene = new HashMap<>();
        // popunjavamo mapu nekim vrednostima
        cene.put("jabuka", 50);
        cene.put("kruska", 60);
        cene.put("banana", 80);

        System.out.print("Unesite novu cenu za jabuku: ");
        int nova = Integer.parseInt(in.readLine());
        cene.put("jabuka", nova);

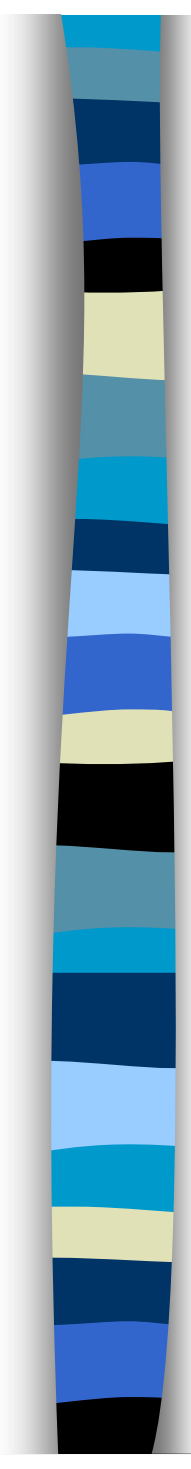
        System.out.print("Unesite naziv voca da biste saznali cenu: ");
        String naziv = in.readLine();
        Integer cena = cene.get(naziv); // vrati vrednost za ovaj kljuc
        if (cena == null)
            System.out.println("Ne postoji trazeno voce");
        else
            System.out.println("Cena je: " + cena + " din/kg");
    }
}
```

Ukoliko bismo zapisali `int cena = cene.get(naziv)`, indirektno bismo vršili pretvaranje `Integer` vrednosti u `int`, jer su vrednosti mape referencijalnog tipa. Međutim, ukoliko ključ ne postoji, pokušali bismo pretvoriti `null` u `int`, što bi dovelo do generisanja izuzetka. Ovo je jedna od čestih i teško uočljivih početničkih grešaka.



Mape

- Podatke iz mape možemo dobiti na više načina. Jedan je pozivom metoda `get` kao u prethodnom primeru. Pored toga, interfejs `Map` sadrži još nekoliko korisnih metoda:
 - `keySet()` : vraća skup svih ključeva. Povratna vrednost je tipa `Set`, jer ključevi moraju biti različiti;
 - `values()` : vraća kolekciju svih vrednosti. Povratna vrednost je tipa `Collection`; i
 - `entrySet()` : vraća skup svih `Entry` objekata.
- Pošto sva tri metoda vraćaju kolekcije, kroz njih možemo prolaziti na načine opisane ranije: pomoću `for-each` petlje i iteratora.
- U narednom primeru je dat metod `ispisiNajskuplje` koji se nadovezuje na prethodni program i ispisuje naziv najskupljeg voća.



Primer 10.9: Prolazak kroz elemente mape uputrebom iteratora

```
private static void ispisiNajskuplje(Map<String, Integer> cene) {  
    Entry<String, Integer> najskuplje = null;  
    // svi podaci mape  
    Set<Entry<String, Integer>> podaci = cene.entrySet();  
    // iterator za prolazak kroz podatke  
    Iterator<Entry<String, Integer>> i = podaci.iterator();  
    while (i.hasNext()) {  
        Entry<String, Integer> voce = i.next();  
        if (najskuplje == null || najskuplje.getValue() < voce.getValue())  
            najskuplje = voce;  
    }  
    System.out.println("Najskuplje voce je " + najskuplje.getKey());  
}
```



Red opsluživanja

- Red opsluživanja je predstavljen interfejsima **Queue** i **Deque**.
- Kao što je rečeno, **Queue** predstavlja standardni red opsluživanja: elementi se dodaju na kraj, a uzimaju sa početka.
- **Deque** (skraćeno od eng. *double-ended queue* dvostrani red) nudi mogućnost dodavanja i uklanjanja elemenata i s kraja i sa početka.
- Postoje dve osnovne implementacije reda opsluživanja: **LinkedList** opisana ranije, i **PriorityQueue** koja automatski sortira elemente.



Red ospluživanja

- Bitno je napomenuti da, iako su elementi sortirani, prolazak kroz `PriorityQueue` upotrebom iteratora ne garantuje redosled elemenata.
- Za dodavanja i uklanjanje elemenata interfejs `Queue` nudi po dva metoda.
- Razlika je u načinu obrade grešaka: jedan generiše izuzetak, dok drugi vraća unapred određenu vrednost (null ili false, u zavisnosti od operacije).
- Pregled metoda je dat u sledećoj tabeli.

	<i>Throws exception</i>	<i>Returns special value</i>
Insert	<code>add (e)</code>	<code>offer (e)</code>
Remove	<code>remove ()</code>	<code>poll ()</code>
Examine	<code>element ()</code>	<code>peek ()</code>



Red ospluživanja

- Pri tome, greška „Red je pun“ se ne može javiti kod pomenute dve implementacije, već samo kod nekih specijalizovanih.
- Analogan skup metoda postoji i u interfejsu Deque, sa sufiksima First i Last koji označavaju da metod radi sa početkom, odnosno krajem kolekcije.
- S obzirom na to da klasa LinkedList implementira interfejs Deque (i interfejs List), možemo je koristiti i za simulaciju steka i za simulaciju reda opsluživanja. U narednom primeru je dat program koji koristi **LinkedList** kao stek i **PriorityQueue** kao red opsluživanja.

Primer 10.10: Rad sa stekom i redom opsluživanja

```
import java.util.Deque;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class StekIRed {
    public static void main(String[] args) {
        Deque<Integer> stek = new LinkedList<>();
        stek.addFirst(1);
        stek.addFirst(2);

        System.out.print(stek.removeFirst() + " ");
        System.out.println();

        Queue<Integer> red = new PriorityQueue<>();
        red.add(1);
        red.add(2);
        red.add(3);
        while (!red.isEmpty())
            System.out.print(red.remove() + " ");
    }
}
```



Jednakost i sortiranje elemenata

- Veoma bitni koncepti.
- U primeru smo videli da ne možemo ubaciti dva ista stringa u skup, ali šta ako pokušamo da ubacimo objekte klase koju smo mi napisali?
- Šta ako stringove želimo da sortiramo opadajuće, a ne rastuće kako to čini kolekcija `TreeSet`?
- U nastavku je data klasa `Osoba` koja sadrži ime i prezime, kao i odgovarajući konstruktor i get metode. Ovu klasu ćemo koristiti u primerima koji slede.

Primer 10.11: Implementacija klase `Osoba`

```
public class Osoba {  
    private String ime;  
    private String prezime;  
  
    public Osoba(String ime, String prezime) {  
        this.ime = ime;  
        this.prezime = prezime;  
    }  
  
    public String getIme() {  
        return ime;  
    }  
  
    public String getPrezime() {  
        return prezime;  
    }  
}
```



Jednakost elemenata

- U narednom primeru je dat program koji u skup ubacuje dve osobe sa istim imenom i prezimenom. Broj elemenata skupa će na kraju programa biti 2, jer kolekcija nema način da utvrdi jednakost osoba.

Primer 10.12: Prvi pokušaj ubacivanja jednakih objekata u skup

```
public class Ubacivanje {  
    public static void main(String[] args) {  
        Osoba a = new Osoba("Petar", "Petrovic");  
        Osoba b = new Osoba("Petar", "Petrovic");  
        Set<Osoba> osobe = new HashSet<>();  
        osobe.add(a);  
        osobe.add(b);  
        System.out.println("Broj elemenata skupa: " + osobe.size());  
    }  
}
```

- Bilo bi pogrešno da kolekcija automatski poredi sva polja objekta.
- Šta ako, na primer, Osoba sadrži i jedinstveni matični broj, u kom slučaju ne treba porediti imena?
- Da bi se rešio ovaj problem, klasa Object sadrži metod **equals** koji se poziva kad god je potrebno utvrditi jednakost objekata.

Jednakost elemenata

- U osnovnoj implementaciji, tj. u implementaciji koja je data u klasi `Object`, metod **equals** jednostavno poredi reference.
- Zbog toga je potrebno proširiti klasu `Osoba` tako da sadrži i implementaciju ovog metoda.

Primer 10.13: Implementacija metoda `equals` u klasi `Osoba`

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Osoba other = (Osoba) obj;
    if (ime == null) {
        if (other.ime != null)
            return false;
    }
    else if (!ime.equals(other.ime))
        return false;
    if (prezime == null) {
        if (other.prezime != null)
            return false;
    }
    else if (!prezime.equals(other.prezime))
        return false;
    return true;
}
```



Jednakost elemenata

- Metod **equals** treba implementirati po sledećim pravilima:
 1. Ako su reference objekata jednake (`this == obj`), objekti su jednaki (tj. u pitanju je isti objekat);
 2. Poređenje nekog ne-null objekta sa null je uvek false. Dve null vrednosti su po definiciji jednake;
 3. Ako su objekti različitih klasa (npr. dobili smo objekat tipa Integer), rezultat je false; i
 4. Poredimo jednakosti odgovarajućih polja, pri čemu ponovo koristimo pravilo 2.
- Ovo je standardna preporučena implementacija metoda **equals** i navedena pravila se mogu primeniti u svakoj klasi.



Jednakost elemenata

- Ako objekte ovako proširene klase Osoba iskoristimo u primeru, ponovo nećemo dobiti rezultat koji očekujemo, tj. skup će ponovo imati dva elementa.
- To je zato što kolekcije zbog brzine koriste još jedan metod klase Object: **hashCode**. Ovaj metod treba da vrati celobrojnu heš vrednost objekta. Kolekcije ga pozivaju pre metoda equals i oslanjaju se na sledeća dva principa:
 - Ako dva objekta imaju različite heš vrednosti, onda su objekti sigurno različiti, i kolekcija neće ni pozvati metod equals;
 - Ako dva objekta imaju istu heš vrednost, onda oni mogu, ali ne moraju biti jednaki. U ovom slučaju, kolekcija će pozvati i metod equals da bi proverila jednakost.



Jednakost elemenata

- Neka, na primer, heš funkcija za stringove jednostavno sabira ASCII vrednosti karaktera.
- Stringovi “abc” i “xyz” imaju različite heš vrednosti i samim tim nisu jednaki, te metod equals neće biti pozvan. Ovo je pravilo 1.
- Međutim, stringovi “abc” i “cba” imaju istu heš vrednost, ali moramo porediti pojedinačne karaktere da bismo uvideli da zapravo nisu jednaki. Ovo je pravilo 2.
- Kolekcije koriste heš vrednosti, jer važi pretpostavka da je njeno računanje brže od poređenja jednakosti objekata. Pored toga, kolekcija će metod hashCode pozvati samo jednom i zapamtiti povratnu vrednost, što će znatno ubrzati rad.
- Naredni primer demonstrira kako se pozivaju metodi hashCode i equals prilikom ubacivanja objekata u skup.

Primer 10.14: Pozivanje metoda hashCode i equals

```
class Znak {
    private char zn;

    public Znak(char zn) { this.zn = zn; }

    @Override
    public int hashCode() {

        System.out.print("; hashCode za " + zn);
        return zn;
    }

    @Override
    public boolean equals(Object o) {
        // zbog jednostavnosti preskacemo punu implementaciju
        Znak drugi = (Znak) o;
        System.out.println("; equals poredi " + zn + " sa " + drugi.zn);
        return zn == drugi.zn;
    }
}

public class Jednakost {
    public static void main(String[] args) {
        Set<Znak> s = new HashSet<>();
        System.out.print("Ubacujem a");
        s.add(new Znak('a'));
        System.out.println();

        System.out.print("Ubacujem b");
        s.add(new Znak('b'));
        System.out.println();

        System.out.print("Ubacujem c");
        s.add(new Znak('c'));
        System.out.println();

        System.out.print("Ubacujem ponovo b");
        s.add(new Znak('b'));
    }
}
```



Jednakost elemenata

- Program će ispisati sledeće:

```
Ubacujem a; hashCode za a
Ubacujem b; hashCode za b
Ubacujem c; hashCode za c
Ubacujem ponovo b; hashCode za b; equals poredi b sa b
```

- Dakle, poređenje jednakosti objekata u kolekcijama funkcioniše na sledeći način:
 1. Metod **hashCode** se poziva samo jednom, prilikom ubacivanja objekta. Povratna vrednost se čuva interno u kolekciji;
 2. Kada ubacimo nove elemente, b i c, računaju se njihove heš vrednosti i interno porede sa svim postojećim. Pošto se njihove heš vrednosti razlikuju od postojećih, poređenje se zaustavlja;
 3. Kada ponovo ubacimo element b, kolekcija pronalazi postojeći element sa istom heš vrednošću i tek tada poziva metod equals da bi proverila da li su elementi zaista jednaki.



Jednakost elemenata

- Implementacija dobre heš funkcije je težak zadatak.
- Algoritam koji je jednostavan, a daje relativno dobre rezultate, funkcionije po sledećem principu.
- Početna vrednost rezultata je 1. Za svako polje klase koje se koristi za poređenje jednakosti, množimo rezultat nekim prostim brojem i na to dodajemo heš vrednost polja.
- Pri tome, heš vrednost celobrojnog polja je sam broj, dok većina klasa standardne Java biblioteke (poput klase String) implementira dobru heš funkciju.
- Za klase koje su pisali drugi programeri možemo samo da se nadamo da su implementirali heš funkciju kako treba. Implementacija metoda hashCode za klasu Osoba je data u nastavku.
- Ako objekte ovako implementirane klase ubacimo u skup, program će konačno funkcionisati kao što se očekuje.

Jednakost elemenata

Primer 10.15: Implementacija metoda hashCode u klasi Osoba

```
@Override
public int hashCode() {
    final int p = 31; // neki prost broj
    int result = 1;
    result = p * result + ((ime == null) ? 0 : ime.hashCode());
    result = p * result + ((prezime == null) ? 0 : prezime.hashCode());
    return result;
}
```

- Da zaključimo: ako objekte ubacujemo u HashSet ili LinkedHashSet, ili ih koristimo kao ključeve u kolekcijama HashMap ili LinkedHashMap, onda njihove klase moraju implementirati i hashCode i equals.
- Ako elemente pretražujemo na neki drugi način, na primer u listama pozivom metoda indexOf, tada je dovoljno implementirati samo metod equals.

Umesto pamćenja ovih pravila, jednostavnije je (i bolje) implementirati i hashCode i equals kad god ćemo elemente ubacivati u neku kolekciju. Možda ćemo dve godine kasnije odlučiti da promenimo kolekciju, i, ako nismo implementirali ove metode, program može prestati da radi “iz neobjašnjivih razloga”.



Sortiranje elemenata

- Slično kao kod poređenja jednakosti, kod sortiranja elemenata je neophodno implementirati mehanizam pomoću kog kolekcija može utvrditi poredak elemenata.
- Ovo se postiže ili implementacijom interfejsa `java.lang.Comparable`,
- ili interfejsa `java.util.Comparator`. Prvi sadrži metod `compareTo` (T o), koji treba da poredi tekući objekat (this) sa prosleđenim.
- Drugi interfejs sadrži metod `compare` (T o1, T o2) koji treba da poredi dva prosleđena objekta.
- Oba metoda vraćaju celobrojnu vrednost, i to manju od nule ukoliko je prvi objekat manji od drugog ($this < o$, odnosno $o1 < o2$), veću od nule ukoliko je prvi objekat veći od drugog, odnosno 0 ako su objekti jednaki.



Sortiranje elemenata

- Interfejs **Comparable** koristimo za sortiranje objekata klase čiji nam je izvorni kod dostupan i koji možemo menjati (na primer, mi smo napisali klasu).
- Ovo je zato što se **Comparable** implementira u klasi čiji se objekti sortiraju (što se može zaključiti iz naziva interfejsa - objekti postaju *uporedivi*).
- Interfejs **Comparator** koristimo za sortiranje objekata klase koje ne možemo menjati (na primer, stringovi). Comparator se implementira u zasebnoj klasi, čija će instanca (upoređivač) vršiti poređenje željenih objekata, dva po dva.
- U narednom primeru je dat program koji sortira objekte klase Osoba rastuće po prezimenu, a ako su prezimena jednaka, onda rastuće po imenu. Program se oslanja na postojeći metod compareTo klase String.

Primer 10.16: Sortiranje objekata upotrebom interfejsa Comparable

```
class Osoba implements Comparable<Osoba> {
    ...

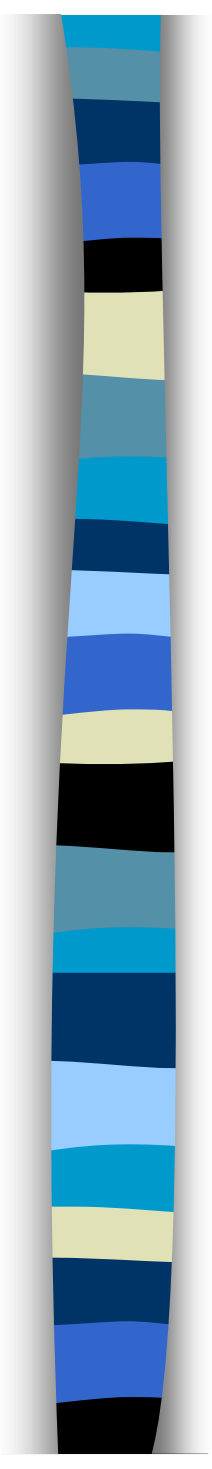
    @Override
    public int compareTo(Osoba o) {
        int n = prezime.compareTo(o.prezime);
        if (n != 0) // prezimana su razlicita
            return n;
        return ime.compareTo(o.ime);
    }

    @Override
    public String toString() {
        return prezime + " " + ime;
    }
}

public class SortComparable {
    public static void main(String[] args) {
        Set<Osoba> s = new TreeSet<>();
        s.add(new Osoba("Petar", "Petrovic"));
        s.add(new Osoba("Ana", "Petrovic"));
        s.add(new Osoba("Petar", "Petrovic"));
        System.out.println(s);
    }
}
```

Program će ispisati [Petrovic Ana, Petrovic Petar].

Dakle, kolekcije koje automatski vrše sortiranje pozivaju metod `compareTo` da bi utvrdile uređenje objekata. Ako metod vrati vrednost 0, kolekcija podrazumeva da su objekti jednaki, kao u datom primeru.



Ako za sortiranje koristimo Comparator i implementiramo ga u novoj klasi, onda objekat te nove klase prosleđujemo kolekciji u konstruktoru, kao što je prikazano u sledećem primeru.

Primer 10.17: Sortiranje objekata upotrebom interfejsa Comparator

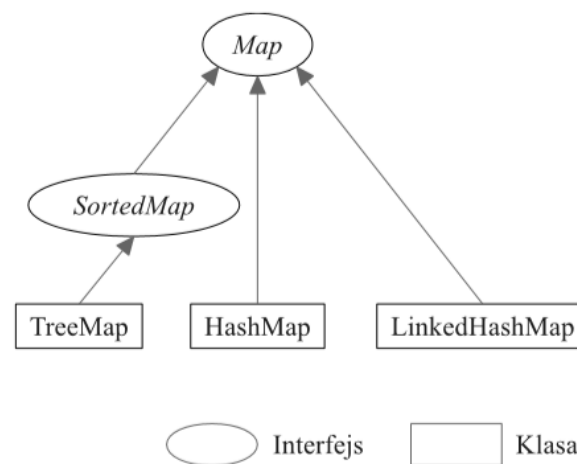
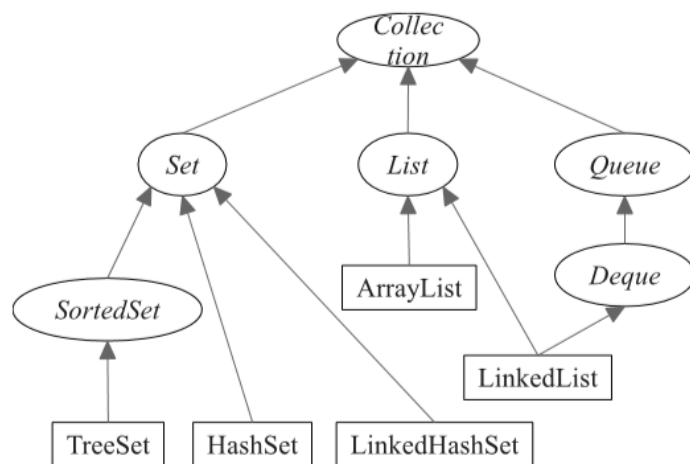
```
class StringComp implements Comparator<String> {
    @Override
    public int compare(String o1, String o2) {
        // obrnemo znak, opadajuće sortiranje
        return -o1.compareTo(o2);
    }
}

public class OpadajuciStringovi {
    public static void main(String[] args) {
        StringComp uporedjivac = new StringComp();
        Set<String> s = new TreeSet<>(uporedjivac);
        s.add("mika");
        s.add("pera");
        s.add("zika");
        System.out.println(s);
    }
}
```

Redosled stringova će sada biti određen povratnim vrednostima metoda compare objekta upoređivač.

Odabir kolekcije

- Prilikom rada sa kolekcijama je najpre potrebno odabrati najpogodniji interfejs.
- Nakon toga, na osnovu diskusije o performansama i funkcionalnostima, potrebno je odabrati odgovarajuću implementaciju. Za uspešno obavljanje ovog posla neophodno je poznavati hijerarhiju standardnih Java kolekcija. Hijerarhija interfejsa i klasa opisanih u ovoj prezentaciji je data na slici.



Odabir kolekcije

- Sve opisane kolekcije su za tzv. opštu namenu. Pored ovih, postoje i specijalizovane kolekcije. Na primer, paket `java.util.concurrent` sadrži brojne dodatne kolekcije koje se mogu slobodno koristiti u višenitnim okruženjima.
- Upotreba starijih kolekcija, kao što su Vector i Stack, se ne preporučuje u novim aplikacijama. Ove kolekcije su implementirane tako da obezbeđuju sinhronizovan pristup čak i kada ih koristi samo jedna nit. Zbog toga mogu imati lošije performanse nego nove kolekcije.

