



CS202 - OBJEKTNO-ORIJENTISANO PROGRAMIRANJE 2

Liste, stekovi, redovi i prioritetni
redovi

Lekcija 07

PRIRUČNIK ZA STUDENTE

CS202 - OBJEKTNO-ORIJENTISANO PROGRAMIRANJE 2

Lekcija 07

LISTE, STEKOVI, REDOVI I PRIORITETNI REDOVI

- ✓ Liste, stekovi, redovi i prioritetni redovi
- ✓ Poglavlje 1: Kolekcije
- ✓ Poglavlje 2: Iteratori
- ✓ Poglavlje 3: Liste
- ✓ Poglavlje 4: Interfejs Comparator
- ✓ Poglavlje 5: Statički metodi u listama i kolekcijama
- ✓ Poglavlje 6: Studija slučaja: lopte koje skaču
- ✓ Poglavlje 7: Klase Vector i Stack
- ✓ Poglavlje 8: Redovi i prioritetni redovi
- ✓ Poglavlje 9: Vežba – Pokazni primeri
- ✓ Poglavlje 10: Vežba – Zadaci za individualni rad
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

Izbor najboljih struktura podataka i algoritama za određeni zadatak je jedan od ključnih faktora razvoja softvera visokih performansi

Ova lekcija treba da ostvari sledeće ciljeve:

- Da analizira vezu između interfejsa i klase u hijerarhiji **JCF (Java Collections Framework)**
- Da ukaže na zajedničke metode definisane u interfejsu **Collection** koji se koristi pri radu sa kolekcijama
- Da pokaže upotrebu interfejsa **Iterator** za rad sa elementima kolekcije
- Da istakne upotrebu **for-each** petlje kod rada sa elementima kolekcije
- Da ispita kakao i gde se upotrebljava **ArrayList** ili **LinkedList** za smeštaj liste elemenata
- Da pokaže kako se mogu upoređivati elementi implementiranjem interfejsa **Comparable** i **Comparator**
- Da pokaže upotrebu statičkih metoda klase **Collections** za čuvanje, pretraživanje, mešanje listi i nalaženje najvećeg ili najmanjeg elementa kolekcija
- Da prikaže razvoj aplikacije sa više lopti koje skaču upotrebom klase **ArrayList**
- Da ukaže na razliku između klase **Vector** i **ArrayList**, kao i na upotrebu klase **Stack** za kreiranje stekova
- Da ispita veze između klase **Collection**, **Queue**, **LinkedList** i **PriorityQueue** i načina kreiranja redova čekanja upotrebom klase **PriorityQueue**
- Da pokaže upotrebu stekova pri pisanju programa koji izračunavaju izraze

Referenca: Y. Daniel Liang, INTRODUCTION TO JAVA PROGRAMMING (COMPREHENSIVE VERSION), Tenth Edition, Pearson, ISBN 10: 0-13-376131-2, ISBN 13: 978-0-13-376131-3

Ovo je osnovni udžbenik za ovaj predmet i preporučuje se studentima da ga koriste

▼ Poglavlje 1

Kolekcije

ŠTA JE STRUKTURA PODATAKA?

Struktura podataka je kolekcija (zbirka) podataka organizovanih na neki način a podržava i operacije neophodne za pristup podacima i za rad sa njima.

Struktura podataka je kolekcija (zbirka) podataka organizovanih na neki način. Struktura ne služi samo za smeštaj (memorisanje) podataka, već takođe podržava operacije neophodne za pristup podacima i za rad sa njima.

Ako se razmišlja na objektno-orijentisan način, na strukturu podataka se može gledati kao na neki kontejner ili kontejner objekat koji čuva druge objekte, koji se nazivaju podacima ili elementima.

Da bi se definisala neka kolekcija podataka, neophodno je da se definiše odgovarajuća klasa. Ta klasa bi trebalo da upotrebljava polja podataka (engl., **data fields**) za uskladištenje podataka i da obezbeđuje metode koji podržavaju operacije za podacima, kao što su operacije pretraživanja, ubacivanja i brisanja. Da bi se kreirala određena struktura podataka, prema tome, neophodno je da se kreira primerak, tj. objekat, takve klase. Onda možete da koristite metode njene klase da bi manipulisali strukturom podataka, kao na primer, da bi ubacivali elemente u neku strukturu podataka, ili da bi obrisali neki element iz strukture podataka.

Pored klase **ArrayList**, Java obezbeđuje nekoliko drugih struktura podataka za efikasno organizovanje i za manipulaciju podataka. One se nazivaju zajedničkim imenom **Java Collections Framework**, tj. okvirom Java kolekcija. **Kolekcije čine liste, vektori, stekovi, redovi čekanja i prioritetni redovi čekanja.**

VRSTE STRUKTURE PODATAKA

Kolekcije čine liste, vektori, stekovi, redovi čekanja i prioritetni redovi čekanja.

Java Collections Framework podržava dva tipa kontejnera:

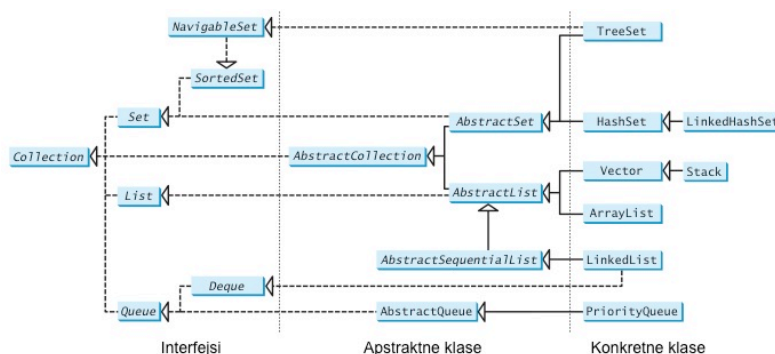
1. Kolekcija - za uskladišćenje kolekcije elemenata
2. Mapa - za uskladišćenje parova ključ/vrednost

Mape su strukture podataka za efikasno nalaženja nekog elementa upotrebom ključa. U ovom tekstu se one dalje ne izučavaju, jer je fokus na kolekcijama.

Postoji više vrsta kolekcija:

1. **Skupovi (eng. sets)** - za uskladištenje grupe elemenata bez ponavljanja (istih elemenata)
2. **Liste (eng. lists)** - za uskladištenje uređene kolekcije elemenata
3. **Redovi (eng. queues)** - za uskladištenje objekata koji se uređuju po principu FIFO, tj. "first-in, first-out", tj. po redosledu ubicivanja i uzimanja elemenata iz kolekcije.

Zajednička svojstva ovih kolekcija se definišu u interfejsima, a njihova implementacija (primena) se realizuje u vidu odgovarajućih klasa, prikazanih na slici 1.



Slika 1.1 Hijerarhija klasa kolekcija koje su kontejneri koji skladište objekte [1]

INTERFEJS COLLECTION

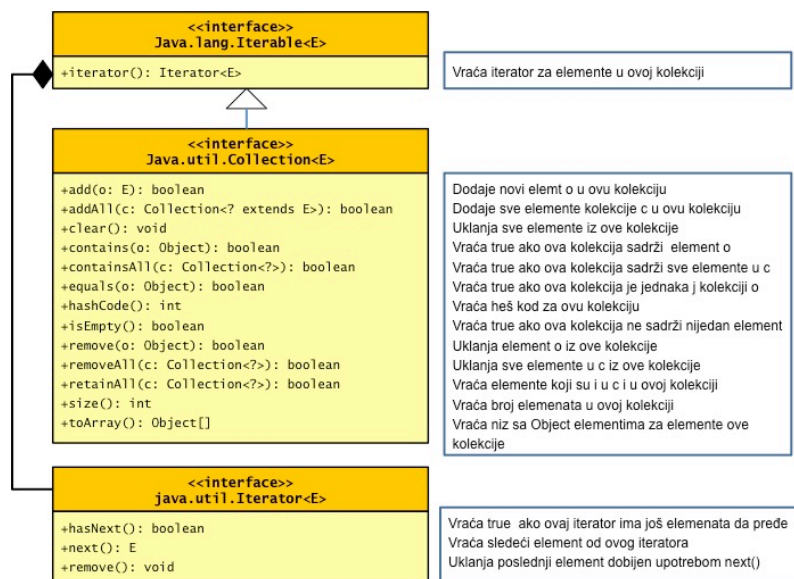
Interfejs Collection definiše zajedničke operacije nad listama, vektorima, stekovima, redovima čekanja, prioritetnim redovima čekanja i skupovima.

Interfejs Collection je koren interfejs klasa za manipulaciju kolekcija objekata. Njegovi javni metodi su prikazani na slici 2. Klasa **AbstractCollection** obezbeđuje delimičnu implementaciju **Collection** interfejsa. Ona primenjuje sve metode iz **Collection** interfejsa sem metoda **size()** i **iterator()**. Ove metode su primenjene u odgovarajućim potklasama.

Interfejs **Collection** obezbeđuje osnovne operacije za dodavanje i uklanjanje elemenata neke kolekcije. Metod **add()** dodaje element u kolekciju. Metod **addAll()** dodaje sve elemente specificirane kolekcije u ovu kolekciju. Metod **remove()** uklanja neki element iz kolekcije. Metod **removeAll()** uklanja elemente iz ove kolekcije. Metod **retainAll()** zadržava elemente u ovoj kolekciji. Sve ovi elementi vraćaju **boolean**, tj. logičku promenljivu. Vraća se vrednost te promenljive sa vrednošću **true** ako je kolekcija promenjena, kao rezultat izvršenja metoda. Metod **clear()** uklanja sve elemente iz kolekcije.

Interfejs **Collection** omogućava i različite operacije upita. Metod **size()** vraća broj elemenata u kolekciji. Metod **contains()** proverava da li kolekcija sadrži specificiran element. Metod **containsAll()** proverava da li kolekcija sadrži sve elemente specificirane kolekcije. Metod **isEmpty()** vraća **true** ako je kolekcija prazna.

Metod **toArray()** interfejsa **Collection** vraća niz koji je sadržan u kolekciji.



Slika 1.2 Metodi interfejsa Collection, koji manipulišu elementima u kolekciji i objekat Iterator za pristup elementima kolekcije [1]

PRIMER UPOTREBE INTERFEJSA COLLECTION

*Primer kreira listing klase **TestCollection** u kome se testiraju metodi interfejsa **Collection**: **contains()**, **remove()**, **size()**, **addAll()**, **retainAll()** i **removeAll()***

Program kreira konkretan objekat kolekcije upotrebom klase **ArrayList** (linija 6), i poziva metode **contains()** (linija 15), **remove()** (linija 17), **size()** (linija 18), **addAll()** (linija 31), **retainAll()** (linija 36) i **removeAll()** (linija 41) interfejsa **Collection**.

Umesto klase **ArrayList**, mogu se koristiti i klase **HashSet**, **LinkedList**, **Vector** i **Stack**, pri čemu se koriste isti metodi interfejsa **Collection**. Svaka klasa sadržana u **Java Collection Framework**, sem klase **java.util.PriorityQueue**, primenjuje metod **clone()**. Program kreira kopiju niza (linija 30, 35, 40), da bi se zadržao originalni niz i upotrebljava njegovu kopiju da bi primenio metode **addAll()**, **retainAll()** i **removeAll()**. Na slici 3 prikazan je rezultat, tj. prikaz na monitoru računara, izvršenja programa **TestCollection**.

```

A list of cities in collection1:
[New York, Atlanta, Dallas, Madison]
Is Dallas in collection1? true
3 cities are in collection1 now
A list of cities in collection2:
[Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 or collection2:
[New York, Atlanta, Madison, Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 and collection2: [Atlanta]
Cities in collection1, but not in 2: [New York, Madison]
  
```

Slika 1.3 Dobijen rezultat izvršenja programa **TestCollection** [1]

```

import java.util.ArrayList;
import java.util.Collection;
public class TestCollection {
  
```

```
public static void main(String[] args) {
    ArrayList<String> collection1 = new ArrayList<String>();
    collection1.add("New York");
    collection1.add("Atlanta");
    collection1.add("Dallas");
    collection1.add("Madison");

    System.out.println("A list of cities in collection1:");
    System.out.println(collection1);

    System.out.println("\nIs Dallas in collection1? " +
collection1.contains("Dallas"));

    collection1.remove("Dallas");
    System.out.println("\n" + collection1.size() + " cities are in collection1
now");

    Collection<String> collection2 = new ArrayList<String>();
    collection2.add("Seattle");
    collection2.add("Portland");
    collection2.add("Los Angeles");
    collection2.add("Atlanta");

    System.out.println("\nA list of cities in collection2:");
    System.out.println(collection2);

    ArrayList<String> c1 = (ArrayList<String>) (collection1.clone());
    c1.addAll(collection2);
    System.out.println("\nCities in collection1 or collection2: ");
    System.out.println(c1);

    c1 = (ArrayList<String>) (collection1.clone());
    c1.retainAll(collection2);
    System.out.print("\nCities in collection1 and collection2: ");
    System.out.println(c1);

    c1 = (ArrayList<String>) (collection1.clone());
    c1.removeAll(collection2);
    System.out.print("\nCities in collection1, but not in 2: ");
    System.out.println(c1);
}
}
```

VIDEO: LISTE

Jakob Jenkov: Java List Tutorial (16,34 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 2

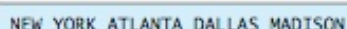
Iteratori

ŠTA JE ITERATOR?

*Svaka kolekcija ima objekat tipa **Iterator** koji se koristi da bi se pristupilo svakom od elemenata kolekcije*

Iterator je klasičan šablon projektovanja (eng. **design pattern**) za pretraživanje neke strukture podataka bez potrebe da se ulazi u detalje kako su podaci uskladišteni u strukturi podataka.

Interfejs **Collection** nasleđuje interfejs **Iterable**. Interfejs **Iterable** definiše metod **iterator()**, koji vraća iterator. Interfejs **Iterator** obezbeđuje uniformni način za pristupanje elementima različitog tipa u nekoj kolekciji. Metod **iterator()** u interfejsu **Collection** vraća primerak interfejsa **Iterator**. On obezbeđuje sekvencijalni pristup elementima kolekcije primenom metoda **next()**. Možete koristiti takođe i **hasNext()** metod da bi proverili da li ima još elemenata u iteratoru, a metod **remove()** da bi uklonili poslednji element koji je vratio iterator. Listing klase **TestIterator** daje primer upotrebe iteratora radi pristupa svim elementima jednom nizu, a na slici 1 jedan je prikazan dobijeni rezultat. Program kreira konkretni objekat kolekcije upotrebom klase **ArrayList** (slika 5) i dodaje četiri stringa na linijama 9-12. Program onda obezbeđuje iterator za kolekciju (linija 14) i upotrebljava iterator radi pristupa svim stringovima liste i prikazuje stringove sa velikim slovima (linije 15-17).



Slika 2.1 Dobijen rezultat izvršenja programa TestIterator [1]

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

public class TestIterator {

    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<String>();
        collection.add("New York");
        collection.add("Atlanta");
        collection.add("Dallas");
        collection.add("Madison");

        Iterator<String> iterator = collection.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next().toUpperCase() + " ");
        }
    }
}
```



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

VIDEO: ITERATORI - DEO 1

Iterators Part 1 (Java) Nathan Schutz (4,06 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: ITERATORI - DEO 2

Iterators Part 2 (Java) Nathan Schutz Nathan Schutz (2,32 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: ITERATORI - DEO 3

Iterators Part 3: ListIterator (Java) Nathan Schutz Nathan Schutz (5,03 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 3

Liste

INTERFEJS LIST

Interfejs List nasleđuje interfejs Collection i definiše kolekciju za smeštaj elemente u sekvencijalnom rasporedu, a kreira se korišćenjem klase ArrayList ili LinkedList.

Interfejs List nasleđuje interfejs Collection i definiše uređenu kolekciju u kojoj elementi mogu da se ponavljaju u listi. Klase **ArrayList** i **LinkedList** koriste interfejs **List**. Interfejs **List** dodaje operacije povezane sa pozicijom, kao i novi iterator liste koji omogućuje korisniku prelaz po listi u oba smera. Na slici 1 prikazani su metodi interfejsa **List**.

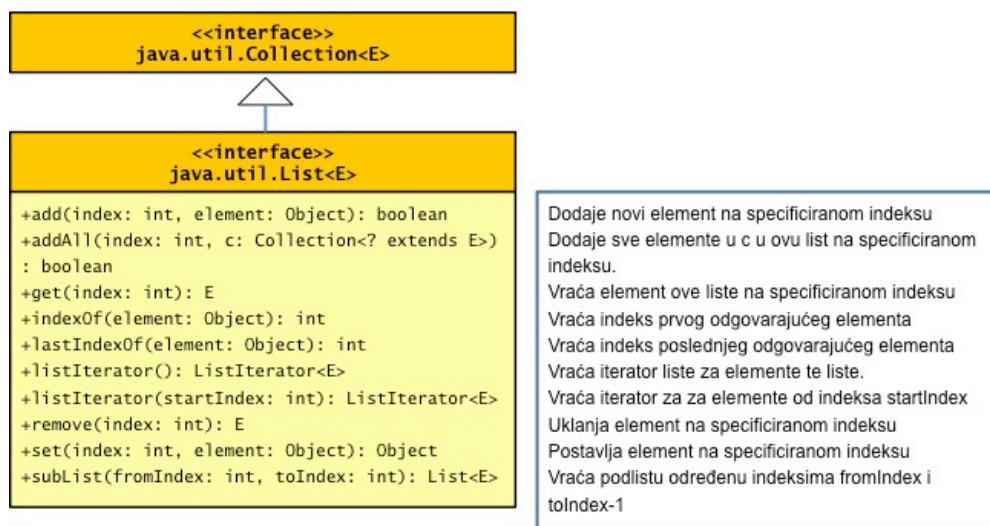
Metod **add(index, element)** se koristi za ubacivanje elementa na mesto određenog indeksa, a metod **addAll(index, collection)** ubacuje kolekciju elemenata od mesta određenom indeksom.

Metod **remove(index)** uklanja element koji je na mestu definisanom indeksom liste.

Novi element se može ubaciti na mesto određenim indeksom upotrebom metoda **set(index, element)**.

Metod **indexOf(element)** daje indeks prvog javljanja određenog elementa liste, a metod **lastIndexOf(element)** daje indeks poslednjeg javljanja elementa u listi.

Može se dobiti i podlista upotrebom metoda **subList(fromIndex, toIndex)**. Metodi **listIterator()** ili **listIterator(startIndex)** vraća primerak interfejsa **ListIterator**.



Slika 3.1 Dobijen rezultat izvršenja programa TestIterator [1]

INTERFEJS LISTITERATOR

Interfejs ListIterator nasleđuje interfejs Iterator da bi omogućio prolaženje kroz elemente liste u oba smeru

InterfejsListIterator nasleđuje interfejs Iterator da bi omogućio prolaženje kroz elemente liste u oba smeru. Metodi interfejsa **ListIterator** su prikazani na slici 2.

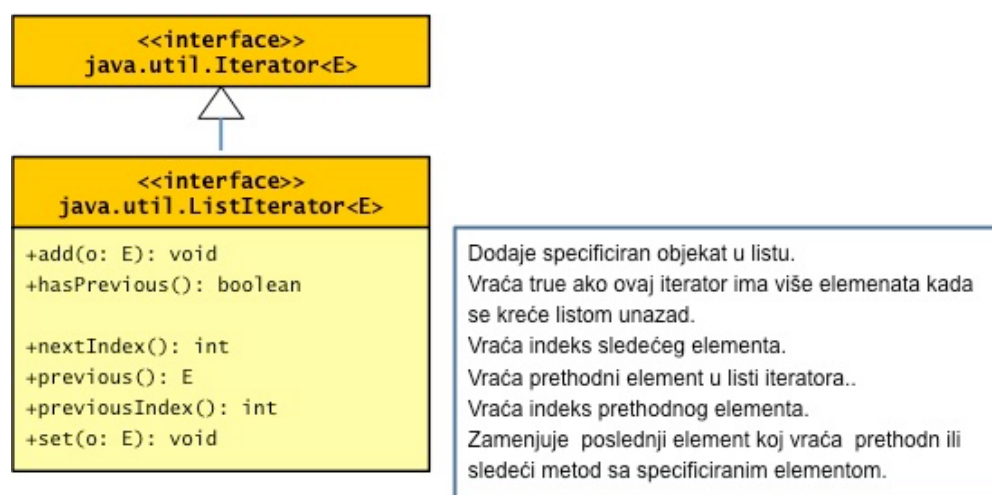
Metod **add(element)** ubacuje specificiran element u listu. On se ubacuje odmah posle pre sledećeg elementa, koji se vraća primenom metoda **next()** definisanog u interfejsu Iterator interface, ako postoji, a vraća se posle elementa, ako postoji, koji se dobija metodom **previous()**.

Ako lista nema nijedan element, novi element postaje jedini element u listi. Metod **set(element)** zamenjuje poslednji element u listi, a koji bi se vratio metodom **next()** ili sa metodom **previous()** sa specificiranim elementom.

Metod **hasNext()** proverava da li iterator ima još elemenata kada on ide u smeru kraja liste.

Metod **hasPrevious()** proverava da li iterator ima još elemenata kada se ide u smeru unazad. Metod **next()** vraća sledeći element u iteratoru, a metod **previous()** vraća prethodni element u iteratoru. Metod **nextIndex()** vraća indeks sledećeg elementa u iteratoru, a metod **previousIndex()**, vraća indeks prethodnog elementa.

Klasa **AbstractList** implementira interfejs List. Klasa **AbstractSequentialList** nasleđuje **AbstractList** da bi se obezbedila podrška povezanih listi.



Slika 3.2 Dobijen rezultat izvršenja programa TestIterator [1]

KLASE ARRAYLIST I LINKEDLIST

ArrayList smešta elemente u niz koji se dinamički kreira. Ako se preskorači kapacitet niza, kreira se veći novi niz. LinkedList smešta elemente u vidu povezane liste

Klasa `ArrayList` i klasa `LinkedList` su dve konkretne primene interfejsa `List`, tj. implementiraju interfejs `List`. `ArrayList` smešta elemente u niz koji se dinamički kreira. Ako se prekorači kapacitet niza, kreira se veći novi niz i svi elementi tekućeg niza se kopiraju u novi niz. `LinkedList` smešta elemente u vidu povezane liste. Zavisno od vaših potreba, koristite jednu od ove dve klase.

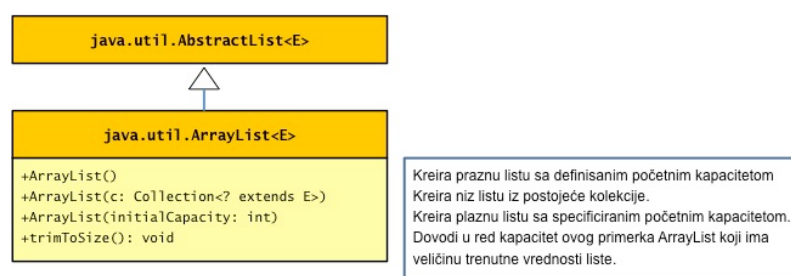
- Ako treba da podržavate slučajni (eng. `random`) pristup elementima preko indeksa bez ubacivanja ili uklanjanja elemenata na početku liste, klasa `ArrayList` nudi najefikasnije rešenje.
- Ako vaša aplikacija zahteva ubacivanje ili brisanje elemenata na početku liste, onda bi trebalo da koristite klasu `LinkedList`.

Lista može da se dinamički povećava ili smanjuje, dok je niz, kada je kreiran fiksno. *Ako vaša aplikacija ne zahteva ubacivanje ili brisanje elemenata, niz je najefikasnija struktura podataka.*

Klasa `ArrayList` je niz promenljive veličine koji implementira interfejs `List`. On sadrži i metode za manipulaciju veličinom niza radi uskladišćenja liste, kao što je prikazano na slici 3. Svaki primerak klase `ArrayList` ima svoj kapacitet koji je određen veličinom niza koji se upotrebljava za smeštaj elemenata liste..

Njegova najmanja veličina je kao veličina liste. Kako se elementi dodaju (u aplikaciji), `ArrayList` automatski povećava porast kapaciteta niza. Međutim, `ArrayList` ne obezbeđuje automatsko smanjivanje niza (kada aplikacija briše elemente). Da bi smanjili smanjili niz, treba da koristite metod `trimToSize()` koji smanjuje niz tako da bude jednak veličini smeštenog niza.

Primerak (objekat) klase `ArrayList` se može kreirati ili konstruktorom bez argumenata, ili korišćenjem dva konstruktora sa argumentima: `ArrayList(collection)` i `ArrayList(initialCapacity)`.



Slika 3.3 Klasa `ArrayList` implementira interfejs `List` upotrebom nekog niza [1]

PITANJA

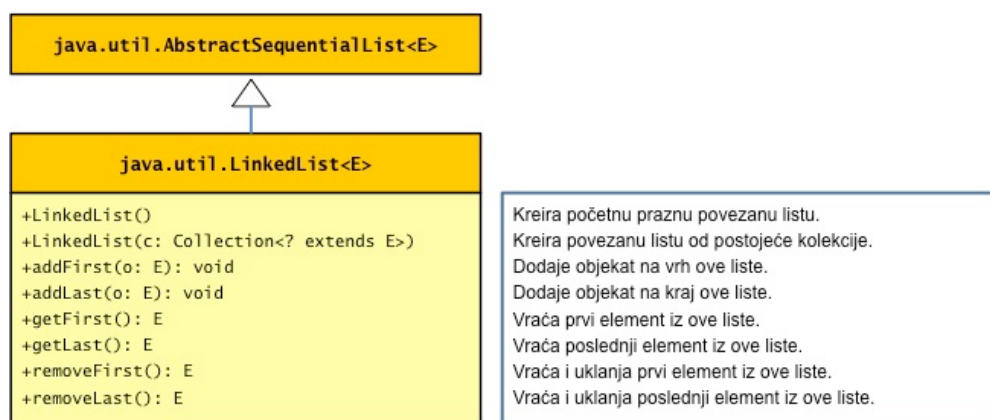
Proverite svoje znanje stečeno u ovoj sekciji.

1. Kako kreirate objekat klase **Vector**? Kako dodajete ili ubacujete elemente u vektor? Kako uklanjate elemente iz vektora? Kako nalazite veličinu vektora?
2. Kako kreirate objekat klase **Stack**? Kako dodajete novi element u stek? Kako uklanjate element iz steka? Kako nalazite veličinu steka?
3. Da li se listing klase **TestCollection** kompajlira i izvršava ako se **ArrayList** zameni sa **LinkedList**, **Vector** ili **Stack**?

KLASA LINKEDLIST

*Klasa **LinkedList** omogućava implementaciju povezanih listi implementiranjem interfejsa **List***

Klasa **LinkedList** omogućava implementaciju povezanih listi implementiranjem interfejsa **List**. Pored toga, ova klasa obezbeđuje i metode za pretraživanje, ubacivanje i uklanjanje elemenata na oba kraja liste, kao što je prikazano na slici 4. Konstruktor za **LinkedList** je ili bez argumenata ili **LinkedList(collection)**.



Slika 3.4 Klasa **LinkedList** obezbeđuje metode za dodavanje i ubacivanje elemenata na oba kraja liste [1]

PRIMER KORIŠĆENJA KLASA ARRAYLIST I LINKEDLIST

Primer kreira povezanu listu iz liste niza i ubacuje i izbacuje elemente sa liste.

Ovde prikazan listing programa koji kreira listu niza sa brojevima i koji ubacuje nove elemente na određene lokacije u listi. Primer kreira povezanu listu iz liste niza i ubacuje i izbacuje elemente sa liste. Na kraju, primer prikazuje pristup elementima liste unapred i unazad.

Na slici 4 prikazan je dobijeni rezultat (na displeju) izvršenja programa **TestArrayAndLinkedList**.

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forward:
green 10 red 1 2 30 3 1
Display the linked list backward:
1 3 30 2 1 red 10 green
```

Slika 3.5 Dobijeni prikaz na displeju, kao rezultat izvršenja programa TestArrayAndLinkedList [1]

Lista može da ima identične elemente. Ceo broj 1 je uskladišten dva puta u listi (linije 10, 13). Na sličan način rade **ArrayList** i **LinkedList**. Kritična razlika između njih se odnosi na unutrašnju implementaciju, koja utiče na performanse. Klasa **ArrayList** je efikasna za pronalaženje elemenata, a **LinkedList** je efikasna klasa za ubacivanje i uklanjanje elemenata na početku liste. Obe klase imaju slične performanse za ubacivanje i uklanjanje elemenata u sredini i na kraju liste.

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class TestArrayAndLinkedList {

    public static void main(String[] args) {
        List<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(1); // 1 is autoboxed to new Integer(1)
        arrayList.add(2);
        arrayList.add(3);
        arrayList.add(1);
        arrayList.add(4);
        arrayList.add(0, 10);
        arrayList.add(3, 30);

        System.out.println("A list of integers in the array list:");
        System.out.println(arrayList);

        LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
        linkedList.add(1, "red");
        linkedList.removeLast();
        linkedList.addFirst("green");

        System.out.println("Display the linked list forward:");
        ListIterator<Object> listIterator = linkedList.listIterator();
        while (listIterator.hasNext()) {
            System.out.print(listIterator.next() + " ");
        }
        System.out.println();

        System.out.println("Display the linked list backward:");
        listIterator = linkedList.listIterator(linkedList.size());
```

```
while (listIterator.hasPrevious()) {  
    System.out.print(listIterator.previous() + " ");  
}  
}
```

PRIMENA METODA GET()

Metod get() se upotrebljava za povezane liste, ali radi dosta sporo. Ne treba ga koristiti za pristup svim elementima liste

Metod get(1) se upotrebljava za povezane liste, ali radi dosta sporo. Ne treba ga koristiti za pristup svim elementima liste, kao što je prikazano na slici 6a. Umesto takvog korišćenja, bolje je koristiti iterator, kao što je pokazano na slici 6b. Kao što se vidi, za svaku petlju upotrebljava se implicitno iterator.

```
for (int i = 0; i < linkedList.size(); i++) {  
    process linkedList.get(i);  
}
```

(a)

```
for (listElementType s: linkedList) {  
    process s;  
}
```

(b)

Slika 3.6 Primer neefikasne i efikasne primene metoda get() [1]

Java obezbeđuje statički metod **asList()** za kreiranje liste koristeći liste različitih veličina generičkog tipa. Na ovaj način, možete koristiti sledeći kod radi kreiranja liste sa stringovima i liste sa celim brojevima:

```
List<String> list1 = Arrays.asList("red", "green", "blue");  
List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);
```

VIDEO: UVOD U POVEZANE LISTE

Data Structures: Introduction to Linked Lists (13,39 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: POVEZANE LISTE - DEO 1

Derek Banas: Linked List in Java (17,38 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: POVEZANE LISTE - DEO 2

Derek Banas: Linked List in Java 2 (19,11 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 4

Interfejs Comparator

FUNKCIJA INTERFEJSA COMPARATOR

Comparator se koristi za upoređivanje objekata klase koja ne implementira interfejs Comparable.

Više klasa Java API, kao što su String, **Date**, **Calendar**, **BigInteger**, **BigDecimal** i sve omotačke klase primitivnih tipova koriste interfejs **Comparable**. On definiše metod **compareTo()** metod, koji služi za upoređivanje dva elementa klase koje implementiraju interfejs Comparable. On omogućava upoređenje objekata iste klase.

Kako uporediti elemente klase koje ne implementiraju interfejs **Comparable**, ili ako su elementi različitog tipa? Da li se ovi elementi mogu da upoređuju? To omogućava interfejs Comparator, jer obezbeđuje upoređenje objekata različitih klasa. Možete definisati neki način upoređivanja, tj. upoređivač, da bi poredili elemente različitih klasa. Da bi ovo uradili, neophodno je da se definiše klasa koja implementira **interfejs java.util.Comparator<T>**. **Comparator<T>** interfejs ima dve metode, compare i equals:

```
public int compare(T element1, T element2)
```

Metod **compare()** vraća nenegativnu vrednost ako element1 je manji od element2, a pozitivnu vrednost ako je element1 veći od element2 i vraća nulu ako su jednaki.

Metod **equals()** vraća true ako je specificiran objekat je takođe upoređivač (eng. **comparator**) i daje isti redosled kao i ovaj upoređivač:

```
public boolean equals(Object element)
```

Metod **equals()** se takođe definiše u klasi **Object**. Zato, nećete dobiti grešku kompajliranja ako ne primenite **equals()** metod u vašoj upoređivač klasi. Međutim, njegova primena može da određenim slučajevima poveća efikasnost dozvoljavajući programima da brzo odrede da li dva različita upoređivača nameću isti redosled.

PRIMER U SLUČAJU KLASSE GEOMETRICOBJECT

Klasa GeometricObject ne primenjuje interfejs Comparable, te se mora primeniti interfejs Comparator.

Klasa **GeometricObject** ne primenjuje interfejs **Comparable**. Da bi se ovi objekti upoređivali treba definisati neku klasu za upoređivanje, kao što je to pokazano u listingu klase **GeometricObjectComparator**

```
import java.io.Serializable;
import java.util.Comparator;

public class GeometricObjectComparator implements Comparator<GeometricObject>,
    Serializable {

    @Override
    public int compare(GeometricObject o1, GeometricObject o2) {
        double area1 = o1.getArea();
        double area2 = o2.getArea();

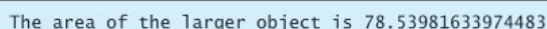
        if (area1 < area2) {
            return -1;
        } else if (area1 == area2) {
            return 0;
        } else {
            return 1;
        }
    }
}
```

Linija 8 primenjuje **Comparator<GeometricObject>**. Linija 8 redefiniše i metod **compare()** radi upoređivanja dva geometrijska objekta. Klasa takođe implementira interfejs **Serializable**. To je dobro zato što se može iskoristiti za metode uređivanja u serijalizovanim (eng. **serializable**) strukturama podataka. Da bi struktura podataka bila uspešno uređena, upoređivač (eng. **comparator**) mora da implementira interfejs **Serializable**.

PRIMENA INTERFEJSA COMPARATOR

*Listing na klase **TestComparator** prikazuje metod koji vraća veći od dva objekta. Objekti se upoređuju primenom **GeometricObjectComparator***

Listing na klase **TestComparator** prikazuje metod koji vraća veći od dva objekta. Objekti se upoređuju primenom **GeometricObjectComparator**. Na slici 1 je dat i prikaz dobijenog rezultata na displeju.



The area of the larger object is 78.53981633974483

Slika 4.1 Rezultat izvršenja programa **TestComparator**

Program kreira objekte **Rectangle** i **Circle** u linijama 5-6. One su potklase klase **GeometricObject**. Program poziva metod **max()** da bi se dobio geometrijski objekat sa većom površinom (linije 8-10).

GeometricObjectComparator je kreiran i dat metodu **max()** (linija 8) i ovaj upoređivač je korišćen u metodu **max()** za upoređenje geometrijskih objekata u liniji 15.

Comparable se koristi za poređenje objekata klase koji implementiraju interfejs **Comparable**. Interfejs **Comparator** se koristi za poređenje objekata klase koja ne primenjuje interfejs **Comparable**.

Poređenje elemenata upotrebom interfejsa **Comparable** se vrši poređenjem koristeći prirodni redosled, a upoređeni elementi upotrebom interfejsa **Comparator** vrše poređenje upotrebom upoređivača (eng. **comparator**).

```
import java.util.Comparator;

public class TestComparator {
    public static void main(String[] args) {
        GeometricObject g1 = new Rectangle(5, 5);
        GeometricObject g2 = new Circle(5);

        GeometricObject g = max(g1, g2, new GeometricObjectComparator());

        System.out.println("The area of the larger object is " + g.getArea());
    }

    public static GeometricObject max(GeometricObject g1,
        GeometricObject g2, Comparator<GeometricObject> c) {
        if (c.compare(g1, g2) > 0)
            return g1;
        else
            return g2;
    }
}
```

VIDEO: INTERFEJSI COMPARABLE I COMPARATOR

Java Sorting - Comparable vs Comparator Trevor Page Trevor Page (29,41minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 5

Statički metodi u listama i kolekcijama

STATIČKI METODI ZA MANIPULACIJU SA LISTAMA I KOLEKCIJAMA

*Klasa **Collections** sadrži statičke metode radi izvršenja zajedničkih operacija i u kolekciji i u listi.*

Često imate potrebu da sortirate neku listu. **Java Collections Framework** obezbeđuje statičke metode u klasi **Collections** koji se koriste za sortiranje liste. **Klasa Collections** takođe sadrži metode **binarySearch**, **reverse**, **shuffle**, **copy** i **fill** za liste, i metod **max**, **min**, **disjoint** i **frequency** za kolekcije, kao što je prikazano na slici 1.



Slika 5.1 Klasa Collections sadrži statičke metode za manipulaciju sa listama i kolekcijama [1]

SORTIRANJE ELEMENATA NIZA I LISTE

*Možete sortirati uporedljive elemente liste po prirodnom rasporedu sa metodom **compareTo()** interfejsa **Comparable**. Metod **reverseOrder()** uređuje element po obrnutom rasporedu.*

Sortiranje elemenata liste: Možete sortirati uporedive (eng. **comparable**) elemente liste po prirodnom rasporedu sa metodom **compareTo()** interfejsa **Comparable**. Možete takođe

da definišete upoređivač da bi sortirali elemente. Na primer, sledeći kod sortira tekstualne elemente u listi:

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.sort(list);
System.out.println(list);
```

Rezultat je **[blue, green, red]**.

Prethodni kod sortira listu po rastućem redosledu. Sortiranje po opadajućem redosledu, možete upotrebiti **`Collections.reverseOrder()`** da bi vratili objekat tipa **`Comparator`** koji uređuje elemente po obrnutom redosledu. Na primer, sledeći kod sortira listu stringova po opadajućem redosledu.

```
List<String> list = Arrays.asList("yellow", "red",
    "green", "blue");
Collections.sort(list, Collections.reverseOrder());
System.out.println(list);
```

Rezultat je **[yellow, red, green, blue]**.

Možete koristiti metod **`binarySearch()`** metod za nalaženje ključa u nekoj listi. Da bi koristili ovaj metod, lista mora da bude sortirana po rastućem redosledu. Ako ključ nije u listi, metod vraća $-(\text{insertion point} + 1)$. **`Insertionpoint`** je indeks elementa u kome bi neka stavka bila, ako bi postojala. Na primer, sledeći kod pretražuje ključeve u listi celih brojeva i listi stringova.

```
List<Integer> list1 =
    Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));
System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));
List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " +
    Collections.binarySearch(list2, "red"));
System.out.println("(4) Index: " +
    Collections.binarySearch(list2, "cyan"));
```

Rezultat je:

```
(1) Index: 2  
(2) Index: -4  
(3) Index: 2  
(4) Index: -2
```

Slika 5.2 Dobijen rezultat [1]

PREUREĐENJE ELEMENATA U LISTI METODIMA: REVERSE() I SHUFFLE()

Metodi `reverse()` i `shuffle()` svaki na svoj način, preuređuju elemente u jednoj listi, a metod `copy()` dodaje elemente druge liste u prvu listu.

Možete upotrebiti metod `reverse()` da bi obrnuli redosled elemenata u listi. Na primer, sledeći kod prikazuje [blue, green, red, yellow].

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");  
Collections.reverse(list);  
System.out.println(list);
```

Možete da upotrebite metod `shuffle(list)` da bi prerasporedili elemente liste na slučajan način. Na primer, sledeći kod meša elemente u list

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");  
Collections.shuffle(list);  
System.out.println(list);
```

Možete da upotrebite `shuffle(list, random)` metod za slučajno raspoređivanje elemenata liste korišćenjem specificiranog objekta `random` klase `Random`. Upotrebom specificiranog `Random` objekta je korisno radi generisanja liste sa identičnim rasporedom elemenata za istu originalnu listu. Na primer, sledeći kod meša elemente u listi **list**.

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");  
List<String> list2 = Arrays.asList("yellow", "red", "green", "blue");  
Collections.shuffle(list1, new Random(20));  
Collections.shuffle(list2, new Random(20));  
System.out.println(list1);  
System.out.println(list2);
```

Videćete da liste **list1** i **list2** imaju isti redosled elemenata pre i posle mešanja.

Možete koristiti metod `copy(dest, src)` da bi kopirali sve elemente iz početne liste u rezultujuću listu sa istim indeksom. Rezultujuća lista mora biti duga koliko i početna lista. Ako je duža, ostali elementi rezultujuće liste ostaće netaknuti. Na primer, sledeći kod kopira listu **list 2** u **list1**.

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("white", "black");
Collections.copy(list1, list2);
System.out.println(list1);
```

Rezultat za list1 je [white, black, green, blue]. Metod copy izvršava kopiranje u senci: samo reference elemenata iz izvorne liste su kopirane.

PREUREĐENJE ELEMENATA MODIMA NCOPIES(), FILL(), I DISJOINT()

Upotrebom metoda `nCopies()`, `fill()` i `disjoint()` mogu se preuređivati elementi jedne ili više listi, a pomoću metoda `frequency()` može se naći broj pojavljivanja elementa u kolekciji.

Možete upotrebiti metod **`nCopies(int n, Object o)`** da bi kreirali nepromenljivu listu koja ima n kopija specificiranog objekta. Na primer, sledeći kod kreira listu za pet objekata tipa **`Calendar`**.

```
List<GregorianCalendar> list1 = Collections.nCopies
    (5, new GregorianCalendar(2005, 0, 1));
```

Lista kreirana metodom **`nCopies()`** je nepromenljiva, te ne možete dodati, uklanjati ili menjati elemente u listi. Svi elementi imaju iste reference.

Možete upotrebiti metod **`fill(List list, Object o)`** da zamenite sve elemente liste sa specificiranim elementom. Na primer, sledeći kod prikazuje [black, black, black].

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.fill(list, "black");
System.out.println(list);
```

Metod **`disjoint(collection1, collection2)`** vraća **`true`** ako dve kolekcije nemaju ni jedan zajednički element. Na primer, u sledećem kodu metod **`disjoint(collection1, collection2)`** vraća **`false`**, ali metod **`disjoint(collection1, collection3)`** vraća **`true`**.

```
Collection<String> collection1 = Arrays.asList("red", "cyan");
Collection<String> collection2 = Arrays.asList("red", "blue");
Collection<String> collection3 = Arrays.asList("pink", "tan");
System.out.println(Collections.disjoint(collection1, collection2));
System.out.println(Collections.disjoint(collection1, collection3));
```

Metod **`frequency(collection, element)`** nalazi broj pojavljivanja elementa u kolekciji. Na primer, **`frequency(collection, "red")`** vraća 2 u sledećem kodu:

```
Collection<String> collection = Arrays.asList("red", "cyan", "red");  
System.out.println(Collections.frequency(collection, "red"));
```


▼ Poglavlje 6

Studija slučaja: lopte koje skaču

OPIS PROBLEMA

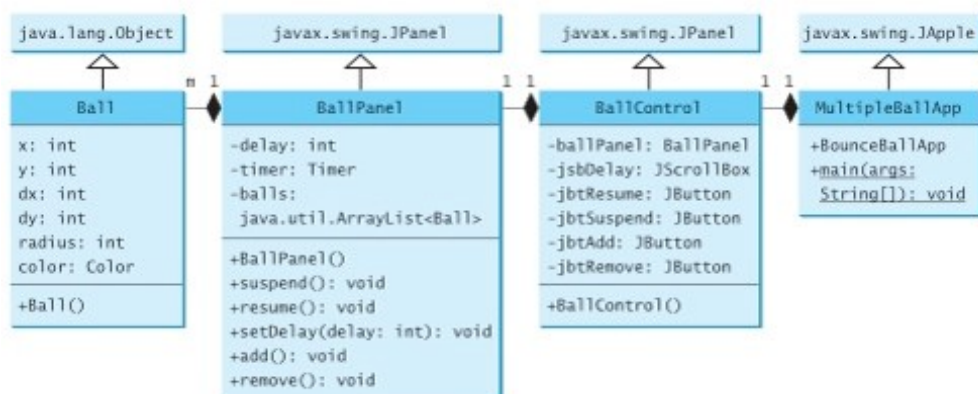
Primer prikazuje korišćenje apleta koji prikazuje lopte koje skaču i koji omogućava korisniku da doda ili da ukloni lopte.

Koristi se dva dugmeta za prekid i nastavak kretanja lopti, traka za pomeranje (enl. scroll bar) za kontrolu brzine lopti, kao i dugmad sa +1 i -1 sa kojima se dodaje ili izbacuje po jedna lopta (slika 1). Koristi se niz za smeštanje lopti. Na početku lista je prazna. Kada se lopta kreira, ona se dodaje na kraj liste. Kada se uklanja lopta, uklanja se poslednja lopta u listi niza. Svaka lopta ima svoje stanje: lokaciju, boju i smer kretanja. Možete kreirati klasu **Ball** sa odgovarajućim atributima radi smeštaja ovih informacija. Kada se kreira lopta, ona počinje kretanje iz gornjeg levog ugla okvira i kreće se nadole i desno. Boja lopte se dodeljuje slučajno.

Klasa **BallPanel** je odgovorna za prikaz lopte i klasu **BallControl** za kontrolu kretanja lopti (elemenata liste). Klasa **MultipleBallApp** postavlja **BallControl** u aplet. Veza ovih klasa je prikazana na slici 2.



Slika 6.1 Pritiskom na dugme +1 i -1 dodaje se ili uklanja po jedna lopta [1]



Slika 6.2 MultipleBallApp sadrži BallControl, BallControl sadrži BallPanel, a BallPanel sadrži Ball [1]

LISTING PRIKAZANIH KLASA

Ovaj program upotrebljava klasu ArrayList za uskladišćenje lopti.

Listing klase **MultipleBallApp**:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.AdjustmentEvent;
import java.awt.event.AdjustmentListener;
import java.util.ArrayList;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JScrollBar;
import javax.swing.Timer;

public class MultipleBallApp extends JApplet {

    public MultipleBallApp() {
        add(new BallControl());
    }

    class BallControl extends JPanel {

        private BallPanel ballPanel = new BallPanel();
        private JButton jbtSuspend = new JButton("Suspend");
        private JButton jbtResume = new JButton("Resume");
        private JButton jbtAdd = new JButton("+1");
        private JButton jbtSubtract = new JButton("-1");
        private JScrollBar jsbDelay = new JScrollBar();

        public BallControl() {
            // Grupa lopti u panelu
            JPanel panel = new JPanel();
            panel.add(jbtSuspend);
            panel.add(jbtResume);
            panel.add(jbtAdd);
            panel.add(jbtSubtract);

            // Dodavanje lopte i dugmadi u panel
            ballPanel.setBorder(
                new javax.swing.border.LineBorder(Color.red));
            jsbDelay.setOrientation(JScrollBar.HORIZONTAL);
            ballPanel.setDelay(jsbDelay.getMaximum());
            setLayout(new BorderLayout());
```

```

add(jsbDelay, BorderLayout.NORTH);
add(ballPanel, BorderLayout.CENTER);
add(panel, BorderLayout.SOUTH);

// Registracija osluškivača
jbtSuspend.addActionListener(new Listener());
jbtResume.addActionListener(new Listener());
jbtAdd.addActionListener(new Listener());
jbtSubtract.addActionListener(new Listener());
jsbDelay.addAdjustmentListener(new AdjustmentListener() {
    @Override
    public void adjustmentValueChanged(AdjustmentEvent e) {
        ballPanel.setDelay(jsbDelay.getMaximum() - e.getValue());
    }
});
}

class Listener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == jbtSuspend) {
            ballPanel.suspend();
        } else if (e.getSource() == jbtResume) {
            ballPanel.resume();
        } else if (e.getSource() == jbtAdd) {
            ballPanel.add();
        } else if (e.getSource() == jbtSubtract) {
            ballPanel.subtract();
        }
    }
}

class BallPanel extends JPanel {

    private int delay = 10;
    private ArrayList<Ball> list = new ArrayList<Ball>();

    // Kreiranje tajmera sa početnim kašnjenjem
    protected Timer timer = new Timer(delay, new ActionListener() {
        @Override
        /**
         * Obrađuje događaj akcije
         */
        public void actionPerformed(ActionEvent e) {
            repaint();
        }
    });

    public BallPanel() {
        timer.start();
    }
}

```

```

    public void add() {
        list.add(new Ball());
    }

    public void subtract() {
        if (list.size() > 0) {
            list.remove(list.size() - 1); // Uklanjanje poslednje lopte
        }
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        for (int i = 0; i < list.size(); i++) {
            Ball ball = (Ball) list.get(i); // Uzima loptu
            g.setColor(ball.color); // Set ball color

            // Provera granica
            if (ball.x < 0 || ball.x > getWidth()) {
                ball.dx = -ball.dx;
            }

            if (ball.y < 0 || ball.y > getHeight()) {
                ball.dy = -ball.dy;
            }

            // Podešavanje pozicije lopte
            ball.x += ball.dx;
            ball.y += ball.dy;
            g.fillOval(ball.x - ball.radius, ball.y - ball.radius,
                ball.radius * 2, ball.radius * 2);
        }
    }

    public void suspend() {
        timer.stop();
    }

    public void resume() {
        timer.start();
    }

    public void setDelay(int delay) {
        this.delay = delay;
        timer.setDelay(delay);
    }
}

class Ball {

    int x = 0;

```

```
int y = 0; // Trenutna pozicija lopte
int dx = 2; // Inkrement pomeranja lopte po x-koordinati
int dy = 2; // Inkrement pomeranja lopte po y-koordinati
int radius = 5; // Ball radius
Color color = new Color((int) (Math.random() * 256),
                        (int) (Math.random() * 256), (int) (Math.random() * 256));
    }
}
```

Lista niza je kreirana za smeštaj lopti (linija 82). Kada korisnik pritisne +1 dugme, stvara se nova lopta i dodaje u listu niza (linija 100). Kada korisnik klikne -1 dugme, poslednja lopta u listi nizu se uklanja (linija 105).

Metod **paintComponent()** u klasi **BallPanel** dobija svaku loptu u listi nizi, podešava poziciju lopte (linije 110-128), i boji ih (linija 129).

Ovaj program upotrebljava klasu **ArrayList** za uskladišćenje lopti. Program radi i kada se klasa **ArrayList** zameni sa **LinkedList**, ali je efikasniji kada koristi **ArrayList** u ovom primeru.

▼ Poglavlje 7

Klase Vector i Stack

KLASA VECTOR

Klasa Vector je podklasa klase ArrayList u Java API.

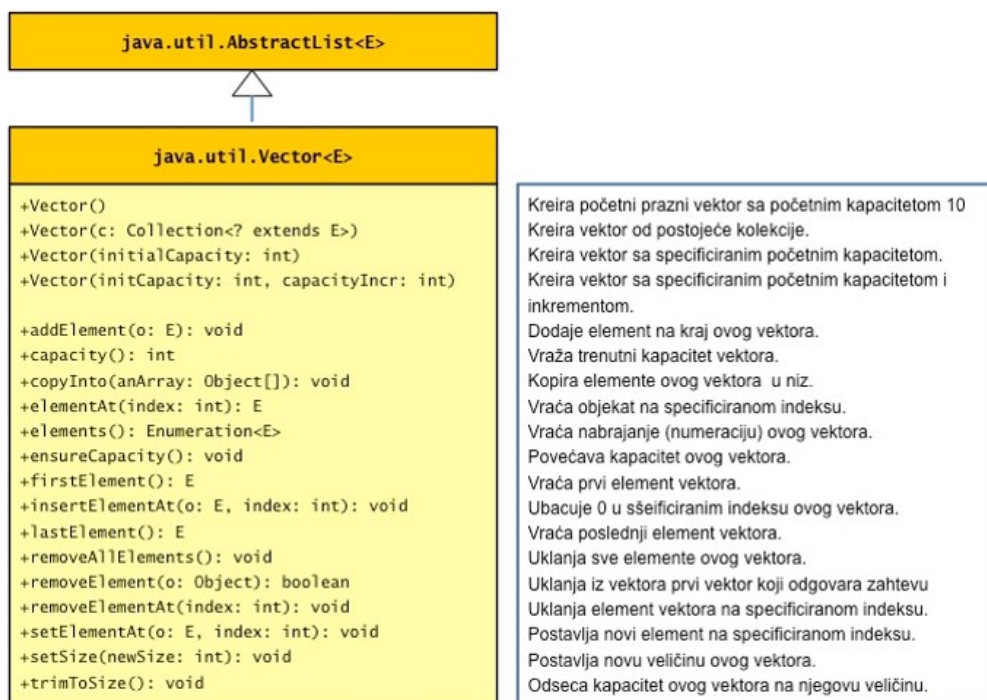
Java Collections Framework je uveden u sistemu Java 2. Pre ovoga, Java je podržavala nekoliko struktura podataka, kao što su i klase Vector i Stack. One su redizajnirane, ali su njihovi metodi zadržani radi kompatibilnosti sa starim programima.

Klasa Vector je ista kao i klasa ArrayList, sem što poseduje i metode za sinhronizaciju pristup i promenu vektora., tj. objekata klase Vector. Ovi metodi sprečavaju oštećenje podataka kada se pristupa vektoru i kada se menja dejstvo dva ili više simultana toka. Za aplikacije koje ne zahtevaju sinhronizaciju, upotreba klase ArrayList je efikasnija od upotrebe klase Vector.

Klasa Vector proširuje klasi ArrayList, tj. predstavlja njenu potklasu (slika 1). Ona sadrži i metode koje je sadržala i originalna klasa Vector koja je definisana pre pojave Java 2.

0.

Najveći broj metoda su slične metodima List interfejsa. Ovi metodi su korišćeni pre pojave Java Collection Framework-a.



Slika 7.1 Od Jave 2, klasa Vector je potklasa AbstractList klase i zadržava sve metode iz originalne klase Vector [1]

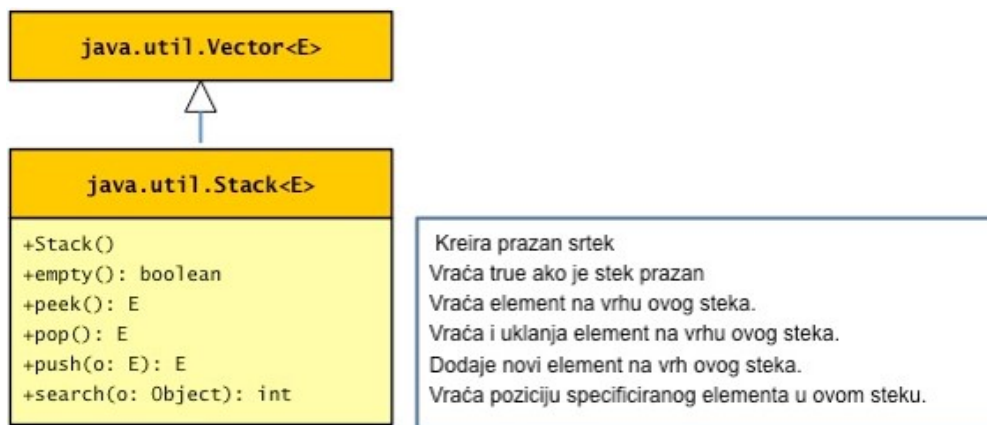
KLASA STACK

Klasa Stack je podklasa klase Vector u Java API

U **Java Collection Framework**, klasa **Stack** se primenjuje kao podklasa klase **Vector**, kao što je prikazano na UML dijagramu na slici 2.

Klasa Stack se koristila i pre Jave 2. Metodi prikazani na slici 2 su se takođe koristili pre Jave 2. Metod **empty()** je isti kao metod **isEmpty()**. Metod **peak()** pristupa elementu na vrhu steka, bez njegovog uklanjanja. Metod **pop()** uklanja element na vrhu steka i vraća taj element. Metod **push(Object element)** dodaje specificiran element na stek. Metod **search(Object element)** proverava da li je specificiran element u steku.

UML dijagram klase **Stack**:



Slika 7.2 Klasa Stack nasleđuje klasu Vector da bi obezbedila strukturu podataka koja radi na principu LIFO (eng. last in, first out) - "poslednji ušao, prvi izašao" [1]

VIDEO: POVEZANE LISTE - PRIMER

Java Programming - Lists, Stacks, Queues, and Priority Queues - LinkedList Example (15,51)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 8

Redovi i prioritetni redovi

INTERFEJS QUEUE

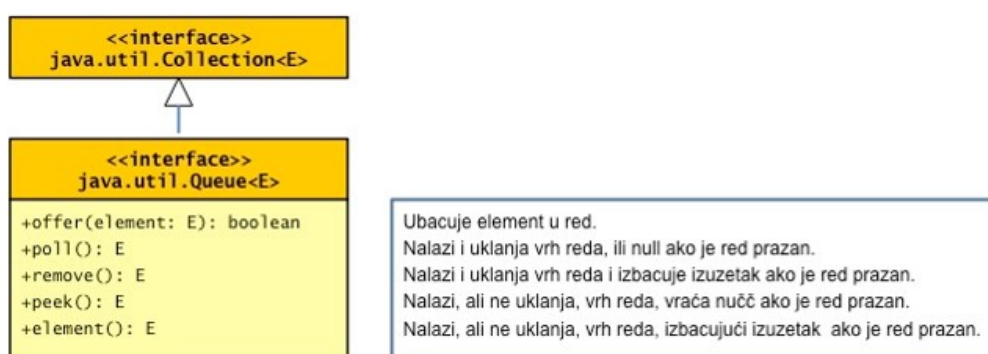
Red (eng. queue) je struktura podataka koja primenjuje princip "prvi ušao, prvi izašao", elementi se uklanjaju sa kraja reda, a dodaju na početak reda.

Red (eng. **queue**) je struktura podataka koja primenjuje princip "prvi ušao, prvi izašao", FIFO (eng. first in, first out). Elementi se dodaju na kraj reda, a uklanjaju se sa početka reda. Kod prioritetnog reda (eng. **priority queue**) elementima se dodeljuje prioritet. Kada se pristupa elementima, prvo se uklanja iz reda element koji ima najveći prioritet.

Interfejs Queue nasleđuje interfejs **java.util.Collection** sa dodatnim operacijama ubacivanja i nalaženja elemenata, kao što je pokazano na slici 1.

Metod **offer()** se upotrebljava radi dodavanja elementa u red. Metod je sličan metodu **add()** interfejsa **Collection**, ali je metod **offer()** bolji za redove. Metodi **poll()** i **remove()** su slični, sem što **poll()** vraća **null** ako je red prazan, a metod **remove()** izbacuje izuzetak. Metodi **peek()** i **element()** su slični, sem što **peek()** vraća **null**, ako je red prazan, dok metod **element()** izbacuje izuzetak.

UML dijagram interfejsa **Queue**.



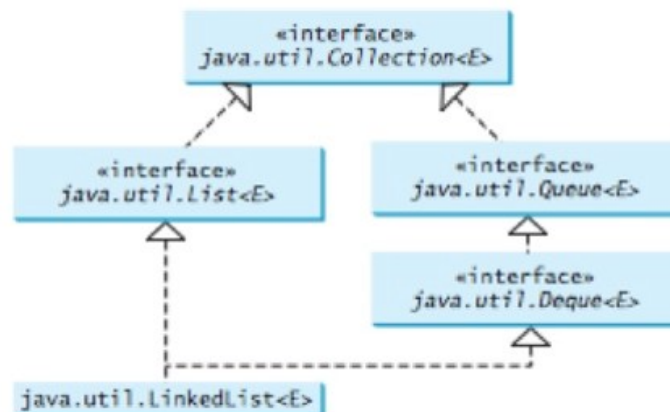
Slika 8.1 Metodi interfejsa Queue [1]

DEQUEUE I LINKEDLIST

*Klasa **LinkedList** implementira interfejs **Deque**, koji nasleđuje interfejs **Queue**, je idealna klasa za operacije sa redovima, jer menja elemente na oba kraja reda.*

Klasa **LinkedList** implementira interfejs **Deque**, koji nasleđuje interfejs **Queue**, kao što je prikazano na slici 2. Prema tome, vi možete koristiti i klasu **LinkedList** da kreirate red. **LinkedList** je idealna klasa za operacije sa redovima jer je efikasna kod ubacivanja i uklanjanja elemenata sa oba kraja liste.

Interfejs Deque podržava ubacivanje i uklanjanje elemenata na oba kraja reda. Interfejs **Deque** nasleđuje interfejs **Queue** sa dodatnim metodima za ubacivanje i uklanjanje elemenata na oba kraja reda. Metodi **addFirst()**, **removeFirst()**, **addLast()**, **removeLast()**, **getFirst()** i **getLast()** su definisani u interfejsu **Deque**.



Slika 8.2 Klasa **LinkedList** primenjuje interfejse **List** i **Deque** [1]

Listing klase **TestQueue** pokazuje primer upotrebe reda za skladišćenje stringova. Linija 6 kreira red upotrebom klase **LinkedList**. Četiri stringa se dodaju u red u linijama 7-10. Metod **size()** definisan u interfejsu **Collection** vraća broj elemenata u redu (linija 12). Metod **remove()** nalazi i uklanja element na vrhu reda (linija 13).

```
import java.util.Queue;

public class TestQueue {

    public static void main(String[] args) {
        Queue<String> queue = new java.util.LinkedList<String>();
        queue.offer("Oklahoma");
        queue.offer("Indiana");
        queue.offer("Georgia");
        queue.offer("Texas");

        while (queue.size() > 0) {
            System.out.print(queue.remove() + " ");
        }
    }
}
```

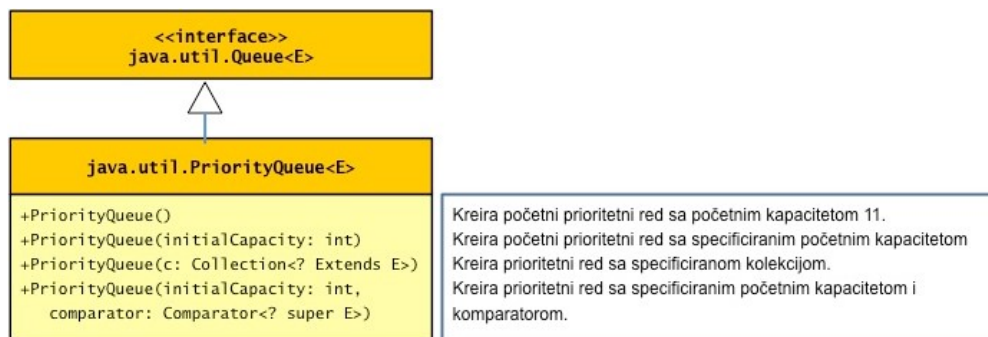
Oklahoma Indiana Georgia Texas

Slika 8.3 Dobijen rezultat [1]

KLASA PRIORITYQUEUE

Prioritetni redovi imaju elemente poređane u skladu sa njihovim prirodnim rasporedom, upotrebom interfejsa Comparable, element sa najmanjom vrednošću ima najveći prioritet.

Klasa PriorityQueue predstavlja prioritetni red, kao što je prikazano na slici 4. Po pravilu, prioritetni redovi imaju elemente poređane u skladu sa njihovim prirodnim rasporedom, upotrebom interfejsa **Comparable**. Element sa najmanjom vrednošću ima najveći prioritet, te se i prvi uklanja iz reda. Ako nekoliko elemenata ima isti najviši prioritet, oni se uklanjaju po slučajnom redosledu. Možete da definišete redosled upotrebom objekta tipa **Comparator** u konstruktoru **PriorityQueue(initialCapacity, comparator)**.



Slika 8.4 Klasa PriorityQueue primenjuje prioritetni red [1]

VIDEO: STEKOVI I REDOVI

Derek Banas: Stacks and Queues (16,4 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 9

Vežba – Pokazni primeri

PRIMER 1

Cilj ovog primer je provežbavanje korišćenja ArrayListe

Kreirati ArrayList-u stringova i u nju dodati 4 elementa. Prikazati rad metoda: indexOf, isEmpty, size, contains, get, remove i set. Takođe prikažite sadržaj liste koristeći for petlju, foreach petlju kao i iterator.

Potrebno vreme: 10 minuta

Main.java:

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

public class Main {

    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Item1");
        list.add("Item2");
        list.add(2, "Item3");
        list.add("Item4");
        System.out.println("Nasa lista sadrzi sledece elemente: "
            + list);

        // Proveravamo index elemneta
        int pos = list.indexOf("Item2");
        System.out.println("Index elementa Item2 je: " + pos);

        // Proveravamo da li je prazna array lista
        boolean check = list.isEmpty();
        System.out.println("Provera da li je prazna lista : " + check);

        // Proveravamo velicinu liste
        int size = list.size();
        System.out.println("Velicina liste je: " + size);

        // da li element postoji u listi
        boolean element = list.contains("Item5");
        System.out
            .println("Provera da li element Item5 postoji u listi: ")
```

```
        + element);

// uzimanje elementa na poziciji
String item = list.get(0);
System.out.println("Element na poziciji 0 je: " + item);

// Ispisivanje svih elemenata liste
System.out.println("Svi elementi liste korsiteci petlju su");
for (int i = 0; i < list.size(); i++) {
    System.out.println("Index: " + i + " - Vrednost: " + list.get(i));
}

// Koriscenje foreach petlje
System.out.println("Koriscenje foreach petlje");
for (String str : list) {
    System.out.println("Element je: " + str);
}

// Koriscenje iteratora za stampanje
System.out.println("Koriscenje iteratora za stampanje");
for (Iterator<String> it = list.iterator(); it.hasNext();) {
    System.out.println("Vrednost elementa je: " + it.next());
}

// Zamena elementa u listi
list.set(1, "NewItem");
System.out.println("Lista nakon zamene je: " + list);

// Brisanje elementa na poziciji 0
list.remove(0);

// Brisanja elementa sa vrednoscu Item3
list.remove("Item3");

System.out.println("Krajnji rezultat liste je: " + list);

// Pretvaranje ArrayListe u obican niz
String[] simpleArray = list.toArray(new String[list.size()]);
System.out.println("Nakon pretvaranja ArrayListe u niz. Vrednost kreiranog
niza je: "
        + Arrays.toString(simpleArray));
    }
}
```

PRIMER 1 - OBJAŠNJENJE

Objašnjenje primera 1

Objašnjenje:

Ovaj zadatak ima za cilj demonstriranje korišćenja ArrayList-e.

- Metoda `indexOf(item)` pronalazi indeks pojavljivanja određenog objekta *item* u listi
- Metoda `isEmpty()` je metoda koju korisimo za proveru da li lista sadrži elemente ili je prazna
- Metoda `size()` određuje broj elemenata liste
- Metoda `contains(item)` ispituje da li se objekat *item* nalazi u listi
- Metoda `get(i)` vraća objekat sa pozicije *i* iz liste
- Metoda `remove(item)` uklanja objekat *item* iz liste
- Metoda `set(i, item)` služi za zamenu objekta na poziciji *i* objektom *item*
- Svaka kolekcija ima objekat tipa `Iterator` koji se koristi da bi se pristupilo svakom od elemenata kolekcije. Ovde koristimo iterator za prikaz sadržaja liste.

PRIMER 2

Cilj ovog primera je provežbavanje korišćenja `Comparable` interfejsa

Napraviti klasu `Student` (indeks, ime, prezime). Napravite 50 instanci klase `student` sa random podacima i smestite ih u `ArrayList`-u. Sortirati sve studente u listi po broju indeksa od najmanjeg ka najvećem. **Napomena: koristiti interfejs `Comparable`**

Potrebno vreme: 10 minuta

Klasa `Main`:

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

public class Main {

    public static void main(String[] args) {
        new Main();
    }

    public Main() {
        ArrayList<Student> studenti = new ArrayList<Student>();

        for (int i = 0; i < 20; i++) {
            Student s = new Student(Util.getRandomIndeks(), Util.getRandomIme(),
Util.getRandomIme());
            studenti.add(s);
        }

        Collections.sort(studenti);
        for (Student s : studenti) {
            System.out.println(s);
        }

    }

}
```

Klasa Student:

```
public class Student implements Comparable<Student> {

    private int indeks;
    private String ime;
    private String prezime;

    public Student() {
    }

    public Student(int indeks, String ime, String prezime) {
        this.indeks = indeks;
        this.ime = ime;
        this.prezime = prezime;
    }

    public int getIndeks() {
        return indeks;
    }

    public void setIndeks(int indeks) {
        this.indeks = indeks;
    }

    public String getIme() {
        return ime;
    }

    public void setIme(String ime) {
        this.ime = ime;
    }

    public String getPrezime() {
        return prezime;
    }

    public void setPrezime(String prezime) {
        this.prezime = prezime;
    }

    @Override
    public int compareTo(Student o) {
        return Integer.valueOf(indeks).compareTo(o.getIndeks());
    }

    @Override
    public String toString() {
        return "Student{" + "indeks=" + indeks + ", ime=" + ime + ", prezime=" +
prezime + '}';
    }
}
```

Klasa Util:

```
import java.util.Random;

public class Util {

    static Random r = new Random();

    public static int getRandomIndeks() {
        return r.nextInt(2700);
    }

    public static String getRandomIme() {
        String s = "";
        for (int i = 0; i < 12; i++) {
            s += (char) (r.nextInt(25) + 'a');
        }
        return s;
    }
}
```

PRIMER 2 - OBJAŠNJENJE

Objašnjenje primera 2

Objašnjenje:

Da bi sortiranje bilo moguće, potrebno je da definišemo način upoređivanja objekata klase Student. U ovom slučaju definišemo da su studenti jednaki ukoliko imaju isti broj indeksa. Da bi poredili objekte neophodno je definisati klasu koja implementira interfejs Comparable<T>. On definiše compareTo() metod, koji služi za upoređivanje dva elementa klase koja implementira interfejs Comparable<T>.

Metod compareTo() vraća nenegativnu vrednost ako element1 je manji od element2, a pozitivnu vrednost ako je element1 veći od element2 i vraća nulu ako su jednaki. Ova osobina se direktno koristi u statičkoj metodi sort iz klase Collections koja vrši sortiranje:

```
Collections.sort(studenti);
```

PRIMER 3

Cilj ovog primera je provežbavanje korišćenja Stacka u Javi.

Napraviti klasu Student (indeks, ime, prezime), a potom kreirati dve instance ove klase. Staviti obe instance na stack koristeći push metodu, a potom prikažite rad LIFO stacka.

Potrebno vreme: 5 minuta

Objašnjenje:

Ovaj zadatak ima za cilj demonstriranje metoda push i pop klase Stack. Stack je struktura koja koristi pristup LIFO (Last In First Out), tako da metoda push(item) dodaje objekat item na vrh steka. Metoda pop() sa steka uklanja poslednje dodat objekat.

Klasa Main:

```
import java.util.Stack;

public class Main {

    public static void main(String[] args) {
        new Main();
    }

    public Main() {
        Stack stack = new Stack();
        Student milos = new Student(1650, "Milos", "Peric");
        Student pera = new Student(1630, "Pera", "Peric");
        stack.push(milos);
        stack.push(pera);
        System.out.println(stack.pop());
        System.out.println(stack.pop());

        try {
            System.out.println(stack.pop());
        } catch (java.util.EmptyStackException e) {
            stack.push(milos);
            System.out.println(stack.pop());
        }
    }
}
```

Klasa Student;

```
public class Student implements Comparable<Student> {

    private int indeks;
    private String ime;
    private String prezime;

    public Student() {
    }

    public Student(int indeks, String ime, String prezime) {
        this.indeks = indeks;
        this.ime = ime;
        this.prezime = prezime;
    }

    public int getIndeks() {
        return indeks;
    }
}
```



```

    public void setIndeks(int indeks) {
        this.indeks = indeks;
    }

    public String getIme() {
        return ime;
    }

    public void setIme(String ime) {
        this.ime = ime;
    }

    public String getPrezime() {
        return prezime;
    }

    public void setPrezime(String prezime) {
        this.prezime = prezime;
    }

    @Override
    public int compareTo(Student o) {
        return Integer.valueOf(indeks).compareTo(o.getIndeks());
    }

    @Override
    public String toString() {
        return "Student{" + "indeks=" + indeks + ", ime=" + ime + ", prezime=" +
prezime + '}';
    }
}

```

PRIMER 4

Cilj ovog zadatka je provežbavanje implementacije sopstvenog stacka u Javi.

Koristeći ArrayList-u napraviti sopstvenu implementaciju LIFO Stack-a i demonstrirati rad Stack-a isto kao i u prethodnom zadatku.

Potrebno vreme: 5 minuta

Objašnjenje:

Ovaj zadatak ima za cilj demonstriranje implementacije push i pop metoda klase Stack koristeći se ArrayList-om. Kao što je pomenuto, Stack je struktura koja koristi pristup LIFO (Last In First Out), tako da u implementaciji metoda push(item) koristimo metodu add koja dodaje objekat item na kraj liste. Metoda pop() sa steka uklanja poslednje dodat objekat, u ovom slučaju objekat sa poslednjeg elementa u listi na poziciji size() - 1

```
public class Main {

    public static void main(String[] args) {
        new Main();
    }

    public Main() {
        StackLista stack = new StackLista();
        Student milos = new Student(1650, "Milos", "Peric");
        Student pera = new Student(1630, "Pera", "Peric");
        stack.push(milos);
        stack.push(pera);
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

Klasa Student:

```
public class Student implements Comparable<Student> {

    private int indeks;
    private String ime;
    private String prezime;

    public Student() {
    }

    public Student(int indeks, String ime, String prezime) {
        this.indeks = indeks;
        this.ime = ime;
        this.prezime = prezime;
    }

    public int getIndeks() {
        return indeks;
    }

    public void setIndeks(int indeks) {
        this.indeks = indeks;
    }

    public String getIme() {
        return ime;
    }

    public void setIme(String ime) {
        this.ime = ime;
    }

    public String getPrezime() {
        return prezime;
    }
}
```

```
public void setPrezime(String prezime) {
    this.prezime = prezime;
}

@Override
public int compareTo(Student o) {
    return Integer.valueOf(indeks).compareTo(o.getIndeks());
}

@Override
public String toString() {
    return "Student{" + "indeks=" + indeks + ", ime=" + ime + ", prezime=" +
prezime + '}';
}
}
```

Klasa StackLista:

```
import java.util.ArrayList;

public class StackLista {

    ArrayList<Object> objekti = new ArrayList<Object>();

    public void push(Object e) {
        objekti.add(e);
    }

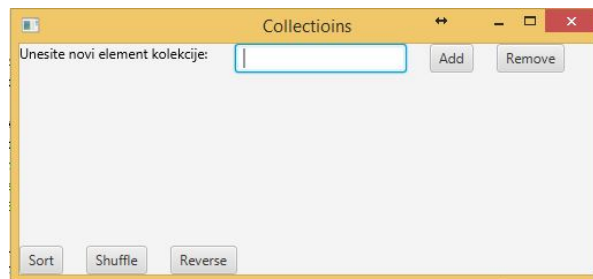
    public Object pop() {
        Object o = objekti.get(objekti.size() - 1);
        objekti.remove(o);
        return o;
    }
}
```

PRIMER 5

Cilj ovog zadatka je provežbavanje grafičkog prikazivanja liste u JaviFX

Napraviti grafički prikaz dodavanja elementa u listu, sortiranja, mešanja i okretanja redosleda elemenata liste koristeći LinkedList.

Potrebno vreme: 10 minuta



Slika 9.1 Prikaz forme za unos kolekcija

Klasa Controller:

```
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;

public class Controller {

    private LinkedList<Integer> list;

    public Controller() {
        list = new LinkedList<>();
    }

    public boolean isDuplicate(Integer val) {
        if (list.isEmpty()) {
            return false;
        }
        for (Integer tmp : list) {
            if (tmp.equals(val)) {
                return true;
            }
        }
        return false;
    }

    public void addElement(Integer val) {
        if (!isDuplicate(val)) {
            list.addLast(val);
        }
    }

    public void removeLast() {
        list.removeLast();
    }

    public void sort() {
        Collections.sort(list);
    }

    public void shuffle() {
        Collections.shuffle(list);
    }
}
```

```

    }

    public void reverse() {
        Collections.reverse(list);
    }

    public String showAsString() {
        //System.out.println("Usao showAsString");
        Iterator<Integer> i = list.iterator();
        String result = "";
        while (i.hasNext()) {
            result += i.next() + " ";
        }
        return result;
    }

    public boolean listEmty() {
        return list.isEmpty();
    }
}

```

Klasa CollectionsDemo:

```

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javax.swing.JOptionPane;

public class CollectionsDemo extends Application {

    private Controller controller = new Controller();
    private Text tElem = new Text("Unesite novi element kolekcije: ");
    private TextField tfElem = new TextField();
    private Button addBtn = new Button("Add");
    private Button removeBtn = new Button("Remove");
    private Button sortBtn = new Button("Sort");
    private Button shuffleBtn = new Button("Shuffle");
    private Button reverseBtn = new Button("Reverse");
    private Text display = new Text();

    @Override
    public void start(Stage primaryStage) {

```

```

    HBox top = new HBox(20);
    HBox bottom = new HBox(20);
    setDisplay();
    addListeners();
    top.getChildren().addAll(tElem, tfElem, addBtn, removeBtn);
    bottom.getChildren().addAll(sortBtn, shuffleBtn, reverseBtn);

    BorderPane root = new BorderPane();

    root.setTop(top);
    root.setBottom(bottom);
    root.setCenter(display);

    Scene scene = new Scene(root, 500, 200);

    primaryStage.setTitle("Collectionins");
    primaryStage.setScene(scene);
    primaryStage.show();
}

public void setDisplay() {
    display.setFont(Font.font("Arial", FontWeight.BOLD, 32));
}

public void addListeners() {
    addBtn.setOnAction(new EventHandler<ActionEvent>() {

        @Override
        public void handle(ActionEvent t) {
            String text = tfElem.getText();
            try {
                if (text != null && !text.isEmpty()) {
                    Integer el = Integer.parseInt(text);
                    controler.addElement(el);
                    display.setText(controler.showAsString());
                    tfElem.setText("");
                }
            } catch (NumberFormatException ex) {
                JOptionPane.showMessageDialog(null, "Element mora biti ceo
broj. ");
            }
        }
    });

    removeBtn.setOnAction(new EventHandler<ActionEvent>() {

        @Override
        public void handle(ActionEvent t) {
            if (!controler.listEmty()) {
                controler.removeLast();
                display.setText(controler.showAsString());
            }
        }
    });
}

```

```
});

sortBtn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent t) {
        controler.sort();
        display.setText(controler.showAsString());
    }
});

shuffleBtn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent t) {
        controler.shuffle();
        display.setText(controler.showAsString());
    }
});

reverseBtn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent t) {
        controler.reverse();
        display.setText(controler.showAsString());
    }
});
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    launch(args);
}
}
```

PRIMER 5 - OBJAŠNJENJE

Objašnjenje primera 5

Objašnjenje:

Ovaj zadatak ima za cilj demonstriranje metoda za rad sa LinkedList-om:

- metod isDupliacate(item) ispituje da li u listi postoji objekat item i vraća true ukoliko postoji i false ukoliko ne postoji
- metod addElement(item) dodaje element na kraj liste ukoliko ne postoji. Na ovaj način sprečavamo dodavanje duplikata u listu

- metod `removeLast()` uklanja poslednji dodat element iz liste koristeći postojeći metod iste signature
- metode `sort`, `shuffle`, `reverse` služe za sortiranje, mešanje elemenata i okretanje liste elemenata koristeći metode iste signature iz klase `Collections`

PRIMER 6

Uklanjanje elemenata iz `LinkedList`

Napraviti listu čiju su elementi tipa `String`.

Dodati proizvoljan broj elemenata, ispisati listu.

Nakon ispisa,, odabrati jedan element koji ćete ukloniti i ispisati novonastalu listu.

Potrebno vreme: 5 minuta

OUTPUT:

Before Remove:

Item1

Item2

Item3

Item4

Item5

After Remove:

Item1

Item2

Item3

Item5

```
import java.util.LinkedList;

public class RemoveExample {

    public static void main(String[] args) {

        // kreiramo listu
        LinkedList<String> linkedlist = new LinkedList<String>();

        // dodajemo elemente u listu
        linkedlist.add("Item1");
        linkedlist.add("Item2");
        linkedlist.add("Item3");
        linkedlist.add("Item4");
        linkedlist.add("Item5");

        // ispis liste
        System.out.println("Before Remove:");
```



```
    for (String str : linkedlist) {  
        System.out.println(str);  
    }  
  
    // uklanjanje elementa iz liste  
    linkedlist.remove("Item4");  
  
    // ispis liste nakon uklanjanja elementa  
    System.out.println("\nAfter Remove:");  
    for (String str2 : linkedlist) {  
        System.out.println(str2);  
    }  
}  
}
```

PRIMER 7

Primer sa vektorom

Kreirati vektor sa elementima tipa String.

dajte elemente u vektor

vec.addElement("Apple");

ispišite veličinu vektora metodom vec.size()

ispišite kapacitet vektora vec.capacity()

Potrebno vreme: 5 minuta

OUTPUT:
Size is: 4
Default capacity increment is: 4
Size after addition: 7
Capacity after increment is: 8
Elements are:
Apple Orange Mango Fig fruit1 fruit2 fruit3

```
import java.util.*;  
  
public class VectorExample {  
  
    public static void main(String args[]) {  
        /*inicijalno kreiramo vektor capacity(size) 2 */  
        Vector<String> vec = new Vector<String>(2);  
  
        /* dodajemo elemente u vektor*/  
    }  
}
```

```
vec.addElement("Apple");
vec.addElement("Orange");
vec.addElement("Mango");
vec.addElement("Fig");

/* proveramo velicinu i kapacitet - capacityIncrement*/
System.out.println("Size is: " + vec.size());
System.out.println("Default capacity increment is: " + vec.capacity());

vec.addElement("fruit1");
vec.addElement("fruit2");
vec.addElement("fruit3");

/*velicina i kapacitet nakon novog unosa*/
System.out.println("Size after addition: " + vec.size());
System.out.println("Capacity after increment is: " + vec.capacity());

/*prikaz elemenata vektora*/
Enumeration en = vec.elements();
System.out.println("\nElements are:");
while (en.hasMoreElements()) {
    System.out.print(en.nextElement() + " ");
}
}
```

▼ Poglavlje 10

Vežba – Zadaci za individualni rad

ZADACI ZA INDIVIDUALAN RAD (1 - 6)

Cilj ovih zadataka da student provežba šta je naučio na predavanjima i vežbama

Student samostalno radi ove zadatke, uz eventualu pomoć asistenta. Zadatake koje ne završi za 3 časa, student može da radi kod kuće. Data vremena su orijentaciona, i za neke studente mogu biti kratka, a za neke duge.

Zadatak 1 (15 minuta):

Napraviti klasu automobil (marka, model, godiste). Napraviti listu od 50 instanci klase Automobil, a potom sortirajte listu po godištu automobila.

Zadatak 2 (15 minuta):

Napraviti sopstvenu implementaciju Stack-a i iskoristite je za igricu pogađanja. Igrač vuče pitanja sa Stack-a sve dok ne odgovori na sva pitanja. Ukoliko pogreši igrač je izgubio.

Zadatak 3 (15 minuta):

Napravite implementaciju HighScore-a prethodne igrice. Dodati vreme na igricu tako da na kraju igrice onaj ko pobedi sa najmanje potrošenog vremena bude prvi. Serializovanu listu rezultata treba čitati i upisivati u fajl. .

Zadatak 4 (15 minuta):

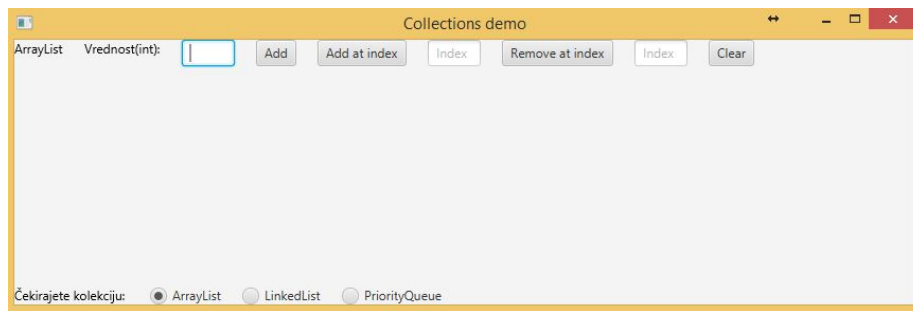
Napraviti klasu CD (autor, brojPesama, zanr). Napraviti listu od 50 instanci klase CD-a a potom sortirajte listu po autoru

Zadatak 5 (15 minuta):

Na prethodni zadatak dodati metodu za pretraživanje CD-ova po autoru i prikazati njenu upotrebu.

Zadatak 6 (15 minuta):

Proširiti zadatak 5 iz pokažnih vežbi tako da radi sa različitim tipovima listi kao na sledećoj slici:



Slika 10.1 Slika uz zadatak 6 [1]

ZADACI ZA INDIVIDUALNI RAD (7 - 11)

Proverite novostečena znanja u ovoj lekciji rešavanjem zadataka od 7. do 11.

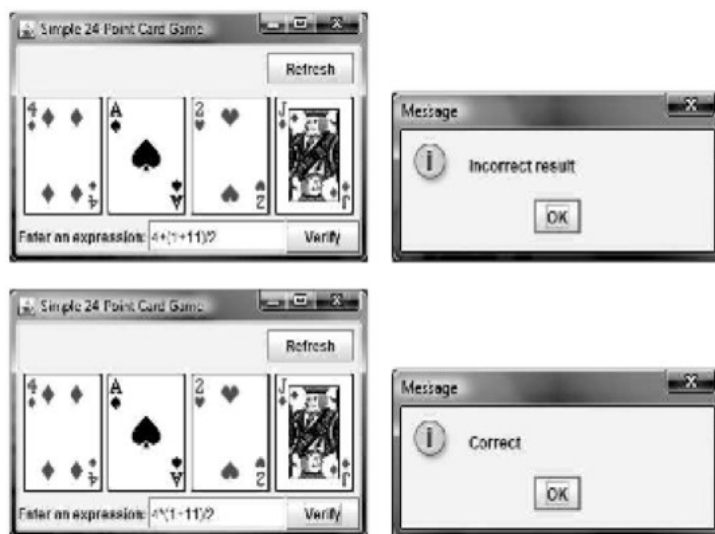
Zadatak 7 (15 minuta): Uklanjanje najveće lopte: Promenite program MultipleBallApp da bi poluprečnici lopti bili slučajno dodeljeni loptama veličine od 2 do 20, u vreme kreiranja lopte. Kada se klikne dugme 1, uklanja se najveća lopta (Upotrebite PriorityQueue za memorisanje lopti).

Zadatak 8 (15 minuta): Primena operacija skupova na prioritetnom redu: Kreirajte dva prioritetna reda {"George", "Jim", "John", "Blake", "Kevin", "Michael"} i {"George", "Katie", "Kevin", "Michelle", "Ryan"} i nađite njihovu uniju, razliku i presek.

Zadatak 9 (15 minuta): Kloniranje PriorityQueue: Definišite klasu MyPriorityQueue koja nasleđuje klasu PriorityQueue radi implementiranja interfejsa Cloneable i implementiranja metoda clone() za kloniranje prioritetnog reda.

Zadatak 10 (15 minuta): Igra sa 24 karte: Uzimaju se četiri karte od 52 karte, kao što je pokazano na slici V1. Ne koriste se džokeri. Svaka karta predstavlja jedan broj. As, kralj, kraljica i žandar predstavljaju 1, 13, 12 i 11, posebno. Možete kliknuti dugme Refresh da bi se dobilo četiri karte. Unesite izraz koji upotrebljava četiri broja od 1 do 13, pri čemu svaki broj može da se koristi samo jedanput. Možete koristiti operacije sabiranja, oduzimanja, množenja i deljenja, kao i zagrade u izrazu. Izraz treba da da broj 24.

Zadatak 11 (15 minuta): Možete koristiti operacije sabiranja, oduzimanja, množenja i deljenja, kao i zagrade u izrazu. Izraz treba da da broj 24. Posle unosa izraza, kliknite dugme Verify da bi se izvršila provera da li brojevi u izrazu su trenutno izabrani i da li je rezultat izraza tačan. Verifikacije se prikazuje upotrebom klase JOptionPane. Pretpostavimo da su slike memorisane u datotekama 1.png, 2.png, ...52.png. po redu prema oznakama: pik, srce, karo i tref. Prema tome, prvih 13 slika su pikovi sa brojevima 1,2,3, i 13



Slika 10.2 Slika uz zadatak 4 [1]

▼ Zaključak

REZIME

Glavne pouke ove lekcije.

1. **Java Collections Framework** podržava skupove, liste, redove i mape. Oni su definisani u interfejsima: **Set**, **List**, **Queue** i **Map**.
2. Lista skladišti uređenu kolekciju elemenata.
3. Sve konkretne klase u **Java Collection Framework** implementiraju interfejse **Cloneable** i **Serializable**. Prema tome, njihovi primerci mogu se klonirati i serijalizovati.
4. Ako želite da imate duple elemente u kolekciji, morate da koristite listu. Lista ne samo da dozvoljava duple elemente, već takođe dozvoljava korisniku da specificira gde želi da ih smesti. Korisnik može da pristupi elementu preko indeksa.
5. Podržavaju se dva tipa liste: **ArrayList** i **LinkedList**. **ArrayList** je niz promenljive veličine koji implementira interfejs **List**. Svi metodi u **ArrayList** su definisani u **List** interfejsu. **LinkedList** je povezani lista koja implementira interfejs **List**. Pored primene interfejsa **List**, ova klasa obezbeđuje metode za nalaženje, ubacivanje i uklanjanje elemente sa oba kraja liste.
6. **Comparator** može se koristiti za poređenje objekata klase koja ne implementira **Comparable**.
7. **Vector** klasa nasleđuje **AbstractList** klasu. Od Java 2, klasa **Vector** je ista kao klasa **ArrayList**, sem što su metodi za pristup i modifikaciju elemenata vektora sinhronizovani. Klasa **Stack** nasleđuje klasu **Vector** i obezbeđuje nekoliko metoda za manipulaciju sa stekom.
8. Interfejs **Queue** predstavlja red. Klasa **PriorityQueue** implementira interfejs **Queue** za prioritetni red.

REFERENCE

Literatura korišćena u ovoj lekciji

1. Y. Daniel Liang, Introduction to Java Programming, Comprehensive Version, Chapter 20, 10th edition, Pierson – preporučeni udžbenik