DIZAJN I RAZVOJ SOFTVERA

Upravljanje izuzecima

ArrayList i LinkedList kolekcije

Upravljanje izuzecima

- **Izuzetak** je indikacija da se desio problem tokom izvršenja programa.
- **Upravljanje izuzecima** (eng. *exception handling*) predstavlja "hvatanje" i obradu izuzetaka. Neki izuzeci se mogu obraditi tako da program neometano nastavi sa radom, dok će drugi zahtijevati prekid rada programa, kad obrada izuzetka treba da omogući postupan prekid rada.
- Upravljanje izuzecima se koristi u slučaju **sinhronih grešaka**, koje se dešavaju prilikom izvršenja naredbi. Ove greške uključuju pristupanje nepostojećem elementu niza, cjelobrojno dijeljenje sa nulom, prosljeđivanje neregularnih argumenata metodi, prekid izvršavanja niti, neuspješnu alokaciju memorije.
- Obrada izuzetaka se ne primjenjuje kod problema vezanih za **asinhrone događaje**, koji se dešavaju paralelno sa izvršavanjem programa i nezavisni su od njega. Tu spadaju ulazno-izlazne operacije diska, prijem poruke, klik miša ili pritisak dugmeta tastature.

Primjer bez upravljanja izuzecima

```
import java.util.Scanner;

public class RazlomakBezObradeIzuzetaka {

  public static void main(String[] args) {
    Scanner unos = new Scanner(System.in);

    System.out.print("Unijeti imenilac i brojilac razlomka: ");
    int imenilac = unos.nextInt();
    int brojilac = unos.nextInt();
    int rezultat = kolicnik(imenilac, brojilac);

    System.out.printf("%d / %d = %d\n ", brojilac, imenilac, rezultat);
  }

  public static int kolicnik(int im, int br) {
    return br / im;
  }
}
```

Primjer bez upravljanja izuzecima

Prvo izvršavanje

```
Unijeti imenilac i brojilac razlomka: 5
8
8 / 5 = 1
```

Drugo izvršavanje

```
Unijeti imenilac i brojilac razlomka: 0
5
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at RazlomakBezObradeIzuzetaka.kolicnik(RazlomakBezObradeIzuzetaka.java:17)
        at RazlomakBezObradeIzuzetaka.main(RazlomakBezObradeIzuzetaka.java:11)
```

Treće izvršavanje

```
Unijeti imenilac i brojilac razlomka: 5
tetris
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at RazlomakBezObradeIzuzetaka.main(RazlomakBezObradeIzuzetaka.java:10)
```

Komentari primjera

- Kad unesemo imenilac = 0, dolazi do izuzetka tipa `ArithmeticException`. Java ne dozvoljava dijeljenje sa nulom u celobrojnoj aritmetici, pa dolazi do izuzetka u metodi `kolicnik`.
- Kaže se još da Java „**baca**“ izuzetak.
- Java dozvoljava dijeljenje sa nulom u aritmetici sa pomičnim zarezom, pri čemu se dobija rezultat `Infinity` ili `-Infinity`. Također, dozvoljeno je i dijeljenje tipa `0.0/0.0`, kada se dobija vrijednost `NaN` (Not-a-Number).
- Kada dođe do izuzetka, ispisuje se odgovarajuća poruka koja sadrži ime izuzetka koje ukazuje na tip problema koji se desio i stanje steka u trenutku pojave izuzetka. Stanje steka sadrži niz poziva metoda, pri čemu se u zadnjoj metodi u tom pozivu (onoj sa vrha steka) desio izuzetak.
- Vrh steka ukazuje na **tačku bacanja** (eng. *throw point*), koji je određen imenom fajla, metodom i linijom koda koja je uzrokovala izuzetak. Ovo može značajno pomoći u ispravljanju grešaka.
- U trećem izvršenju programa, umjesto `int` vrijednosti unosimo string, pa dolazi do greške tipa `InputMismatchException`.
- U slučaju ova dva tipa izuzetka, prekida se izvršavanje programa.

Primjer sa upravljanjem izuzecima

```
import java.util.InputMismatchException;
public class RazlomakSaObradomIzuzetaka {
   public static void main(String[] args) {
      Scanner unos = new Scanner(System.in);
      boolean losUnos = true;
      while(losUnos) {
         System.out.print("Unijeti imenilac i brojilac razlomka: ");
         try {
            int imenilac = unos.nextInt();  int brojilac = unos.nextInt();
            int rezultat = kolicnik(imenilac, brojilac);
            System.out.printf("%d / %d = %d", brojilac, imenilac, rezultat);
            losUnos = false;
         }
         catch(ArithmeticException e) {
            System.out.printf("\nIzuzetak: %s\nImenilac je 0.\n\n", e);
         }
         catch(InputMismatchException e) {
            System.out.printf("\nIzuzetak: %s\n Morate unijeti cijele brojeve\n\n", e);
            unos.nextLine();
         }
      }
      System.out.print("\nIzlazak iz programa");
   }
   public static int kolicnik (int im, int br) throws ArithmeticException {
      return br / im;
   }
}
```

Deklaracija izuzetaka koji se obrađuju.

Klasu ArthmeticException ne moramo uvoditi jer se nalazi u paketu java.lang.

Primjer sa upravljanjem izuzecima

Ispis

```
Unijeti imenilac i brojilac razlomka: 0 5

Izuzetak: java.lang.ArithmeticException: / by zero
Imenilac je 0.

Unijeti imenilac i brojilac razlomka: 3 smokva

Izuzetak: java.util.InputMismatchException
Morate unijeti cijele brojeve

Unijeti imenilac i brojilac razlomka: 5 6
6 / 5 = 1
Izlazak iz programa
```

try i catch blokovi

- U `try` blok se stavljaju naredbe koje mogu baciti izuzetak, kao i one koje se neće izvršiti ako dođe do izuzetka.
- Nakon `try` bloka, dolazi bar jedan `catch` blok ili `finally` blok.
- Svaki `catch` blok ima tačno jedan parametar izuzetka, koji se navodi u zagradi odmah nakon ključne riječi `catch`, i koji određuje tip izuzetka koji `catch` blok može da obradi.
- Kada se desi izuzetak u okviru `try` bloka izvršava se prvi `catch` blok čiji tip parametra odgovara tipu bačenog izuzetka. Tip parametra odgovara tipu izuzetka ako pripada klasi ili superklasi izuzetka.
- Unutar `catch` bloka koristimo ime parametra izuzetka da interagujemo sa objektom izuzetka.
- Između `try` bloka i njegovih `catch` blokova se ne smije naći nijedna izvršna naredba. Ako se nađe, to je sintaksna greška.

Izuzetak prekida nit

- Ukoliko se desi `InputMismatchException` izuzetak, naredba `nextInt` neće uspješno učitati podatak. Ukoliko bi pokušali da opet učitamo podatak (što radi `while` petlja u ovom primjeru), desio bi se isti izuzetak.
- Pogrešan unos se na neki način mora „protjerati“, što u našem slučaju vrši metoda `nextLine`.
- Metoda `nextLine` vraća liniju teksta, počev od tekuće pozicije čitanja.
- Neuhvaćeni izuzetak je onaj koji nema odgovarajućih `catch` blokova.
- U prvom primjeru smo imali dva neobrađena izuzetka, kada se prekinulo izvršenje programa. Ovo nije uvek slučaj.
- Java koristi višenitni model izvršenja programa – svaka nit se izvršava paralelno. Jedan program može imati više niti.
- Ako ima samo jednu nit, neuhvaćeni izuzetak prekida izvršenje programa.
- Ako program ima više niti, prekida se izvršenje samo one niti u kojoj se desio izuzetak. Ipak, može se desiti da su niti međusobno povezane, pa prekid izvršenja jedne niti može nepovoljno uticati na ostale niti.

Terminacioni model upravljanja izuzecima

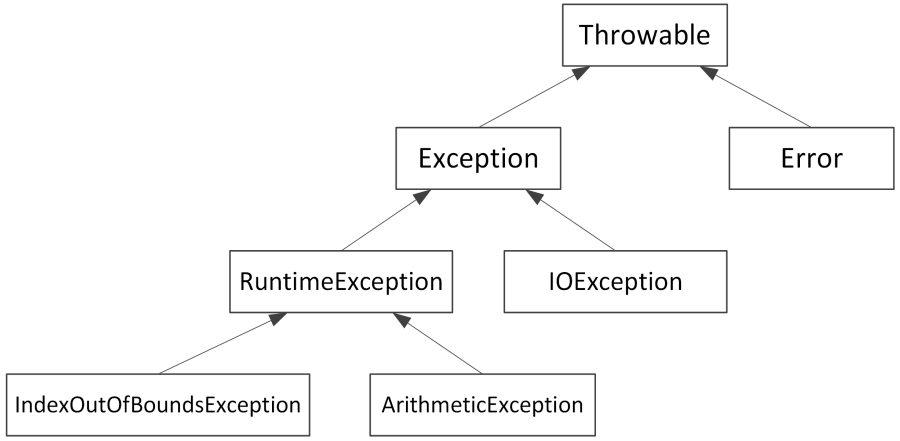
- Kada se desi izuzetak u `try` bloku, prekida se izvršenje `try` bloka (preostale naredbe tog bloka se neće izvršiti) i prelazi se na izvršenje prvog odgovarajućeg `catch` bloka.
- Nakon obrade izuzetka u `catch` bloku, kontrola toka se ne vraća na naredbu `try` bloka koja je bacila izuzetak, jer su promjenljive `try` bloka dealocirane, već se nastavlja nakon poslednjeg `catch` bloka.
- Ovo je poznato pod imenom **terminacioni model upravljanja izuzecima** (eng. *termination model of exception handling*).
- Nasuprot ovom modelu, neki jezici (Common Lisp i Dylan) koriste **nastavljački model upravljanja izuzecima** (eng. *resumption model of exception handling*) u kome se, nakon obrade izuzetka, kontrola vraća u `try` blok nakon tačke bacanja. Ipak, mnogo je češći terminacioni model.
- Ako se u okviru `try` bloka ne desi izuzetak, `catch` blokovi se preskaču i nastavlja se sa prvom naredbom nakon poslednjeg `catch` bloka. Ukoliko postoji `finally` blok, o čemu će biti riječi kasnije, on će se izvršiti.
- `try` blok i odgovarajući `catch` i/ili `finally` blok čine **`try` naredbu**.

Klauzula throws

- Metoda `kolicnik` može da baci izuzetak tipa `ArithmeticException`.
- Pomoću `throws` klauzule, koja se navodi nakon zagrada sa listom parametara metode, a prije tijela metode, specificira se tip izuzetka koji metoda može baciti i o tome obaveštavaju klijentske metode.
- Metoda može baciti više tipova izuzetaka, kad se tipovi odvajaju zarezima u `throws` klauzuli.
- Izuzetak mogu baciti naredbe date metode ili druge metode pozvane iz date metode. Metoda može baciti izuzetak klasnih tipova navedenih u `throws` klauzuli ili njihovih potklasa.
- Kada se desi izuzetak u okviru metode `kolicnik`, prekida se izvršavanje metode i ona ne vraća rezultat. Lokalne promjenljive metode se dealociraju. Ako u metodi postoje reference na objekte, koji nemaju drugih referenci, ti objekti postaju smeće.

Klasna hijerarhija izuzetaka

- Sve Javine klase izuzetaka direktno ili indirektno nasljeđuju klasu **`Exception`**.
- Klasa **`Throwable`** je superklasa klase `Exception` i samo se `Throwable` objekti mogu koristiti u mehanizmu obrade izuzetaka (može ih baciti JVM ili mogu biti bačeni pomoću `throw` naredbe).
- Klasa `Throwable` ima dvije potklase: **`Exception`** i **`Error`**.
- Klasa `Exception` i njene potklase, na primjer **`RuntimeException`** (paket `java.lang`) i **`IOException`** (paket `java.io`) predstavljaju izuzetne situacije koje se mogu desiti u Java programu i koje može obraditi aplikacija.
- Klasa `Error` i njene potklase predstavljaju abnormalne situacije koje se dešavaju u JVM. Ove greške su retke i obično uzrokuju „pucanje“ programa.



Dio klasne hijerarhije koji počinje od klase `Throwable`

Provjereni i neprovjereni izuzeci

- Izuzeci se dijele na **provjerene** i **neprovjerene**, i to određuje tip izuzetka.
- Sve direktne ili indirektne potklase klase `RuntimeException` pripadaju neprovjerenim izuzecima.
- Neprovjereni izuzeci su obično uzrokovani greškama u programskom kodu. Primjeri neprovjerenih izuzetaka su `ArithmeticException` i `ArrayIndexOutOfBoundsException`. Klase koje nasljeđuju klasu `Error` pripadaju neprovjerenom tipu izuzetaka.
- Provjereni izuzeci su obično uzrokovani situacijama koje se ne tiču programskog koda, na primjer pokušaj otvaranja fajla koji ne postoji.
- Sve klase koje nasljeđuju klasu `Exception`, isključujući klasu `RuntimeException` pripadaju tipu provjerenih izuzetaka.
- Razlika između provjerenih i neprovjerenih izuzetaka je važna, jer kod provjerenih izuzetaka Java kompajler forsira **uhvati-ili-deklariši zahtev** (eng. *catch-or-declare requirement*).
- Kompajler provjerava deklaracije svih metoda pozvanih unutar tekuće metode da utvrdi da li koja od njih baca provjereni izuzetak. Ako je to slučaj, kompajler verifikuje da li je provjereni izuzetak uhvaćen ili deklarisan u `throws` klauzuli.

Uhvati ili deklariši

- Da bi zadovoljio uhvati dio uhvati-ili-deklariši zahtjeva, kôd koji generiše izuzetak se mora naći u okviru `try` bloka i mora se obezbijediti `catch` blok za taj tip izuzetka (ili tip njegove superklase).
- Da bi zadovoljio deklariši dio uhvati-ili-deklariši zahtjeva, metoda koja generiše izuzetak mora navesti pomoću `throws` klauzule tip provjerenog izuzetka koji baca.
- Ukoliko uhvati-ili-deklariši zahtev nije zadovoljen, doći će do greške. Doći će i do greške kompajliranja ukoliko metod pokuša da baci provjereni izuzetak, ili poziva drugu metodu koja baca provjereni izuzetak, a taj izuzetak nije naveden u `throws` klauzuli.
- Ako u potklasi redefinišemo metodu superklase, greška je navesti više tipova izuzetaka u `throws` klauzuli nego što ih ima metoda superklase. Sa druge strane, `throws` klauzula metode potklase može sadržati podskup tipova izuzetaka iz `throws` klauzule metode superklase.
- Javin kompajler ne provjerava da li je neprovjeren izuzetak uhvaćen ili deklarisan, i ovi izuzeci se ne navode u `throws` klauzuli. Ovi izuzeci se mogu preduprijediti ispravnim programiranjem. Na primjer, prije računanja količnika, možemo provjeriti da li je imenilac nula, pa preduzeti korektivne radnje.

Superklase i potklase u hvatanju izuzetaka

- `catch` blok koji hvata objekte izuzetaka superklasnog tipa, može hvatati i sve objekte potklasa date superklase. Na ovaj način se omogućuje polimorfno obrađivanje srodnih izuzetaka.
- Ukoliko postoji više `catch` blokova čiji tip parametra odgovara tipu bačenog izuzetka, izvršava se samo prvi takav `catch` blok.
- Greška je navesti dva `catch` bloka sa istim tipom parametra.
- Sa druge strane, zbog nasljeđivanja i veze tipa jeste, može postojati više `catch` blokova čiji tip odgovara tipu izuzetka. Na primjer, možemo imati `catch` blokove koji hvataju izuzetke tipa `ArithmeticException`, `RuntimeException` i `Exception`, jedan za drugim. Klasa `ArithmeticException` je izvedena iz `RuntimeException`, a ova iz klase `Exception`.
- Ovdje se mora voditi računa da se tip više klase u hijerarhiji nasljeđivanja mora naći nakon nižih klasa. Na primjer, `catch` blok koji hvata tip `Exception` se mora naći nakon `catch` blok koji hvata tip `RuntimeException`, a ovaj nakon catch blok koji hvata tip `ArithmeticException`. U suprotnom, dolazi do greške.

finally blok

- **`finally`** blok je opcioni dio `try` naredbe, i ukoliko postoji, nalazi se nakon posljednjeg `catch` bloka. U slučaju da nema `catch` blokova, `finally` blok se mora naći odmah nakon `try` bloka.
- `finally` blok se izvršava bez obzira da li je došlo do izuzetka ili ne.
- `finally` blok će se izvršiti ako se iz `try` bloka izašlo pomoću naredbi `return`, `break` ili `continue`.
- `finally` blok se neće izvršiti ukoliko se prekine izvršavanje aplikacije, što se može uraditi pomoću metode `System.exit`.
- `finally` blok obično sadrži naredbe koje oslobađaju resurse sistema. Bez obzira da se li desio izuzetak ili ne, resurs koji je alociran u `try` bloku se može osloboditi u `finally` bloku.

finally blok

- `try` naredbe mogu biti ugnježdene, tj. jedna `try` naredba se može naći unutar `try` bloka druge try naredbe.
- Ukoliko nijedan `catch` blok ne hvata izuzetak, kontrola toka prelazi na odgovarajući `finally` blok, a nakon toga se izuzetak prosljeđuje sljedećem spoljašnjem `try` bloku, koji se obično nalazi u pozivajućoj metodi. `catch` blok spoljašnje `try` naredbe može da uhvati i obradi izuzetak, ili se izuzetak može proslediti sledećem spoljašnjem `try` bloku.
- `finally` blok će se izvršiti i ako `catch` blok baci izuzetak, nakon čega se izuzetak prosleđuje sledećem spoljašnjem `try` bloku.
- Izvršavanje `finally` blokova u raznim situacijama je ilustrovano sledećim programom.

Primjer sa izuzecima

```
public class IzuzeciTest {
  public static void main(String[] args) {

    try{ bacaIzuzetak(); }
    catch (Exception e){
      System.out.println("Izuzetak obrađen u metodi main");
    }
    neBacaIzuzetak();
  }

  public static void bacaIzuzetak() throws Exception{
    try{
      System.out.println("try blok metode bacaIzuzetak");
      throw new Exception("### Naš izuzetak ###");
    }
    catch (Exception e){
      System.out.printf("Izuzetak \"%s\" uhvaćen u metodi
bacaIzuzetak\n",
          e.getMessage());
      throw e;
    }
    finally{
      System.out.println("finally blok metode bacaIzuzetak");
    }
  }
```

Ponovno bacanje izuzetka

Primjer sa izuzecima

```
  public static void neBacaIzuzetak(){
    try{
      System.out.println("Metoda neBacaIzuzetak");
    }
    catch (Exception e){
      System.out.println(e);
    }
    finally{
      System.out.println("finally blok metode neBacaIzuzetak");
    }

    System.out.println("Kraj metode neBacaIzuzetak");
  }
}
```

Ispis

```
try blok metode bacaIzuzetak
Izuzetak "### Naš izuzetak ###" uhvaćen u metodi bacaIzuzetak
finally blok metode bacaIzuzetak
Izuzetak obrađen u metodi main
Metoda neBacaIzuzetak
finally blok metode neBacaIzuzetak
Kraj metode neBacaIzuzetak
```

Bacanje i ponovno bacanje izuzetaka

- U `try` bloku metode `bacaIzuzetak` se izuzetak baca pomoću **`throw`** naredbe.
- Argument naredbe `throw` je objekat bilo koje klase izvedene iz `Throwable`.
- Kad se utvrdi da se dati izuzetak ne može u potpunosti obraditi u odgovarajućem `catch` bloku, taj se izuzetak može **ponovo baciti**, što je slučaj u `catch` bloku metode `bacaIzuzetak`.
- Na ovaj način se obrada izuzetka prepušta prvoj spoljašnjoj `try` naredbi, a to je u našem slučaju `try` naredba iz metode `main`. Za ponovno bacanje izuzetaka se također koristi ključna riječ `throw`, pri čemu je njen argument neobrađeni objekat izuzetka.
- Izuzetak bačen u okviru `catch` bloka se ne može obraditi u okviru date `try` naredbe, već se mora proslediti spoljašnjoj `try` naredbi.
- Izuzeci se ne mogu ponovo baciti iz `finally` bloka, jer parametar izuzetka iz `catch` bloka više ne postoji. Ipak, `finally` blok može baciti izuzetak. U tom slučaju bi se spoljašnjoj `try` naredbi proslijedio izuzetak bačen u `finally` bloku, a ne eventualno neobrađeni izuzetak iz odgovarajuće `try` naredbe.
- Metoda `getMessage()` vraća poruku vezanu za objekat izuzetka.

Osnovno o kolekcijama

- U Javi je dužina niza fiksna. Jednom kreiran niz na ovaj način ne može promijeniti dužinu, jer mu je `length` podatak deklarisan ključnom riječju `final`. Promjena dužine niza podrazumijeva kreiranje novog niza.
- Ovaj se nedostatak može eliminisati korišćenjem kolekcija, kakva je recimo `ArrayList`, koja dozvoljava dinamičku promjenu dužine niza.
- Kolekcije su strukture podataka u kojima se čuvaju reference na druge objekte. Najčešće su to reference na objekte istog tipa.
- Tip podatka čije se reference čuvaju u kolekciji se specificiraju tokom kompajliranja, pa nije moguće dodijeliti nekompatibilan tip elementu kolekcije. Na primjer, ako imamo kolekciju tipa `String`, u nju nije moguće upisati referencu na objekat klase `Knjiga`.
- Kolekcije mogu čuvati samo reference, a ne i vrijednosti primitivnih tipova (`int`, `float`, `double`, `char` itd.).
- Prosti tipovi se mogu čuvati u kolekciji indirektno, koristeći omotačke klase.

Pakovanje primitivnih tipova

- Svaki primitivni tip u Javi ima odgovarajuću **omotačku klasu** (eng. *type-wrapper class*) u paketu `java.lang`.
- Omotačke klase su: **`Boolean`**, **`Byte`**, **`Character`**, **`Double`**, **`Float`**, **`Integer`**, **`Long`** i **`Short`**.
- Omotačke klase omogućavaju da se sa primitivnim tipovima radi kao sa objektima.
- Sve omotačke klase nasljeđuju klasu **`Number`**.
- Omotačke klase su finalne klase, pa se ne mogu naslijediti.
- **Pakovanje** i **raspakivanje** (eng. *boxing* i *unboxing*), kao operacije konvertovanja primitivnog tipa u odgovarajući objekat i obrnuto, se u Javi vrše automatski.
- Na ovaj način se primitivne vrijednosti mogu koristiti tamo gdje se očekuju omotačke klase i obrnuto.

Interfejs Collection

- U hijerarhiji kolekcija, interfejs `Collection` je korijeni interfejs iz koga su izvedeni interfejsi `List`, `Set` i `Queue`, kao i mnogi drugi.
- Interfejs `List` definiše kolekciju u kojoj se elementi mogu ponavljati.
- Interfejs `Set` definiše kolekciju u kojoj se elementi ne mogu ponavljati.
- Interfejs `Queue` definiše kolekciju u kojoj se elementi ubacuju na kraj reda, a brišu sa početka. Drugim riječima, `Queue` radi po principu **First-In-First-Out** (FIFO), tj. prvi koji ulaze u kolekciju se prvi brišu iz kolekcije i obrnuto.
- Interfejs `Collection` sadrži metode koje se izvode nad čitavom kolekcijom, kao što su dodavanje i brisanje elemenata kolekcije, poređenje elemenata kolekcije. Ne mogu se vršiti operacije na nivou jednog elementa kolekcije.
- Kolekcija se može konvertovati u niz, što se obično radi da bi se određene operacije sa nizom izvele efikasnije, kao i da bi se konvertovani niz mogao proslijediti metodama koje ne rade sa kolekcijama ili obraditi naslijeđenim starim kodom koji ne poznaje kolekcije.

Interfejs Collection

- Interfejs `Collection` ima metodu **`iterator()`** koja vraća **`Iterator`** objekat, koji omogućava prolazak kroz kolekciju i brisanje elemenata kolekcije.
- Veliki broj kolekcija je implementiran tako da im konstruktor za argument ima objekat tipa `Collection`, čime se omogućava da nova kolekcija sadrži elemente postojeće kolekcije.
- Od interfejsa kolekcije `Collection`, radićemo sa interfejsima `List`, `Queue` i `Set`.
- Obradićemo i interfejs `Map`, koji ne pripada ovoj hijerarhiji.

List kolekcija

- `List` je uređena kolekcija koja može sadržati duplikate elemenata.
- Kao kod nizova, indeks prvog elementa liste je 0. Za razliku od nizova, elementima se ne može pristupiti koristeći zagrade `[]`, već preko odgovarajućih metoda (`set` i `get`).
- Pored metoda naslijeđenih iz interfejsa `Collection`, interfejs `List` obezbjeđuje niz metoda za rad sa elementima liste (preko indeksa), rad sa opsegom elemenata liste, pretragu elemenata itd.
- Nekoliko klasa implementira interfejs `List`, od kojih su najčešće korišćene **`ArrayList`** i **`LinkedList`**.
- `ArrayList` se implementira kao niz promjenljive dužine. Stoga su operacije gdje se često mijenja broj elemenata `ArrayList` objekta neefikasne.
- `LinkedList` sa implementira kao povezana lista. Umetanje novih elemenata u sredinu liste, brisanje elemenata liste i slično su značajno efikasniji u odnosu na `ArrayList` objekte.

Korišćenje iteratora

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

public class RadSaIteratorima {
   public static void main(String[] args) {
      String imena[] = {"Ana", "Rade", "Ines", "Miloš", "Azra", "Ilija"};
      List< String > listaImena = new ArrayList< String >();
      for(String ime: imena)
         listaImena.add(ime);

      String imenaZaBrisanje[] = {"Rade", "Ilija", "Azra"};
      List< String > listaImenaZaBrisanje = new ArrayList< String >();
      for(String ime: imenaZaBrisanje)
         listaImenaZaBrisanje.add(ime);

      Iterator< String > iter = listaImena.iterator();
      System.out.println("Štampanje liste pomoću iteratora:");
      while(iter.hasNext())
         System.out.print(iter.next() + " ");

      iter = listaImena.iterator();
      while(iter.hasNext())
         if(listaImenaZaBrisanje.contains(iter.next()))
            iter.remove();

      System.out.println("\nLista nakon brisanja imena:");
      for(int i = 0; i < listaImena.size(); i++)
         System.out.print(listaImena.get(i) + " ");
   }
}
```

Kreiranje liste imena

Kreiranje liste imena za brisanje

Kreiranje iteratora kolekcije i štampanje liste pomoću iteratora

Brisanje elemenata liste imena koji se nalaze u listi

Ispis

```
Štampanje liste pomoću iteratora:
Ana Rade Ines Miloš Azra Ilija
Lista nakon brisanja imena:
Ana Ines Miloš
```

Komentari primjera

- U prethodnom primjeru, kreirali smo dva niza stringova `imena` i `imenaZaBrisanje`, i na osnovu njih dvije liste `listaImena` i `listaImenaZaBrisanje`. Cilj je iz liste `listaImena` izbrisati sve elemente koji se pojavljuju u `listaImenaZaBrisanje`.
- Liste su generički tipovi, pri čemu se tip elemenata navodi unutar < >:

```
List< String > listaImena
```

- Slično, klasa `ArrayList` je generička klasa.
- Brisanje elemenata liste ćemo vršiti pomoću iteratora `iter`, koji se dobija pomoću metode **`iterator()`** liste (i generalno kolekcije).
- Metoda **`hasNext()`** vraća `true` ako iterator nije stigao do kraja kolekcije i `false` u suprotnom.
- Ukoliko iterator nije stigao do kraja kolekcije, metoda **`next()`** iteratora vraća referencu na naredni element kolekcije.

Komentari primjera

- Kad dođemo do kraja kolekcije, moramo nanovo inicijalizovati iterator pozivom metode **`iterator`** predmetne kolekcije.
- Metoda **`contains`** liste vraća `true` ako lista sadrži element koji se prosleđuje kao argument metode.
- Metoda iteratora **`remove()`** briše iz liste tekući element koji vraća iterator.
- Ako se kolekcija promijeni nakon kreiranja iteratora za tu kolekciju, iterator postaje nevažeći. Pokušaj da se izvrši operacija sa iteratorom će u tom slučaju baciti izuzetak tipa `ConcurrentModificationException`.
- U narednom primjeru, kreiraćemo **`LinkedList`** kolekciju (paket `java.util.LinkedList`) i **`ListIterator`** objekat (kolekcija `java.util.ListIterator`) za prolazak kroz nju.

Primjer sa LinkedList

```
import java.util.*;

public class RadSaListIteratorom {

   public static void main(String[] args) {
      String imena[] = {"Azra", "Rade", "Ines"};
      List< String > listaImena = new LinkedList< String >();
      for(String ime: imena)
         listaImena.add(ime);

      String imena2[] = {"Dragan", "Ana", "Vesna", "Marko"};
      List< String > listaImena2 = new LinkedList< String >();
      for(String ime: imena2)
         listaImena2.add(ime);

      System.out.println("Lista prije dodavanja imena:");
      System.out.println(listaImena);

      listaImena.addAll(listaImena2);
      System.out.println("Lista nakon dodavanja imena:");
      System.out.println(listaImena);

      listaImena.subList(1, 4).clear();
      System.out.println("Lista nakon uklanjanja imena na poziciji 1-3:");
      System.out.println(listaImena);

      obrniListu(listaImena, listaImena2);
      System.out.println("Lista sa obrnutim redosljedom imena:");
      System.out.println(listaImena2);

      prebaciUVelika(listaImena);
      System.out.println("Lista nakon prebacivanja u velika slova:");
      System.out.println(listaImena);
   }
```

Štampanje liste u jednoj liniji [prvi, drugi, …]

Dodavanje svih elemenata jedne liste drugoj

Brisanje elemenata liste sa indeksima 1, 2 i 3

Ispis

```
Lista prije dodavanja imena:
[Azra, Rade, Ines]
Lista nakon dodavanja imena:
[Azra, Rade, Ines, Dragan, Ana, Vesna, Marko]
Lista nakon uklanjanja imena na poziciji 1-3:
[Azra, Ana, Vesna, Marko]
Lista sa obrnutim redosljedom imena:
[Marko, Vesna, Ana, Azra]
Lista nakon prebacivanja u velika slova:
[AZRA, ANA, VESNA, MARKO]
```

Primjer sa LinkedList

```
   public static void obrniListu(List< String > a, List< String > b) {
      b.clear();
      ListIterator< String > iter = a.listIterator(a.size());
      while(iter.hasPrevious())
         b.add(iter.previous());
   }

   public static void prebaciUVelika(List< String > a) {
      ListIterator< String > iter = a.listIterator();
      while(iter.hasNext()) {
         String ime = iter.next();
         iter.set(ime.toUpperCase());
      }
   }
}
```

Brisanje svih elemenata liste

Broj koji se proslijedi konstruktoru `ListIterator`-a određuje koji će element biti vraćen prvim pozivom metode `next`. Metoda `previous` vraća element čiji je indeks za 1 manji od proslijeđenog broja.

Da smo u slučaju iteratora `listIterator(a.size())` pozvali metodu `next`, došlo bi do izuzetka `NoSuchElementException`.

Komentari primjera

- Cilj prethodnog primjera je ilustracija dodatnih operacija sa listama.
- Metoda **`addAll()`** na listu koja poziva metodu nadovezuje elemente liste (kolekcije) koja se prosljeđuje kao argument.
- Metoda **`subList`** vraća dio liste ograničen indeksima elemenata koji se prosljeđuju kao argumenti. Početni indeks je uključen u vraćeni dio liste, a krajnji nije.
- Metoda **`clear()`** liste briše sve elemente liste. U našem slučaju, naredba

  ```
  listaImena.subList(1, 4).clear();
  ```

  će obrisati drugi, treći i četvrti element liste (indeksiranje počinje od 0). Početni indeks je uključen u vraćeni dio liste, krajnji nije.
- Metoda `obrniListu` obrće redosled elemenata liste. Za parametre ima dvije reference na liste, pri čemu u drugu upisujemo elemente prve u obrnutom redosljedu.

ListIterator

- Za razliku od prethodnog primjera, gdje smo koristili iterator tipa `Iterator`, koji predstavlja iterator kroz kolekciju, u ovom koristimo interfejs **`ListIterator`** koji predstavlja iterator kroz listu.
- Za razliku od `Iterator`, pomoću `ListIterator` iteratora možemo:
  - da obilazimo listu dvosmjerno (od prvog elementa ka zadnjem i obrnuto),
  - da dobijemo indeks tekućeg elementa kolekcije, tj. elementa koji će biti vraćen narednim pozivom metode `next()`,
  - da promijenimo vrijednost elementa liste pomoću metode **`set`**,
  - da dodamo novi element liste pomoću metode **`add`**.
- `ListIterator`-e mogu da koriste sve kolekcije koje implementiraju `List` interfejs, dakle `ArrayList`, `LinkedList` itd.
- Ako želimo da se krećemo unazad, konstruktoru `ListIterator` objekta prosljeđujemo kao argument broj elemenata liste, kao što je urađeno u:

```
ListIterator< String > iter = a.listIterator(a.size());
```

ListIterator

- Kad se konstruktoru `ListIterator`-a proslijedi cio broj (indeks elementa), taj broj određuje koji će element biti vraćen sledećim pozivom metode `next`. Poziv metode `previous` vraća element čiji je indeks za 1 manji od proslijeđenog broja.
- Da smo u slučaju iteratora `listIterator(a.size())` pozvali metodu `next`, došlo bi do izuzetka tipa `NoSuchElementException`.
- Pri kretanju unazad, metoda **`hasPrevious()`** vraća `true` ako iterator nije stigao do kraja kolekcije i `false` u suprotnom.
- Ukoliko iterator nije stigao do kraja kolekcije, metoda **`previous()`** iteratora vraća referencu na naredni element kolekcije (krećući se unazad).
- Metoda `obrniListu` se mogla realizovati i na sljedeći način:

```
public static void obrniListu(List< String > a, List< String > b){
   b.clear();
   for(int i = a.size()-1; i >= 0; i--)
      b.add(a.get(i));
}
```

Korisne metode LinkedList kolekcije

Metoda	Opis
`add`	Dodaje element na kraj LinkedList kolekcije (verzija metode sa jednim parametrom) ili na određenu poziciju (pozicija prvi parametar, element drugi parametar).
`addFirst` `addLast`	Dodaju element na prvu, odnosno zadnju poziciju u listi.
`addAll`	Nadovezuje kolekciju (parametar metode) na predmetnu listu.
`remove`	Uklanja prvu pojavu specificirane vrijednosti ili elementa sa specificiranim indeksom. Specificirana vrijednost ili indeks su parametri metode.
`removeFirst` `removeLast`	Uklanjaju prvi, odnosno zadnji element liste.
`clear`	Briše kolekciju (uklanja sve elemente).
`contains`	Vraća true ako kolekcija sadrži traženi element i false u suprotnom.
`get`	Vraća element sa specificiranim indeksom.
`indexOf`	Vraća indeks prve pojave specificiranog elementa kolekcije, odnosno -1 ako element ne postoji
`size`	Vraća broj elemenata smeštenih u kolekciji.
`toArray`	Konvertovanje kolekcije u niz.

asList metoda klase Arrays

- **`asList`** metoda klase `Arrays` omogućava da se predmetni niz posmatra kao `List` kolekcija. Na ovaj način, nizom možemo manipulisati kao da je u pitanju lista.
- `asList` metoda vraća `List` izgled (eng. *`List` view*) niza, koji se u tom slučaju naziva podržavajući niz (eng. *backing array*).
- Bilo kakva promjena `List` izgleda mijenja podržavajući niz i obrnuto.
- Jedina dozvoljena operacija na `List` izgledu koga vraća metoda `asList` je **`set`**, koja mijenja vrijednost elementa izgleda i odgovarajućeg elementa niza. Pokušaj bilo kakve druge promjene izgleda, npr. dodavanje i brisanje elementa izgleda, baca `UnsupportedOperationException` izuzetak.
- Metoda `subList` iz prethodnog primjera vraća izgled dijela liste, određenog početnim i krajnjim indeksom (parametri metode).
- U nastavku dajemo dva primjera korišćenja metode `asList`.

Primjer za asList

```
import java.util.List;
import java.util.Arrays;

public class asListPodrzavajuciNiz {

   public static void main(String[] args) {
      String imena[] = {"Teodora", "Enis", "Robert", "Marija", "Ana"};

      List< String > lista = Arrays.asList(imena);
      System.out.printf("Početna lista:\n%s\n", lista);

      lista.set(0, "Milutin");
      imena[1] = "Eva";

      System.out.println("Krajnji niz:");
      for(String ime: imena)
         System.out.print(ime + " ");

      System.out.printf("\nKrajnja lista:\n%s", lista);
   }
}
```

Promjena elementa izgleda i niza

Ispis

```
Početna lista:
[Teodora, Enis, Robert, Marija, Ana]
Krajnji niz:
Milutin Eva Robert Marija Ana
Krajnja lista:
[Milutin, Eva, Robert, Marija, Ana]
```

Promjene na izgledu se odražavaju na niz i obrnuto!

Primjer za asList i toArray

```
import java.util.LinkedList;
import java.util.Arrays;
public class AsListToArray {
   public static void main(String[] args) {
      String imena[] = {"Petar", "Milan", "Teodora"};
      LinkedList< String > lista = new LinkedList< String >(Arrays.asList(imena));
      System.out.println("Početna lista:\n" + lista);
      lista.add("Rade");
      lista.add(2, "Vesna");
      lista.addFirst("Marija");
      System.out.printf("Lista nakon dodavanja:\n%s\n", lista);
      imena = lista.toArray(new String[lista.size()]); <-- Kreiranje niza imena na osnovu liste imena
      System.out.println("Niz nakon kopiranja liste:");
      for(String ime: imena)
         System.out.print(ime + " ");
   }
}
```

Ispis

```
Početna lista:
[Petar, Milan, Teodora]
Lista nakon dodavanja:
[Marija, Petar, Milan, Vesna, Teodora, Rade]
Niz nakon kopiranja liste:
Marija Petar Milan Vesna Teodora Rade
```

Komentari primjera – Niz u listu

- Naredba

  ```
  new LinkedList< String >(Arrays.asList(imena));
  ```

  kreira novu povezanu listu stringova, koja se inicijalizuje stringovima iz niza `imena`.
- Ovako kreirana lista ne predstavlja izgled, kao što je to slučaj u prethodnom primjeru. Izgled je iskorišćen da se inicijalizuje lista, pa promjene na listi ne utiču na niz i obrnuto.
- Naredbe

  ```
  lista.add("Rade");
  lista.add(2, "Vesna");
  lista.addFirst("Marija");
  ```

  dodaju novi element liste na kraj, na poziciju sa indeksom 2 i na početak liste, respektivno. Postoji i metoda **`addLast`** koja dodaje element na kraj liste, isto kao metoda `add`.

Komentari primjera – Lista u niz

- Naredba

  ```
  lista.toArray(new String[lista.size()]);
  ```

  kreira novi niz stringova čiji su elementi jednaki elementima liste i vraća referencu na taj niz.
- Niz koji se prosleđuje kao argument metode `toArray` određuje tip vraćenog niza.
- Ovako kreiran niz nije povezan sa listom, tj. promjena elementa liste ne utiče na niz i obrnuto.
- Ako je dužina niza argumenta manja od dužine liste, kreira se novi niz čiji tip odgovara nizu argumentu i dužine jednake dužini liste.
- Ako je niz argument duži od liste, u sve elemente niza nakon poslednjeg elementa liste se upisuje `null`.