Java

Uvod

Hello world primer

Java program predstavlja skup objekata koji prozivaju jedni drugima metode i tako komuniciraju. Izvorni kod se uvek čuva u datotekama sa ekstenzijom .java. Na slici ispod je predstavljen „Hello World“ primer napisan u Javi. Za ispis na standardni izlaz koristi se funkcija *System.out.println()*. Ovaj programski kod potrebno je napisati u tekstualnom editoru i kompajlirati ga korišćenjem ***javac*** kompajlera, a nakon toga pokrenuti izvršavanje bajtkod programa u vidu datoteke .class koristeći Javu. Kako bi se kod mogao kompajlirati .java datoteku je potrebno nazvati isto kao klasu, u ovom slučaju MyFirstJavaProgram. Nakon izvršavanja ovog programa na ekranu će biti ispisano „Hello World”.

```
public class MyFirstJavaProgram {
      /* This is my first java program.
       * This will print 'Hello World' as the output
       */

      public static void main(String []args) {
            System.out.println("Hello World"); /* prints Hello World */
      }
}
```

```
>> javac MyFirstJavaProgram.java        - prevođenje programa
>> java MyFirstJavaProgram              - pokretanje programa

Hello World
```

Metoda *main* datog programa je statička, što znači da nije potrebno instancirati objekat klase *MyFirstJavaProgram*, već ju je moguće prozvati metodu bez instanciranja objekta. Statička metoda *main* ima parametre koji mogu da se proslede funkciji prilikom pokretanja samog programa. Dodavanjem ispisa argumenata na ekran to se može potvrditi. U ovom primeru korišten je *foreach* iskaz koji omogućava iteraciju kroz liste, nizove i kolekcije. Ključna reč *public* ispred naziva klase i main-a predstavlja modifikator pristupa o kome će kasnije biti reč.

```
public class MySecondJavaProgram {
      public static void main(Strings []args) {
            for(String arg: args) {
                  System.out.println(arg);
            }
```

```
        }
}
```

```
>> javac MySecondJavaProgram.java
>> java MySecondJavaProgram Pera Peric
```

```
Pera
Peric
```

Klase i objekti

Java je objektno orijentisan programski jezik. To znači da, umesto *osnovnih tipova podataka* (boolean, char, int, double, float...), mogu da se koriste novi entiteti koje korisnik može sam da definiše. Ti entiteti nazivaju se **objekti** i oni predstavljaju osnovni pojam objektno orijentisanog programiranja. Više sličnih objekata sa njihovim atributima i metodama grupišu se u **klasu**. Sa programske tačke gledišta **atributi** predstavljaju *promenljive* koje bliže opisuju atribute. Svaki objekat imaće atribute koji su definisani u klasi, međutim vrednosti tih atributa ne moraju biti iste za svaki objekat. Sa druge strane, **metode** predstavljaju operacije koje mogu da se izvode nad objektima.

Radi lakšeg razumevanja ovih pojmova u nastavku sledi primer. Neka *Student* predstavlja **klasu**, tada svaki određeni student (Pera, Mika, Žika) predstavlja **objekat** klase Student. Neka je *broj indeksa* definisani atribut, tada će student Pera imati broj indeksa 1, Mika 2, a Žika će imati indeks sa brojem 3. Potpuno je validno i da dva ili više studenta imaju atribut iste vrednosti. Na primer, ako je atribut *visina*, može da postoji više studenata sa istom visinom. Neka metoda klase Student bude *oceni studenta* gde se kao parametar funkcije prosleđuje ocena koju je student dobio. Opet, svakog studenta možemo da ocenimo kako želimo, Pera može da bude ocenjen ocenom 10, a Mika i Žika ocenom 9.

Po dogovoru, nazivi *klasa* pišu se velikim početnim slovom, dok se nazivi *atributa* i *metoda* pišu malim slovima.

Konstruktor

Konstruktor je posebna vrsta metoda koja nema povratnu vrednost (nije čak ni void). Ime konstruktora ne može biti proizvoljno nego mora biti isto kao ime klase kojoj pripada. Svaki put kada se kreira novi objekat neke klase pozove se njen konstruktor. Njegova namena je inicijalizovanje vrednosti atributa klase.

Podrazumevani konstruktor je konstruktor bez argumenata. Ako eksplicitno ne napišemo konstruktor, kompajler će sam napraviti podrazumevani konstruktor sa praznim telom. Ukoliko napišemo barem jedan konstruktor, ne generiše se podrazumevani.

Konstruktor sa argumentima predstavlja drugu vrstu konstruktora. Svaki konstruktor može imati proizvoljan broj argumenata. Argument može da se zove isto kao i atribut kojoj konstruktor pripada, ali tada, ako iz tela konstruktora želimo da pristupimo atributu, moramo da koristimo ključnu reč *this*.

Konstruktor kopije se koristi ukoliko želimo da napravimo kopiju nekog objekta koristeći konstruktor. Kopija objekta znači da će atributi novog objekat u trenutku kreiranja imati iste vrednosti kao atributi objekta od kog pravimo kopiju. Međutim, ta dva objekta su dva nezavisna objekta. Ukoliko nakon kreiranja promenimo neki nestatički atribut jednog objekta, isti atribut drugog objekta se ne menja.

Modifikatori pristupa

U javi postoje četiri modifikatora pristupa koji označavaju dostupnost *(scope)* podataka ispred kojih stoje:

1. **private** – podatak je dostupan samo unutar klase; ako je bilo koji konstruktor klase private, nećemo moći napraviti objekat te klase izvan klase
2. **default** – podrazumeva se ukoliko ne navedemo ni jedan drugi modifikator; tada podatak ne može da se vidi van paketa
3. **protected** – podatak može da se vidi van paketa, ali samo ako se radi o nasleđivanju
4. **public** – podatak je vidljiv svugde, unutar i izvan klase, odnosno paketa

Klasa ne može imati *private* niti *protected* modifikator. Ukoliko nam je potrebno da vidimo ili izmenimo podatak van klase, a on je na primer *private* potrebno je implementirati metode koje barataju tim paketom koje se nazivaju **getter** i **setter** metodama.

Varijable

Ime je niz od jednog ili više karaktera. Mora počinjati slovom i mora biti u potpunosti sastavljeno od slova, brojeva i donje crte ("_"). Java koristi Unicode skup karaktera.

Programi upravljaju podacima sačuvanim u memoriji. U mašinskom jeziku podaci se mogu pozvati samo pozivom numeričke adrese memorijske lokacije na koju je podatak smešten. Sa druge strane, u Javi se za poziv podataka koriste imena umesto adresa. Ime koje služi za pozivanje podataka koji su sačuvani negde u memoriji naziva se **varijabla**. Usko gledano, varijabla nije ime samog podatka nego mesta u memoriji koje čuva podatak. Pošto se podatak na jednom mestu u memoriji može menjati, varijabla će se odnositi na različite vrednosti, ali uvek na isto mesto!

Svaka varijabla ima svoj tip. Primitivni tipovi u Javi su:

- byte
- short
- int
- long
- float
- double
- char
- boolean

Prva četiri tipa su celobrojni, a razlikuju se po rasponu brojeva koje podržavaju. Float i double čuvaju realne brojeve, a razlikuju se po opsegu i preciznosti. Char čuva jedan znak iz Unicode skupa, a boolean jednu logičku vrednost – true ili false.

Primeri ispravnog deklarisanja varijabli:

tip_varijable ime_varijable;

```
int a;
char b;
double myFirstDoubleVariable;
```

Primeri dodele vrednosti varijablama:

```
a = 5;
char b = 'b';
```

Primer

```
// Rectangle.java
package rtrk.pnrs;

public class Rectangle {
	private int a;			// a i b se vide samo u klasi Rectangle
	private int b;
	protected int var1;		// var1 se vidi samo u paketu
	public int var2;		// var2 se vidi i van klase ili paketa
	public static int var3;

	public Rectangle() {	/* podrazumevani konstruktor */
		a = 0;
		b = 0;
		var1 = 0;
		var2 = 0;
		var3 = 0;
	}

	public Rectangle(int a, int b) {		/* konstruktor sa argumentima */
		this.a = a;		/* this.a se odnosi na atribut a */
		this.b = b;
		var3 = 0;
	}

	public Rectangle(Rectangle r) {		/* konstruktor kopije */
		a = r.a;
		b = r.b;
		var1 = r.var1;
		var2 = r.var2;
	}
```

```
	public int getVar1() {	/* getter za promenljivu var1 */
	return var1;
	}

	public void setVar1(int var1) {	/* setter promenljive var1 */
	this.var1 = var1;
	}

	public int area() {
	return a*b
	};
	public int perimeter() {
	return 2*a + 2*b;
	}
}

// Test.java
package rtrk;
import rtrk.pnrs.Rectangle;
public class Test {

	public static void main(String[] args) {
		Rectangle r1 = new Rectangle();
		System.out.println("r1 area: " + r1.area());

		Rectangle c2 = new Rectangle(1, 2);
		System.out.println("r2 area: " + r2.povrsina());
		System.out.println("r2 perimeter: " + r2.perimeter());

		System.out.println("r1.var1 = " + r1.var1);
		System.out.println("r1.var1 = " + r1.getVar1());
		System.out.println("r1.var2 = " + r1.var2);

		r1.var3 = 5;
		System.out.println("r1.var3 = " + r1.var3);
		System.out.println("r2.var3 = " + r2.var3);

		r2.var3 = 100;
		System.out.println("r1.var3 = " + r1.var3);
		System.out.println("r2.var3 = " + r1.var3);
```

```
                Rectangle r3 = new Rectangle(r2);
                System.out.println("r2.var1 = " + r2.getVar1());
                System.out.println("r3.var1 = " + r3.getVar1());
                r3.setVar1(55);
                System.out.println("r2.var1 = " + r2.getVar1());
                System.out.println("r3.var1 = " + r3.getVar1());
        }
}
```

Pozicionirati se u jedan direktorijum pre direktorijuma *rtrk* i pozvati **javac** komandu nad datotekom Test.java da bi se ove dve datoteke kompajlirale (Rectangle.java je *import*-ovan u Test.java pa će kompajliranje Test.java povući i kompajliranje Rectangle.java). Zatim pozvati komandu **java** za pokretanje programa.

Ovaj program se neće uspešno kompajlirati zbog linije

```
System.out.println("r1.var1 = " + r1.var1);
```

jer je modifikator pristupa var1 *protected*, a nije reč o nasleđivanju. Ukoliko se ta linija izostavi kompajliranje će biti uspešno.

```
>> javac Test.java
>> java rtrk.Test

r1 area: 0
r2 area: 2
r2 perimeter: 6
r1.var1 = 0
r1.var2 = 0
r1.var3 = 5
r2.var3 = 5
r1.var3 = 100
r2.var3 = 100
r2.var1 = 0
r3.var1 = 0
r2.var1 = 0
r3.var1 = 55
```

Komentar:

Kako bi se napravio objekat neke klase potrebno je pozvati ključnu reč **new** koju prati poziv konstruktora. Kada se pravi objekat r1 poziva se podrazumevani konstruktor

```
public Rectangle() {
```

```
        a = 0;
        b = 0;
        var1 = 0;
        var2 = 0;
        var3 = 0;
}
```

koji će da inicijalizuje atribute a, b, var1 i var2 na 0 za promenljivu r1. Kada posle toga pokušamo da ispišemo površinu kvadrata r1 ona će iznositi a*b = 0*0 = 0. Takođe, i ispis promenljive var2 na standarni izlaz ispisuje 0. Za pristupanje atributu var1 potrebno je pozvati get metodu jer je njegov modifikator *protected*. Modifikator pristupa atributa var2 je *public* pa možemo da mu pristupimo bez get metode čak i van paketa ili klase. Varijabla var3 je *statička* varijabla, to znači da, ukoliko jedan objekat (na primer r1) promeni varijablu var3, i drugi objekat (r2) će videti tu promenu, dok se to neće desiti sa ostalim varijablama, jer nisu statičke. Objekat r3 se pravi putem konstruktora kopije, te su mu nestatički atributi isti kao nestatički atributi objekta r2. Ukoliko setterom promenimo vrednost atributa var1 za objekat r3, isti atribut za objekat r2 se neće promeniti. Vrednost atributa var1 moramo menjati koristeći set metodu, jer je njen modifikator pristupa *protected*.

Reference

U Javi ne postoji operator * (koji u C-u služi za pristupanje vrednosti na koju pokazuje pokazivač). Kada deklarišemo promenljivu tako da bude tipa neke klase (npr. Rectangle r1) time nismo napravili nikakav objekat te klase, niti smo dodelili memoriju za smeštanje objekta. Samo smo rekli da je r1 varijabla koja *može da čuva* **referencu** na objekat klase Rectangle. Izdvojeno je samo parče memorije koje čuva *referencu (adresu) objekta*.

Da bismo kreirali novi objekat klase, moramo da koristimo ključnu reč **new** koju prati poziv odgovarajućeg konstruktora (*new Rectangle()* – za poziv praznog konstruktora, odnosno *new Rectangle(1, 2)* – za poziv konstruktora koji prima dva int parametra). Nakon ovoga, izdvaja se memorija neophodna za *smeštanje objekta* klase Rectangle, tj kreira se po jedan primerak svakog od *nestatičkih* atributa klase i izvrši se telo konstruktora da bi se oni inicijalizovali. Rezultat je **referenca na tako dobijeni objekat**. Ukoliko sada tu referencu sačuvamo u promenljivu tipa Rectangle koju smo prethodno definisali (r1 = new Rectangle()) uspostavljamo vezu između promenljive r1 i upravo kreiranog objekta klase Rectangle.

Kada želimo da raskinemo tu vezu možemo da kažemo da je *r1=null*, tada promenljiva r1 ne pokazuje ni na šta (nije u vezi ni sa jednim objektom). Takođe, možemo promenljivu r1 da povežemo sa drugim objektom (r1 = Rectangle(1,2)).

Nizovi

Niz je konačan skup podataka istog tipa koji se u memoriji čuva na uzastopnim lokacijama. Elementima nizova možemo da pristupamo po indeksima. Indeks prvog elementa je uvek 0. Ako želimo da radimo sa nizovima, moramo da znamo kog tipa su elementi niza.

Za početak, deklarišemo promenljivu a za čuvanje reference na niz int-ova: int a[] ili int []a. U ovom trenutku još uvek ne znamo koliko elemenata naš niz ima i nije izdvojena memorija. Da bismo odvojili parče memorije koje će da čuva naš niz moramo da zadamo broj elemenata niza.

```
int a[] = new int[5];
```

Sada imamo korektno definisan niz od pet elemenata tipa int. Dalje, sa njim možemo da manipulišemo na standardan način – menjamo vrednosti elemenata, iteriramo kroz niz i ispisujemo ih...

Stringovi

U Javi postoji klasa String koja je paket *java.lang* biblioteke. Postoje dva načina pravljenja stringova:

```
String str1 = new String("hello");
String str2 = "hello";
```

U prvom slučaju pravimo novi objekat klase String, a u drugom slučaju pravimo referencu. Međutim, ukoliko imamo pet varijabli tipa String sa istom vrednošću koje smo napravili pomoću ključne reči *new*, na heap-u se stvara pet novih objekata sa istom vrednošću. Sa druge strane, ukoliko se tih pet promenljivih napravi na drugi način, na heap-u će se napraviti samo jedan objekat.

Upoređivanje stringova

Ukoliko stringove upoređujemo na klasičan način, pomoću operatora jednakosti (==), mi ćemo zapravo upoređivati njihove reference, odnosno, proveravaćemo da li predstavljaju isti objekat. Kako bismo zapravo uporedili vrednosti dva stringa, koristimo metodu klase String *equals()*.

Primer:

```
String s1 = new String("hello");
String s2 = "hello";
String s3 = "hello";
System.out.println(s1 == s2);      // false
System.out.println(s2 == s3);      // true
System.out.println(s1.equals(s2)); // true
```

Prvi ispis upoređuje reference objekata s1 i s2, a pošto je s1 napravljen pomoću ključne reči *new*, napraviće se novi objekat. Prilikom pravljenja promenljive s3 na heap-u se neće stvoriti novi objekat, jer takav objekat sa istom vrednošću već postoji (s2). U poslednjem slučaju, proveravaju se **vrednosti** objekata s1 i s2, a one su iste (hello) pa je rezultat upoređivanja *true*.

Upravljačke naredbe

Upravljačke naredbe se koriste za kontrolisanje toka programa i njegovog grananja na osnovu promene stanja programa. Upravljačke naredbe u Javi:

- uslovne naredbe
- naredbe ciklusa
- naredbe skoka

Uslovne naredbe

Java podržava dve vrste uslovnih naredbi – **if** i **switch**.

Opšti oblik naredbe **if**:

```
if (uslov)
        naredba1
else if
        naredba2
else
        naredba3
```

Primer:

```
if (a < b)
        a = 5;
else if (b > a)
        b = 5;
else
{
        a = 15;
        b = 15;
}
```

Da bi se izbegli ugnježđeni uslovi if – else – if uvodi se naredba **switch**.

Opšti oblik naredbe **switch**:

```
switch(izraz) {
        case vrednost1:
                naredba1
                break;
        case vrednost2:
                naredba2
                break;
        …
        case vrednostN:
                naredbaN
                break;
        default
```

```
                narebaN+1
                break;
}
```

Primer:

```
switch(dan) {
        case 1:
                System.out.println("Ponedeljak");
                break;
        case 2:
                System.out.println("Utorak");
                break;
        case 3:
                System.out.println("Sreda");
                break;
        case 4:
                System.out.println("Cetvrtak");
                break;
        case 5:
        case 6:
        case 7:
                System.out.println("Vikend!!!");
                break;
        default:
                System.out.println("U nedelji ima 7 dana!");
}
```

Naredbe ciklusa

Naredbe ciklusa u Javi su: **for**, **while** i **do-while**. Ovim naredbama se prave petlje koje izvršavaju zadati skup instrukcija onoliko puta koliko je potrebno da se zadati uslov ispuni.

While petlja izvršava naredbu dok je zadati uslov tačan. Ukoliko na početku uslov nije tačan, telo petlje se neće izvršiti ni jednom.

Opšti oblik:

```
while(uslov)
        naredbe
```

Primer:

```
int a = 0;
int b = 0;
while (a != 0) {
```

```
        b++;
}
```

Ova petlja se neće izvršiti ni jednom, jer se uslov proverava na početku i odmah na početku uslov je netačan.

Telo **do-while** petlje će se uvek izvršiti barem jednom jer se uslov proverava tek na kraju petlje.

Opšti oblik:

```
do
        naredbe
while(uslov)
```

Primer:

```
int a = 0;
int b = 0;
do {
        b++;
} while(a != 0);
```

Ova petlja će se izvršiti jednom, jer se uslov proverava tek na kraju.

Kada **for** petlja započne rad, na početku se inicijalizuje varijabla koja predstavlja brojač ciklusa, zatim se ispituje uslov petlje koji proverava da li je brojač došao do kraja i, ukoliko je uslov tačan, brojač se menja. Kada se promenljiva definiše unutar petlje, njen životni vek prestaje sa krajem petlje.

Primeri:

```
for(int i = 0; i < 10; i++)
for(int j = 10; j > 0; j--)
for(int k = 0; k < 20; k+=2)
for(a=0, b=50; a<b; a++, b--)
```

Naredbe skoka

Ove naredbe omogućavaju da se promeni tok izvršavanja programa. U Javi postoje tri naredbe skoka: break, continue i return, od kojih nam je najbitnija break naredba. Ona u najvećem broju slučajeva služi za:

1. završavanje niza naredbi u naredbi *switch*
2. izlazak iz petlje.

Kolekcije u Javi

Korišćenje nizova u Javi je pogodno ukoliko znamo koliki broj elemenata će taj niz imati pa se, samim tim, unapred zna koliko uzastopnih lokacija u memoriji treba zauzeti za njega. Međutim, problem nastaje ukoliko u trenutku kreiranja niza ne znamo koliko elemenata će taj niz imati ili ukoliko veličina niza

treba da se menja u toku programa (dodavanje/brisanje elemenata).

Kao rešenje ovog problema uvedene su **kolekcije** – objekti koji se koriste za smeštanje i manipulaciju nedefinisanim brojem objekata. Različite vrste kolekcija imaju različite načine manipulacije elementima smeštenim u njima. Takođe, kolekcije mogu da manipulišu raznim tipovima podataka, kako prostim tipovima, tako i objektima klasa. Sve kolekcije imaju **iterator** koji prolazi kroz sve elemente u kolekciji.

Liste

Liste su implementirane pomoću *java.util.List* interfejsa. Definišu listu u kojoj su elementi uređeni u specifičnom redosledu i u kojoj su dozvoljeni duplikati elemenata. Elementi se mogu umetnuti na specifičnu poziciju i moguća je pretraga po listi. Implementirane su u klasama *java.util.ArrayList* i *java.util.LinkedList*. Neke od bitnijih metoda koje liste podržavaju:

- dodavanje elemenata na određeno mesto

  `void add(int index, Object element)`
- dobavljanje elementa sa određene pozicije

  `Object get(int index)`
- izmena elementa na određenoj poziciji

  `Object set(int index, Object element)`
- brisanje elementa sa određene pozicije

  `Object remove(int index)`
- brisanje svih elemenata

  `void clear()`
- dobavljanje pozicije određenog elementa

  `int indexOf(Object element)`

ArrayList implementira listu kao niz. Moguć je pristup bilo kom elementu jer radi na principu pristupanja po indeksu. Manipulacija ArrayList-om je spora jer je potrebno mnogo pomeranja po listi kada je potrebno ukloniti neki od elemenata.

Primer korišćenja ArrayList:

```
import java.util.ArrayList;

public class Test {

	public static void main(String[] args) {

		ArrayList<String> al = new ArrayList<String>();
```

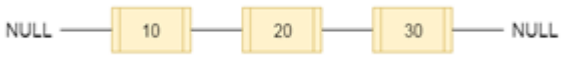
```
                al.add("apple");
                al.add("orange");
                al.add("banana");

                System.out.println("Element at second position: " + al.get(1));

                al.set(1,"strawberry");
                System.out.println("Element at second position: " + al.get(1));

                System.out.println("All elements: ");
                for(String s:al) {
                        System.out.println(s);
                }

                al.remove(2);
                System.out.println("All elements after remove: ");
                for(String s:al) {
                        System.out.println(s);
                }

                al.clear();
                System.out.println("All elements after clear: ");
                for(String s:al) {
                        System.out.println(s);
                }
        }
}
```

Izlaz koji se dobije nakon pokretanja:

```
Element at second position: orange
Element at second position: strawberry
All elements:
apple
strawberry
banana
All elements after remove:
apple
strawberry
All elements after clear:
```

LinkedList koristi mehanizam dvostrukto spregnute liste, odnostno skladišti elemente u čvorovima koji imaju pokazivače na prethodni i naredni element liste. Manipulacija listom je brza jer nema

potrebe za prolaženjem kroz celu listu u slučaju brisanja elementa, pokazivači vode računa o tome. U dvostruko spregnutoj listi novi element možemo da dodajemo sa obe strane određenog elementa:



Za manipulaciju LinkedList koristimo metode *addFirst, addLast, getFirst* i *getLast* za dodavanje odnosno dobavljanje prvog ili poslednjeg elementa liste.

Stek

Stek se implementira pomoću *java.util.Stack* interfejsa. Stek predstavlja LIFO (last-in-first-out) sistem. Stek se kreira prazan. Neke od metoda koje podržava:

- dodavanje elementa na vrh steka

  `Object push(Object element)`
- uklanjanje elementa sa vrha steka (poslednji dodat element)

  `Object pop()`
- provera da li je stek prazan

  `boolean empty()`
- pretraživanje steka

  `int search(Object element)`
- dobavljanje elemenata sa vrha steka (bez njegovog uklanjanja iz steka)

  `Object peek()`

Primer korišćenja steka:

```
import java.util.Stack;

public class Test {

	public static void main(String[] args) {

		Stack<Integer> stack = new Stack<Integer>();

		System.out.println("Is stack empty? " + stack.empty());

		stack.push(100);
		stack.push(200);
		stack.push(300);
		stack.push(400);
```

```
                System.out.println("Elements: " + stack);
                System.out.println("Pop: " + stack.pop());
                System.out.println("Elements after pop: " + stack);
                System.out.println("Search: " + stack.search(100));
                System.out.println("Peek: " + stack.peek());
        }
}
```

Izlaz nakon pokretanja:

```
Is stack empty? true
Elements: [100, 200, 300, 400]
Pop: 400
Elements after pop: [100, 200, 300]
Search: 3
Peek: 300
```

Red

Red se implementira pomou *java.util.Queue* interfejsa. Elementi se skladište u redosledu u kom su unešeni, odnosno dodaju se na kraj, a skidaju se sa početka reda implementirajući FIFO (first-in-first-out) sistem. Neke od metoda koje se koriste:

- dodavanje elementa

  `boolean add(Object element)`
- brisanje prvog elementa

  `Object remove()`
- brisanje prvog elementa uz vraćanje *null* ako je red prazan

  `Object poll()`
- dobavljanje, bez brisanja, prvog elementa

  `Object element()`
- dobavljanje, bez brisanja, prvog elementa, uz vraćanje *null* ako je red prazan

  `Object peek()`

LinkedList pored *List* interfejsa implementira i *Queue* interfejs.

ArrayQueue implementira red pomoću niza.

BlockingQueue implementira red, međutim nudi fleksibilnije rukovanje redovima u smislu da dozvoljava da se, pre umetanja elementa u red, proveri da li ima slobodnih mesta i, ukoliko nema, da se sačeka izvesno vreme da se oslobodi prostor. Takođe, kada se pokuša skidanje elementa iz praznog reda, čeka se

dok se on ne pojavi.

PriorityQueue omogućava sortiranje elemenata po prioritetu koji se određuje na osnovu implementirane metode u konstruktoru.

Primer korišćenja reda:

```
import java.util.PriorityQueue;

public class Test {

	public static void main(String args[]) {
		PriorityQueue<Integer> queue = new PriorityQueue<Integer>();

		System.out.println("Head before add: " + queue.peek());

		queue.add(100);
		queue.add(200);
		queue.add(300);
		queue.add(400);
		queue.add(500);

		System.out.println("Head: " + queue.element());
		System.out.println("Head: " + queue.peek());

		System.out.println("Queue: " + queue);

		queue.remove();
		queue.poll();
		System.out.println("After removing two elements: " + queue);
	}
}
```

Izlaz nakon pokretanja:

```
Head before add: null
Head: 100
Head: 100
Queue: [100, 200, 300, 400, 500]
After removing two elements: [300, 400, 500]
```

Red sa dva kraja

java.util.Deque interfejs omogućava kreiranje dvostruko spregnutih redova. Dok obični redovi dozvoljavanju dodavanje elemenata samo na kraj reda i skidanje samo sa početka reda, dvostruki red

omogućava da se to vrši sa oba kraja. *ArrayDeque* i *LinkedList* implementiraju ovaj interfejs.

Primer korišćenja ArrayDeque:

```
import java.util.ArrayDeque;

public class Test {

	public static void main(String args[]) {
		ArrayDeque<String> deque = new ArrayDeque<String>();

		deque.offer("apple");
		deque.offer("banana");
		deque.add("strawberry");

		System.out.println("Deque: " + deque);

		deque.offerFirst("raspberry");

		System.out.println("Deque afrer offerFirst: " + deque);

		deque.pollLast();
		System.out.println("Deque afrer pollLast: " + deque);
	}
}
```

Izlaz nakon pokretanja:

```
Deque: [apple, banana, strawberry]
Deque after offerFirst: [raspberry, apple, banana, strawberry]
Deque afrer pollLast: [raspberry, apple, banana]
```

Mape

Mape su kolekcije koje asociraju vrednosti sa ključevima. Ključevi se koriste za pronalaženje vrednosti. *HashMap* je jedan primer mape koja koristi hash tabelu. Metode koje se koriste:

- dodavanje para ključ-vrednost u mapu

  `Object put(Object key, Object value)`

- dodavanje mape u mapu

  `void putAll(Map map)`

- brisanje para ključ-vrednost na osnovu ključa

`Object remove(Object key)`

- dobavljanje vrednosti na osnovu ključa

  `Object get(Object key)`

- ispitivanje da li određeni ključ postoji

  `boolean containsKey(Object key)`

Primer korišćenja HashMap:

```
import java.util.HashMap;
import java.util.Map;

public class Test {

	public static void main(String args[]) {
		HashMap<Integer,String> hm = new HashMap<Integer,String>();
		hm.put(100, "apple");
		hm.put(200, "banana");
		hm.put(300, "strawberry");

		System.out.println("Map: ");
		for(Map.Entry m:hm.entrySet()) {
			System.out.println(m.getKey()+" "+m.getValue());
		}

		hm.remove(200);
		System.out.println("Map: ");
		for(Map.Entry m:hm.entrySet()) {
			System.out.println(m.getKey() + " " + m.getValue());
		}

		System.out.println("Value with key 100: " + hm.get(100));
		System.out.println("Does map contain key 400? " + hm.containsKey(400));
	}
}
```

Izlaz nakon pokretanja:

```
Map:
100 apple
200 banana
300 strawberry

Map:
```

```
100 apple
200 strawberry

First value: apple
Does map contain key 400? false
```

Polimorfizam i nasleđivanje

Nasleđivanje je postupak kojim se iz postojećih klasa izvode nove klase. Nova (izvedena) klasa imaće sve metode i atribute stare (bazne) klase, ali moguće joj je dodati nove metode/atribute. Java ne dopušta višestruko nasleđivanje – klasa može da nasledi najviše jednu klasu.

Primer u kom klasa B nasleđuje klasu A – A je bazna klasa, a B izvedena:

```
public class A {
	public int a;
}

public class B extends A {
	public int b;
}
```

Klasa B će pored varijable b imati i varijablu a. Mada klasa B sadrži sve članove bazne klase A, ona ne može da pristupi članovima klase A čiji modifikator pristupa je private.

Kada izvedena klasa treba da se obrati baznoj klasi koristi se rezervisana reč super. Ova reč se koristi u dva slučaja:

- ako se pristupa konstruktoru bazne klase: super(parametri ako postoje)
- ako se pristupa članu bazne klase koji je skriven od izvedene klase: super.imeČlana

Polimorfizam

Polimorfizam predstavlja mogućnost objekta da ima različite oblike. U primeru su definisane dve klase – Pas i Mačka, koje nasleđuju klasu Životinja i njenu klasu zvuk(). Promenljiva koja se koristi za poziv poliformne metode je tipa Životinja. Metoda zvuk() je različito implementirana u sve tri klase. Kada se ta metoda pozove, biće izvršena verzija iz one klase na čiji objekat se čuva referenca u promenljivoj tipa bazne klase.

Primer:

```
public class Animals {
	public String say() {
		return "";
	}
}
```

```
public class Cat extends Animals {
	public String say() {
		return "meow!";
	}
}

public class Dog extends Animals {
	public String say() {
		return "barf!";
	}
}

public class Test {

	public static void main(String args[]) {
		Animals a1 = new Cat();
		Animals a2 = new Dog();

		System.out.println("a1: " + a1.say());
		System.out.println("a2: " + a2.say());
	}
}
```

```
a1: meow!
a2: barf!
```

Apstraktne klase

U prethodnom primeru, metoda *say()* klase Animals imala je prazno telo, jer ne znamo kako da je implementiramo – ne postoji univerzalni način oglašavanja životinja.

Ukoliko metoda nema telo ona je **apstraktna**.

```
public abstract int a();
```

Ukoliko klasa sadrži bar jednu apstraktnu metodu i sama je **apstraktna**.

```
public abstract class A {...}
```

Apstraktna klasa ne može da ima objekte. Može se definisati promenljiva tipa apstraktne klase. Ako imamo klasu koja nasleđuje apstraktnu klasu, kako bismo mogli napraviti objekat te klase moramo implementirati sve metode nasleđene od apstraktne klase.

Interfejsi

Interfejs sadrži samo deklaraciju metode, bez njene definicije. Za metode interfejsa se podrazumeva da su *public* i *abstract*. Za razliku od nasleđivanja, klasa može da *implementira* jedan ili više interfejsa. Da bismo mogli da pravimo objekte klase koja implementira interfejs, ona mora da sadrži implementaciju svih metoda deklarisanih u interfejsu.

Primer u kom klasa B implementira interfejs A:

```
public interface A {
	int a();
}

public class B implements A {
	@Override
	public int a() {
		System.out.println(“a”);
	}
}
```

Ako izvedena klasa ima istu metodu koja je definisana u baznoj klasi, to se zove **preklapanje (overriding) metoda**. **@Override** označava da želimo da implementiramo metodu koja ima isti potpis (ime, povratnu vrednost i parametre) kao metoda interfejsa A.