POGLAVLJE 2

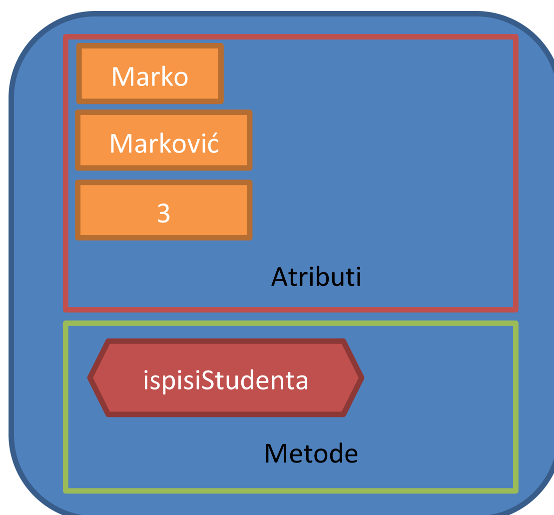
Objektno programiranje

2.1. Objekat

Objektno programiranje predstavlja programsku paradigmu koja se zasniva na objektima kao osnovnim komponentama. **Objekat** sadrži podatke koji ga opisuju i funkcije koje vrše operacije nad podacima objekta. Podatke zovemo **atributi** objekta. Funkcije zovemo **metode** objekta.

Na ovaj način, objekat centralizuje stanje (putem atributa) i ponašanje (putem metoda) nekog programskog entiteta u okviru jedne programske komponente.

Na primer, određenog studenta možemo programski da predstavimo kao jedan objekat. Njegovo ime, prezime i godina studija bi bili podaci o njemu. Ukoliko je potrebno ispisati podatke o ovom studentu, to se može uraditi kroz funkciju koja je takođe deo objekta. Ovo je ilustrovano na slici.

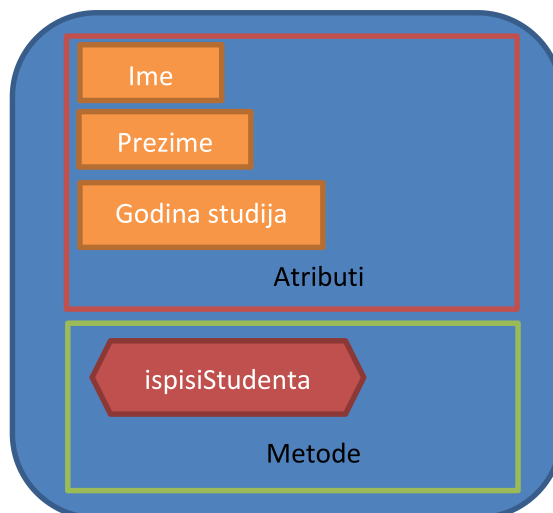


2.2. Klasa

Ako posmatramo prethodni primer, moguće je kreirati više objekata koji predstavljaju studente. Svi ovi različiti objekti su opisani istim skupom podataka i nad svima je moguće izvršiti iste operacije (npr. ispis, računanje prosečne ocene, ...). Možemo formalno definisati koji skup podataka će svaki student sadržati i koje će operacije biti moguće izvršiti nad njim. Ovakav formalan opis se naziva **klasa**. Klasa je šablon na osnovu kojeg se kreiraju konkretni objekti. Objekat se kreira na osnovu određene klase, koja definiše koju strukturu će objekti te klase da imaju. Kažemo da je objekat **instanca** (primerak) klase. Objekti će sadržati konkretne podatke. Klasa samo definiše koje podatke će uopšte biti moguće evidentirati za objekat, kao i koje operacije se nad tim podacima vrše. Dakle, klasa formalno specificira naziv i tip atributa objekta.

Definisanjem klase je definisan novi tip podatka u programu. Svaki objekat određene klase je promenljiva tipa predstavljenog klasom. Na primer, objekat klase Student je tipa Student. Sve instance iste klase su objekti istog tipa. Pošto tip podatka predstavljen klasom u sebi sadrži više podataka i funkcija, ovakve tipove zovemo složeni tipovi. Dakle, klasa je mehanizam da se u program uvede novi proizvoljan tip podatka. U klasi je definisano kakve podatke i ponašanje će imati promenljive tog tipa.

Za objekat prikazan na prethodnoj slici, odgovarajuća klasa je prikazana na sledećoj slici.



Primetimo da klasa definiše koji podaci će se moći evidentirati za objekat, ali ne i same podatke. Konkretni podaci su specifični za svaki pojedinačan objekat te klase (npr. Marko Marković koji je na trećoj godini studija, kao u prethodnom primeru).

Klasa definisana na slici se u Javi može napisati sledećim programskim kodom.

```
class Student {
	String ime;
	String prezime;
	int godinaStudija;

	void ispisiStudenta() {
		System.out.println(ime + " " + prezime
			+ " - godina studija: " + godinaStudija);
	}
}
```

Klasu je obavezno definisati u posebnom fajlu koji mora da ima isti naziv kao naziv klase.

U klasi je moguće definisati i više metoda sa istim nazivom. Ovo se naziva **preklapanje metoda** (eng. *method overloading*). Ove metode se moraju razlikovati po parametrima koje primaju (po broju ili tipu parametara). Nije dozvoljeno definisati više metoda koje imaju isti naziv i parametre, a samo se razlikuju po povratnom tipu. Razlog je taj što pozivaoc metode ne mora da preuzme povratnu vrednost, pa se u tom slučaju ne bi znalo koja od metoda sa istim imenom i parametrima se poziva.

2.3. Rad sa objektima

Kada je klasa definisana, moguće je u programu kreirati objekte ove klase. Pre kreiranja objekta, pogledajmo kako izgleda deklaracija objekta klase koja je prikazana u narednom primeru.

```
Student s;
```

Promenljiva s je deklarisana kao objekat tipa Student. Ipak, još uvek nije izvršeno kreiranje objekta, tako da sa ovom promenljivom nije moguće raditi u programu. Ona nema vrednost objekta tipa student, već vrednost null. Null vrednost je oznaka da promenljiva, deklarisana kao objekat određenog tipa, ne sadrži sam

objekat. Samim tim, koristeći promenljivu koja ima null vrednost nije moguće pristupiti ili definisati podatke objekta niti pozvati njegove metode.

Da bi se objekat kreirao, potrebno je pozvati konstruktor objekta. Sintaksno, konstruktor je funkcija, koja ima isti naziv kao ime klase i nema definisan povratni tip (nije ni void, nego uopšte nema povratni tip). Kasnije ćemo videti kako sami možemo da definišemo konstruktor. Ako sami ne definišemo konstruktor, kompajler automatski kreira konstruktor. Tako da objekat klase Student iz prethodnog primera možemo kreirati na sledeći način:

```
Student s = new Student();
```

Kao što je poznato, operator dodele upisuje vrednost izraza sa svoje desne strane u promenljivu koja se nalazi na levoj strani. Dakle u promenljivu s se smešta rezultat izraza sa desne strane. Izraz sa desne strane predstavlja poziv konstruktora klase Student. Ključnom rečju *new* se vrši poziv konstruktora. Kao što smo naveli, naziv konstruktora je isti kao naziv klase (u ovom primeru Student). Ovaj postupak kreiranja objekta korišćenjem ključne reči *new* nazivamo instanciranje klase.

Sada postoji objekat klase Student i za sada možemo smatrati da je uskladišten u promenljivoj s. Šta je zaista uskladišteno u promenljivoj s biće objašnjeno u narednom poglavlju. Sada promenljiva s "sadrži" sve što je definisano u klasi Student da će objekti klase Student sadržati. To su ime, prezime i godina studija studenta. Takođe, sadrži i metodu ispisiStudenta koja se može pozvati. U primeru je prikazano kako se može pristupiti atributima objekta i pozvati metoda objekta.

Postavljanje vrednosti atributa

```
s.ime = "Marko";
s.prezime = "Markovic";
s.godinaStudija = 3;
System.out.println("Prezime studenta je ": + s.prezime);
s.ispisiStudenta();
```

Preuzimanje vrednosti atributa

Poziv metode objekta

Prikazani primer će ispisati najpre Markovic kao prezime studenta. Zatim će pozivom metode *ispisiStudenta* biti ispisano `Marko Markovic – godina studija: 3`.

Moguće je kreirati više instanci iste klase. Tako možemo kreirati drugi objekat klase Student i postaviti neke druge podatke koje će taj objekat da skladišti. U tom slučaju, pri pozivu metode će se ispisati podaci uskladišteni u atribute objekta nad

kojim se metoda poziva. Objekti iste klase su kreirani na osnovu istog šablona što znači da skladište isti broj podataka određenog tipa i da je nad svima moguće izvršiti iste operacije. Ponavljamo, klasa definiše novi tip podatka, pa je odnos između dva objekta iste klase analogan odnosu koju imaju dve promenljive istog tipa. Npr. ako postoje dve promenljive tipa int, one mogu da skladište različite vrednosti, ali im je tip vrednosti koju skladište isti.

Dakle, svaki objekat je jedan podatak složenog tipa. Klasa predstavlja ovaj složeni tip. Kao i kod podataka primitivnog tipa, i podatke složenog tipa možemo da uskladištimo u promenljivima. Tako je moguće u promenljivu složenog tipa upisati vrednost druge promenljive odgovarajućeg tipa. Ovo je prikazano u primeru.

```
Student s1 = new Student();
s1.ime = “Marko“;
Student s2 = new Student();
s2.ime = “Petar“;
s1 = s2;
System.out.println(“Ime studenta s1 je: “ + s1.ime);
```

Program će ispisati vrednost Petar, jer je u promenjivu s1 upisana promenljiva s2 za koju je u atribut ime ranije upisano Petar.

Kao što smo ranije spomenuli, String je složeni tip, tj. klasa. Kao što vidimo, atributi objekta ne moraju biti primitivnog, nego mogu biti i složenog tipa. Kao što smo postavili atribut tipa String, moguće je definisati i atribut nekog drugog složenog tipa. Npr. grad možemo predstaviti posebnom klasom, pa informaciju o gradu u kojem student živi možemo uskladištiti u objekat klase Grad koji je atribut klase Student. Ovo je prikazano u primeru.

```
class Student {
      String ime;
      String prezime;
      int godinaStudija;
      Grad prebivaliste;

      void ispisiStudenta() {
            System.out.println(ime + “ “ + prezime
                  + “; živi u: “ + prebivaliste.naziv
                  + “; godina studija: “ + godinaStudija);
      }
}
```

2.4. Reprezentacija promenljivih u memoriji

Kao što smo ranije pomenuli, glavna memorija je niz bajtova. Promenljive koje se definišu u programu se skladište u glavnoj memoriji računara. Java specifikacija ne definiše na koji način će na najnižem nivou promenljive biti uskladištene u memoriji. Ovo je ostavljeno implementaciji JVM. Ovde ćemo objasniti kako je ova funkcionalnost standardno implementirana. Radna memorija koju program koristi se sastoji iz dve sekcije:

- *stack* memorija
- *heap* memorija

Obe sekcije skladište podatke, ali je pristup ovim sekcijama drugačiji i unapred je definisano koje promenljive se skladište u kojoj sekciji.

Krenućemo od *stack* sekcije. Lokalne promenljive primitivnog tipa koje metoda definiše u toku rada se skladište u *stack* memoriji. U nastavku je dat primer definisanja dve promenljive i ilustracija njihovog prikaza u memoriji.

```
int a = 5;
int b = 7;
```

Promenljiva	Adresa	Vrednost	
b	43	7	Stack
a	47	5	
		...	
			Heap

Zbog pojednostavljenja, vrednosti su prikazane u decimalnom, a ne binarnom obliku. Pošto int vrednost zauzima 4 bajta (32 bita), treba primetiti da je promenljiva *a* upisana četiri lokacije nakon promenljive *b*.

Ako bi se sada izmenila vrednost neke promenljive, na odgovarajuću memorijsku lokaciju bi bila upisana nova vrednost. Dat je primer izmene vrednosti promenljive *a*.

```
a = 4;
```

Promenljiva	Adresa	Vrednost	
b	43	7	Stack
a	47	4	
		...	
			Heap

Za razliku od promenljivih primitivnog tipa, objekti se skladište u *heap* memoriji. Dat je primer instanciranja objekta klase Student i postavljanja vrednosti u ovaj objekat. Ilustrovan je i sadržaj memorije.

```
Student s = new Student();
s.ime = “Marko”;
s.prezime = “Markovic”;
s.godinaStudija = 3;
```

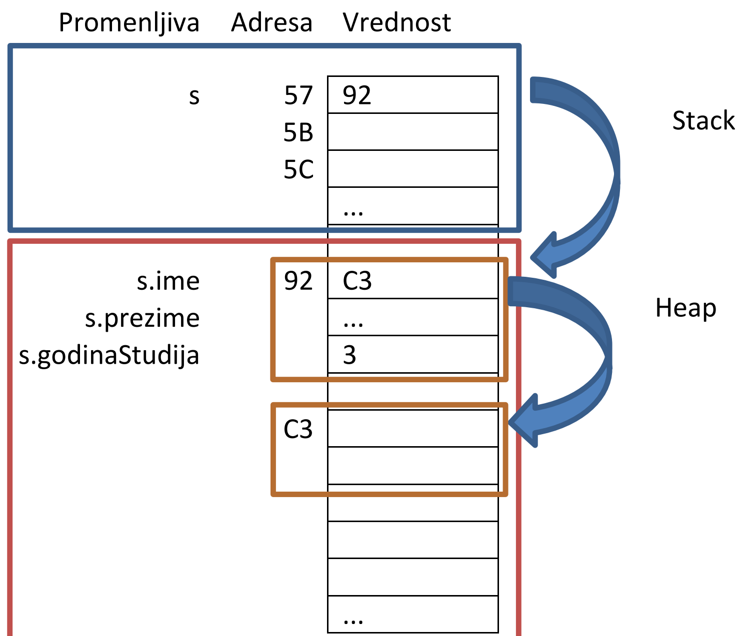
Promenljiva	Adresa	Vrednost	
s	57	92	Stack
	5B		
	5C		
		...	
s.ime	92	...	Heap
s.prezime		...	
s.godinaStudija		3	

Kao što vidimo na slici, u promenljivoj s je uskladištena samo adresa lokacije u heap memoriji na kojoj se nalaze podaci novokreiranog objekta. Iz tog razloga se složeni tipovi zovu i adresni tipovi. Koristi se i termin referentni tipovi, obzirom da se adresa neke memorijske lokacije naziva i referenca na tu memorijsku lokaciju. Dakle, ako sada ponovo pogledamo izraz za kreiranje objekta klase Student, taj izraz izgleda ovako.

```
Student s = new Student();
```

Dakle, u promenljivu s se upisuje rezultat izraza sa desne strane. Izraz sa desne strane je naredba virtuelnoj mašini da na heap memoriji zauzme prostor za objekat klase Student. Koja lokacija će biti zauzeta nije pod kontrolom programera, već Java virtuelna mašina vrši izbor lokacije u skladu sa svojom logikom upravljanja memorijom. Izraz sa desne strane kao rezultat vraća adresu lokacije u heap memoriji na kojoj je objekat kreiran i ta adresa se upisuje u promenljivu s.

U prethodnom primeru nije prikazana vrednost koja se skladišti za promenljive ime i prezime u objektu Student. Ove vrednosti su tipa String. String je klasa (složeni tip), što znači da i za ovaj tip važi isti način skladištenja podataka kao i za klasu Student. Dakle, u promenljivoj tipa String se skladišti samo adresa lokacije u heap memoriji u kojoj je sam sadržaj Stringa upisan. U skladu sa tim, potpuniji izgled memoriji za prethodni primer je ilustrovan na sledećoj slici.

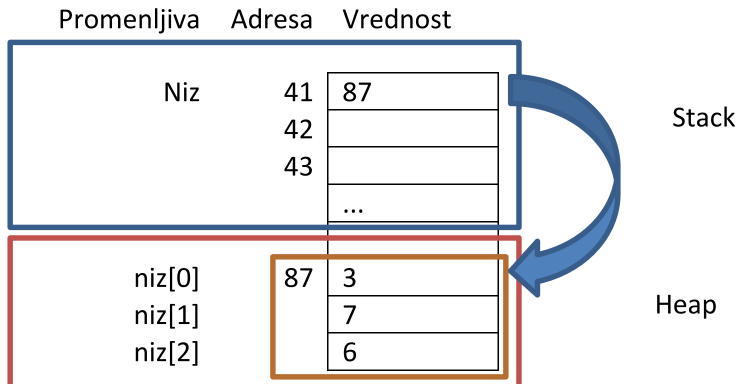


Pošto je String predstavljen kao niz karaktera, svaki karakter se skladišti u posebnim lokacijama. U narednom poglavlju detaljnije je objašnjeno kako se nizovi skladište u memoriji.

2.5. Reprezentacija niza u memoriji

Slično objektima, i elementi niza se skladište u heap memoriji. Na stack memoriji se skladišti samo adresa lokacije u heap memoriji na kojoj je niz upisan. Dat je primer inicijalizacije niza i ilustracija sadržaja memorije.

```
int niz = {3, 7, 6};
```

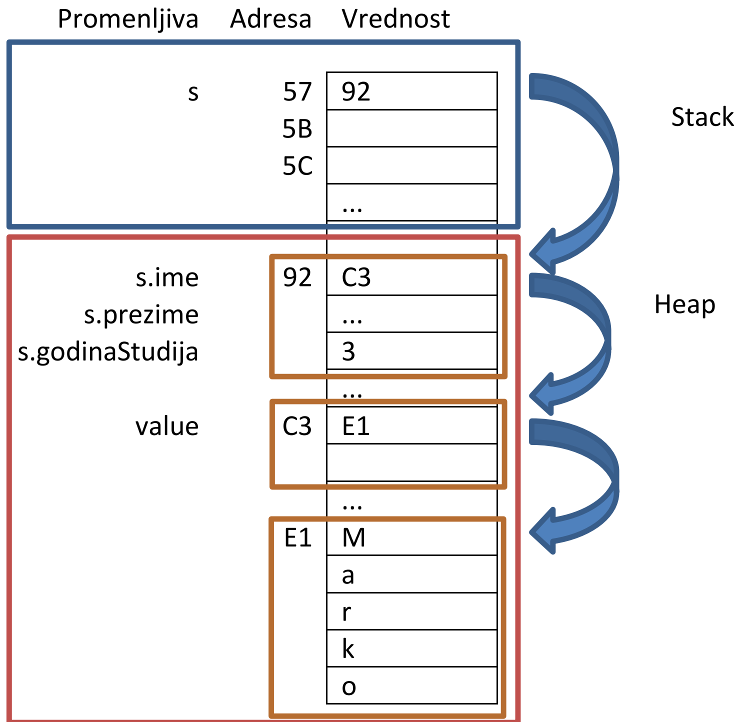


Dakle, promenljiva koja predstavlja niz ustvari skladišti adresu na kojoj su podaci iz niza. Operator [] koji se koristi za indeksiranje niza ustvari pristupa adresi pomerenoj za određeni broj elemenata u odnosu na nulti element niza. Tako izraz niz[2] znači pristup elementu koji je za dva elementa pomeren u odnosu na adresu prvog elementa niza. U slučaju niza int promenljivih, to znači pomeraj od 8 lokacija jer svaki element zauzima 4 lokacije od jednog bajta (2 * 4 = 8).

Pored samih elemenata niza, za niz se skladišti i podatak o dužini niza (ukupnom broju elemenata zauzetih za ovaj niz pri inicijalizaciji). Ovaj podatak je dostupan pod nazivom length na sledeći način:

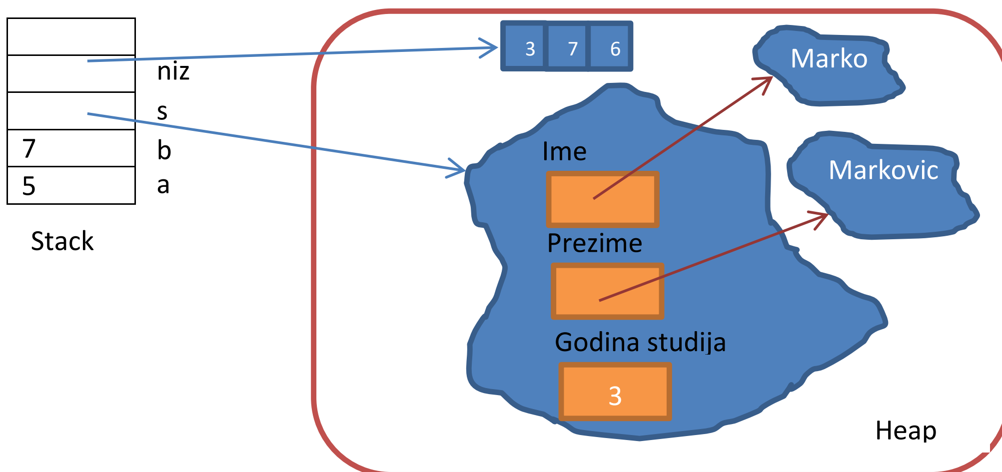
```
int duzina = niz.length;
```

Sada možemo još potpunije prikazati sadržaj memorije za skladištenje imena studenta. Klasa String skladišti karaktere koji čine sadržaj stringa u atributu *value* koji je niz karaktera (char[]). To znači da se u String objektu koji predstavlja ime studenta skladišti adresa iz *heap* memorije na kojoj je uskladišten taj niz karaktera. Ovo je ilustrovano na slici.



2.6. Pojednostavljena ilustracija zauzimanja memorije

Kao što smo rekli, memorija je jedan niz bajtova koji Java virtuelna mašina posmatra kao više sekcija kojima upravlja na različit način. Zbog jednostavnijeg praćenja zauzeća memorije, često se ove dve sekcije predstavljaju kao posebni delovi memorije koji se i grafički drugačije ilustruju. Data je ilustracija kreiranja primitivnih promenljivih, objekata i nizova iz prethodnih primera u ovom obliku.



Kao što vidimo, heap se često predstavlja kao dvodimenzionalni prostor koji skladišti objekte. U stack memoriji se ilustruju reference na delove heap memorije. Manje važni detalji se ignorišu (npr. činjenica da String skladišti svoje karaktere u posebnom nizu).

2.7. Zauzimanje memorije za elemente niza

Kada smo se upoznali sa ključnom rečju new i načinom skladištenja nizova u memoriji, možemo da pogledamo i druge sintaksne načine kreiranja niza u Javi. U prethodnim primerima smo vršili inicijalizaciju niza vrednostima pri kreiranju niza. Ovo nije obavezno, obzirom da je moguće i samo deklarisati niz, kao i samo zauzeti prostor u memoriji za smeštanje elemenata niza. Dat je primer kreiranja niza od 5 celobrojnih elemenata.

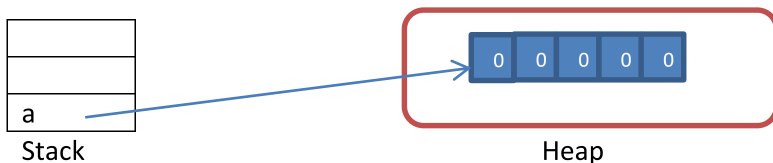
```
int[] a;
a = new int[5];
```

Deklaracija niza

Zauzimanje memorije za elemente niza

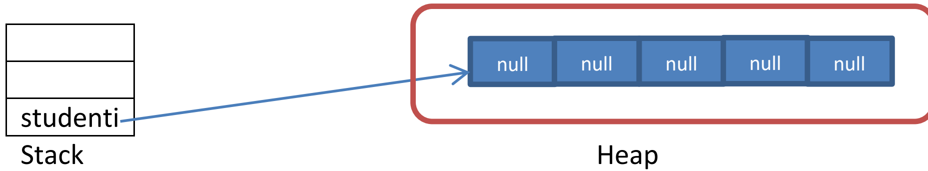
Dakle, kao i kod objekata i ovde se koristi ključna reč *new*, koja inicira zauzimanje memorije i kao rezultat vraća adresu memorijske lokacije iz *heap* memorije na kojoj je sadržaj kreiran. Kada je memorija zauzeta, elementi će biti popunjeni *default* vrednošću za taj tip podatka. Kod celih brojeva, *default* vrednost je nula.

Na slici je ilustrovan izgled memorije za prethodni primer kreiranja niza.



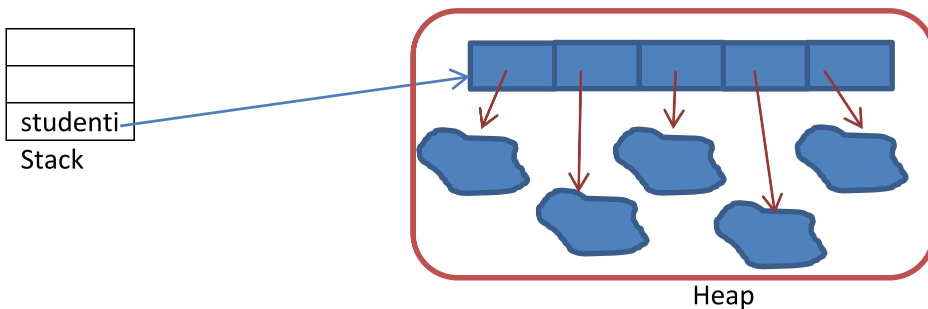
Ne postoji prepreka da se napravi niz elemenata složenog tipa. Npr. moguće je napraviti niz objekata klase String ili niz objekata klase Student. U slučaju niza objekata, niz se inicijalno popunjava *default* vrednošću za adresne tipove, a to je null. Dat je primer kreiranja niza studenata sa *default* vrednostima i ilustrovan je izgled memorije.

```
Student studenti[] = new Student[5];
```



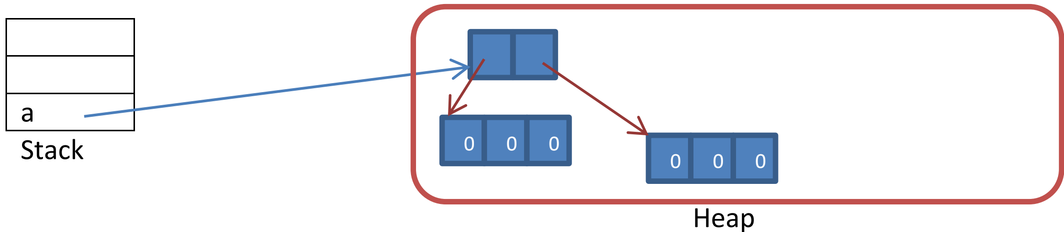
Sada je moguće instancirati svaki element niza. U narednim primeru prikazan je kod za popunjavanje elemenata ovog niza novim objektima klase Student i ilustrovan je izgled memorije.

```
for (int i = 0; i < 5; i++) {
      studenti[i] = new Student();
}
```



Na sličan način je moguće izvršiti deklaraciju i inicijalizaciju višedimenzionalnog niza. Dat je primer kreiranja višedimenzionalnog niza i ilustrovan je izgled memorije.

```
int  a[][] = new int[2][3];
```



Kao što je objašnjeno, višedimenzionalni niz je niz čiji su elementi nizovi. Vidimo da elementi niza skladište adrese ovih nizova.

2.8. Životni ciklus objekta

Kada promenljivoj istekne doseg (eng. *scope*) (kada se završi blok koda u kojem je deklarisana), promenljiva se uklanja sa steka. Kada je reč o objektima, osim reference koja se skladišti u *stack* memoriji, postoji i sam objekat uskladišten u *heap* memoriji. Ovi objekti nastavljaju da zauzimaju heap memoriju i nakon što se njihova referenca uklonila sa stack memorije, tj. nakon što se objekat više ne koristi. Java virtuelna mašina ima podršku za upravljanje memorijom tako da oslobodi programera obaveze da sam oslobađa heap memoriju, a da istovremeno ne postoji opasnost da nekorišćeni objekti prepune heap memoriju.

Ova podrška realizovana je putem *Garbage Collection* mehanizma. Kada se smanji količina raspoložive heap memorije, Java virtuelna mašina automatski aktivira *Garbage Collector* kao poseban program koji uklanja iz *heap* memorije sve objekte koje ne referencira nijedna promenljiva uskladištena u *stack* memoriji.

2.9. Pronalaženje klasa pri izvršavanju programa

U prethodnim primerima videli smo da klasa u svom kodu može da referencira druge klase. Tako je glavni program koristio klasu Student, a klasa Student klasu Grad. Postavlja se pitanje kako java virtuelna mašina pronalazi klase koje klasa referencira. Podrazumevano, java virtuelna mašina traži druge klase u istom paketu u kojem je klasa iz koje se referenciranje vrši.

Ako je referencirana klasa u drugom paketu, potrebno je naznačiti ime paketa u kojem se referencirana klasa nalazi. Ovo možemo uraditi na dva načina. Prvi način je da svaki put pri navođenju imena klase naznačimo i paket u kojem se nalazi. Dat je primer ovakvog korišćenja.

```
vp.administracija.Grad g = new vp.administracija.Grad();
```

Pošto ovakav pristup povećava količinu koda i smanjuje mu čitljivost, češće se primenjuje drugi način. Ovaj način podrazumeva navođenje imena klase u import sekciji fajla. Import sekcija dolazi pre deklaracije klase, a nakon deklaracije paketa. U import sekciji navodimo puna imena klasa koje referenciramo u kodu. Tako bismo za prethodni primer mogli koristiti sledeći import izraz.

```
import vp.administracija.Grad;
```

Sada više nije neophodno navoditi ime paketa pri referenciranju klase, jer je to već navedeno u import sekciji.

Ukoliko želimo da referenciramo sve klase iz nekog paketa, moguće je u import sekciji koristiti džoker znak *. Dat je primer korišćenja ovog džoker znaka.

```
import vp.administracija.*;
```

Sada se postavlja pitanje kako Java virtuelna mašina da zna gde na disku da traži referencirane pakete. Java virtuelna mašina pakete traži u folderima navedenim u CLASSPATH promenljivoj okruženja (eng. *environment variable*). Eclipse IDE pri pokretanju programa, automatski prosleđuje pakete iz trenutnog projekta kao CLASSPATH parametar, tako da java virtuelna mašina može da pronađe referencirane klase.

Interesantno je da smo u dosadašnjim primerima koristili klasu String koja se nalazi u paketu java.lang iako ovu klasu nismo navodili u import sekciji. Razlog je taj što java.lang paket sadrži osnovne klase koje se veoma često koriste, pa JVM ovaj paket tretira na specijalan način tako što se ne zahteva navođenje ovog paketa u import sekciji.

2.10. Java biblioteka klasa

Java distribucija sadrži već implementiran značajan broj klasa koje realizuju često korišćene funkcionalnosti u programima. Ovaj skup klasa se naziva Java biblioteka klasa (eng. *Java Class Library*). Jedna klasa iz ove biblioteke sa kojom smo se već susretali je klasa String. U nastavku su navedene neke od grupa klasa iz Java biblioteke klasa.

- fundamentalne klase za razvoj Java programa (npr. klasa String) - `java.lang` paket
- pristup ulazno izlaznim uređajima (fajl sistem, mrežna komunikacija, ...) - `java.io`, `java.nio` i `java.net` paket
- klase koje predstavljaju različite strukture podataka (npr. liste)
- rad sa bazama podataka – `java.sql` paket

Kompletan spisak i dokumentacija klasa iz Java biblioteke klasa se mogu naći na internet adresi https://docs.oracle.com/javase/7/docs/api/.