PREDMET

Osnove Java Programiranja

Čas 13-14

Principi OO, Enkapsulacija

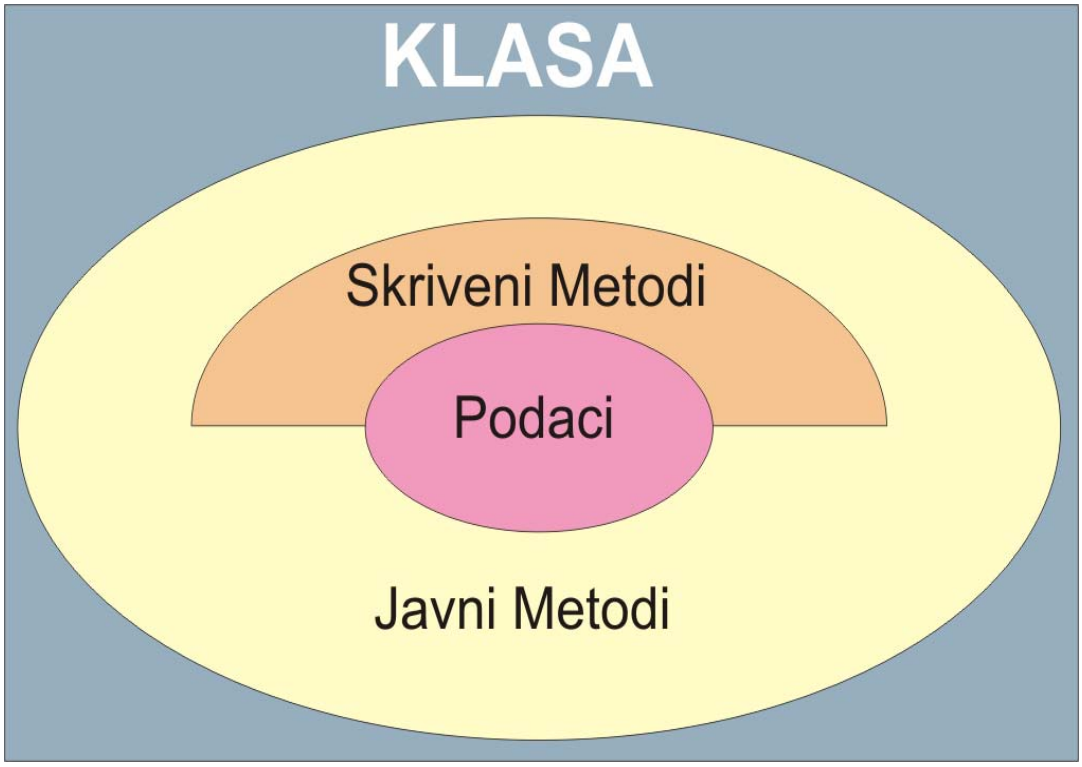
Oktobar 2011.

SADRŽAJ

Čas 13-14

Principi OO, Enkapsulacija

Enkapsulacija



Enkapsulacija ili učaurenje ili sakrivanje. Enkapsulacija je jedan od tri osnovna principa objektno orijentisanog programiranja. Enkapsulacija nam govori da podaci treba da se sakriju od spoljnih pristupa. Kada enkapsuliramo podatke i određene metode obezbeđujemo da naš objekat ima strogo kontrolisani pristup. Primena enkapsulacije je izuzetno bitna, pogotovo kod velikih projekata. Poštovanjem pravila enkapsulacije obezbeđujemo da objekti imaju strogo kontrolisane ulaze i izlaze, a samim tim smanjujemo mogućnost greške, logičke nedoslednosti ili grešaka u programu. Princip enkapsulacije nam govori da podaci moraju da budu privatni, odnosno da moraju da pripadaju samo toj klasi i da niko van klase nema prava da pristupi podacima. Ako obezbedimo da podacima ne može da se pristupi spolja, onda smo obezbedili da niko ne može neplanirano da promeni podatak. Svakako da neko spolja ima potrebu da pristupi tim podacima; pristup podacima se obezbeđuje javnim metodama. Javne metode obezbeđuju pristup podacima na strogo kontrolisan način. Ako imamo nekakvo izračunavanje u okviru same klase koje ne bi treblo niko spolja da pokreće, takve metode takođe proglašavamo privatnim i time obezbeđujemo da niko spolja ne sme da ih pokrene. Ovi principi značajno povećavaju stabilnost aplikacije i obezbeđuju da nam jedna klasa bude celina za sebe. Princip jedna klasa-program za sebe je izuzetno bitan kod objektno orijentisanog programiranja. Principi enkapsulacije obezbeđuju upravo tu osobinu da objekat bude potpuno celina za sebe, celina koja ume da se stara o svojim podacima i vodi racuna o svojim podacima.

Primer ne enkapsulirane klase

Primer ne ENKAPSULIRANE klase

```
public class LagerProst
{
	int kol;
	double cena;
	double vrednost;
	void izracunaj()
	{
		vrednost = cena * kol;
	}
}
```

Nema nikakvih modifikatora vidljivosti.

Ovakva klasa je neenkapsulirana, odnosno podacima kol, cena i vrednost može da se pristupi spolja, iz neke druge klase. Takođe metodi izračunaj može da se pristupi i spolja.

Upotreba ne enkapsulirane klase

```
LagerProst lp = new LagerProst();
lp.kol = 100;
lp.cena = 50.5;
lp.izracunaj();
double rezultat = lp.vrednost;
System.out.println("Rezultat = " + rezultat);
```

U prošlom primeru smo napravili klasu koja se zove LagerProst. Sada ćemo da napravimo instancu te klase i nazvacemo je lp.

```
LagerProst lp = new LagerProst();
lp.kol = 100;
lp.cena = 50.5;
lp.izracunaj();
double rezultat = lp.vrednost;
System.out.println("Rezultat = " + rezultat);
```

Primetićete da u potpunosti funkcioniše. Klasa radi svoj posao i ne pravi nikakve probleme.

Moguća greška pri upotrebi ne enkapsulirane klase

Lako je napraviti grešku koristeći neenkapsulirane klase. Na ovom slajdu ćemo da vidimo jednu od tipičnih grešaka prilikom neenkapsuliranih klasa:

```
LagerProst lp = new LagerProst();
lp.kol = 100;
lp.cena = 50.5;
lp.izracunaj();
double rezultat = lp.vrednost;
System.out.println("Rezultat = " + rezultat);
```

```
lp.cena = 60.5;
rezultat = lp.vrednost;
System.out.println("Rezultat = " + rezultat);
```

Kao što se na primeru vidi izmenili smo cenu ali smo zaboravili da ponovo izračunamo vrednost. Nakon toga smo pokušali da prikažemo novu vrednost ali, s obzirom da smo zaboravili da ponovo izračunamo vrednost dobićemo staru vrednost umesto nove.

Ovaj problem se lako rešava upotrebom enkapsulacije.

Pravilno enkapsulirana klasa

U primeru koji sledi je prikazana klasa koje je pravilno enkapsulirana.

Svi podaci klase su enkapsulirani što znači da imaju modifikator vidljivosti private. Pristup promenljivima klase je obezbeđen preko odgovarajućih metoda. Imamo metodu koja vraća vrednost enkapsulirane promeljive. Ova metoda se zove geter jer vraća vrednost promenljive. Druga metoda se zove seter i ona obezbeđuje da se neka promenljiva setuje na novu vrednost. Naziv ove metode je seter jer ona setuje vrednost promenljivoj.

Kao što se vidi na primeru seteri i geteri imaju strogo definisanu strukturu. Metode geteri moraju vraćaju isti tip podatka koji je i promenljiva čiji su oni geter. Metode koje setuju podatke, seteri, moraju da primaju podatak istog tipa kao što je i podatak koji setuju. Seteri ništa ne vraćaju.

```
public class LagerPravilno
{
	private int kol;
	private double cena;
	private double vrednost;
	public int getKol() { return kol; }
	public double getCena() { return cena; }
	public double getVrednost() { return vrednost; }
	public void setKol(int kol)
	{
		this.kol = kol;
		izracunaj();
	}
	public void setCena(double cena)
	{
		this.cena = cena;
		izracunaj();
	}
	// public void setVrednost(double vrednost)
	// {
	// this.vrednost = vrednost;
	// }
	private void izracunaj()
	{
		vrednost = cena * kol;
```

```
    }
}
```

Seteri za cenu i količinu imaju u sebi i poziv metode izračunaj. Iako je uobičajeno da seteri i geteri imaju jednostavnu formu prikazanu na predhodnom slajdu, ponekad je korisno da ih dopunimo po potrebi. Osnovna forma setera i getera se lako implementira u razvojnom okruženju pozivom odgovarajućeg refaktoringa. U NetBeans razvojnom okruženju to je refaktoring Encapsulate Field.

U ovom primeru smo izvršili izmenu standardnih setera da bi obezbedili ažurnost podataka. Dakle, nakon što se izmeni količina ili cena instanci automatski će se pozvati i metoda izračunaj. Korisnik ove klase neće morati da poziva explicitno metodu izračunaj jer će ona biti automatski, na ovaj način, pozvana svaki put kada se promeni neki od podataka koji utiču na promenljivu vrednost.

Kao što vidimo na slajdu ova klasa je strogo kontrolisana. Metoda izračunaj je deklarisana pomoću modifikatora vidljivosti private tako da korisnik ove klase ne može eksplicitno da poziva metodu izračunaj. Izračunavanje promenljive vrednost se vrši automatski, vrednost se izračunava kad god se promeni cena ili količina. Ovim smo dobili visoko enkapsuliranu klasu koja je u potpunosti zaštićena od nepravilnog korišćenja.

Na slajdu je komentarisana metoda setVrednost. Promenljiva vrednost je zavisna promenljiva i ona se uvek izračunava u zavisnosti od trenutne vrednosti cene i količine. Da bi klasa bila dobro enkapsulirana i zaštićena od eventualnog pogrešne primene ove klase, podatak vrednost ima samo geter. Seter ne postoji za ovu promenljivu. Na ovo moram da obratimo pažnju kada pravimo klase. Ako u klasi imamo neke zavisne podatke treba da onemogućimo direktan pristup tim podacima.

Upotreba pravilno enkapsulirane klase

```
LagerPravilno lager = new LagerPravilno();
lager.setKol(100);
lager.setCena(50.5);
double rezultat = lager.getVrednost();
System.out.println("Rezultat = " + rezultat);
lager.setCena(60.5);
rezultat = lager.getVrednost();
System.out.println("rezultat = " + rezultat);
```

Ovde vidimo prednost pravilno enkapsulirane klase. Pravilno enkapsulirana klasa nikada ne može da da logički neispravan rezultat (kao u prikazanom primeru ne enkapsulirane klase, gde smo uspeli da izazovemo nepravilan rezultat). U ovom slučaju, pravilno enkapsulirane klase, ne može da nam se desi da dobijemo nepravilan rezultat.

Modifikatori vidljivosti

Na prethodnim primerima smo koristili ključne reči private i public da bi odredili da li nekoj promenljivoj metodi sme da se pristupa van klase ili ne sme. Mi smo tom prilikom koristili modifikatore vidljivosti. Ključne reči private i public nam određuju način upotrebe promenljive ili metode. Pored ova dva modifikatora vidljivosti postoji još jedan – protected. U slučaju ne enkapsulirane klase nismo stavili ni jedan modifikator vidljivosti, ali to ne znači da on ne postoji. To je takozvani default modifikator vidljivosti. Kada se ne napiše ni jedan modifikator vidljivosti, on će da ima default ponašanje.

Modifikatori vidljivosti su prikazani od strožih ka manje strogim:

private

protected

default

- public

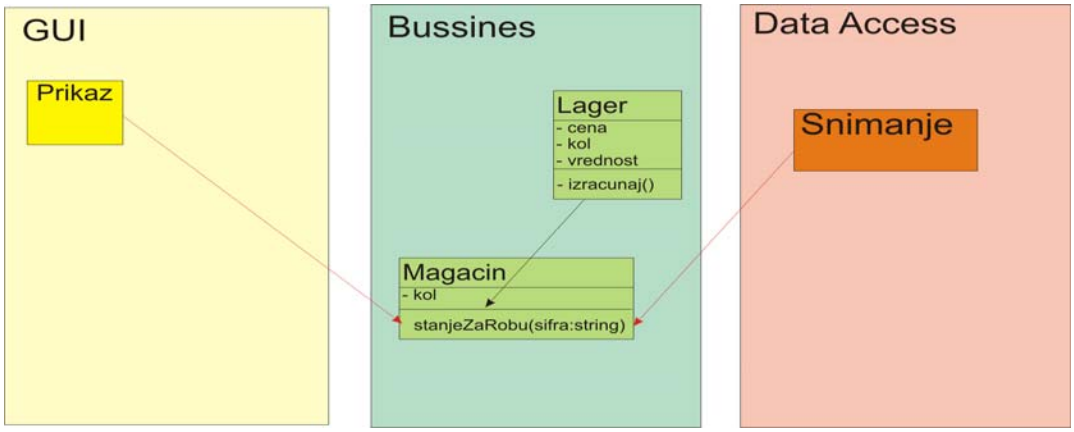
Ako bi probali u enkapsuliranoj klasi da direktno pristupimo promenljivoj dobili bi poruku o grešci jer modifikator vidljivosti private onemogućava direktan pristup toj promenljivoj.

```
lager.cena = 60.5; //vraća grešku ne može da se direktno pristupa
```

Upotreba Default modifikatora vidljivosti

Da bi objasnili funkcionisanje default modifikatora vidljivosti prikazaćemo vizuelno jednu slozenu aplikaciju. Svaka složena aplikacija bi trebalo da se sastoji iz nekoliko slojeva. Svaki sloj aplikacije predstavlja jednu grupu klasa koje se bave određenim delom funkcionalnosti te aplikacije. Najčešća podela na slojeve u klasičnim aplikacijama je na tri dela: GUI, Bussines i Data Access, kao što je prikazano na slici.

Uobičajeno je da se svaki od ovih delova (slojeva) smesti u poseban paket:



GUI – rad sa korisnikom (grafički deo)

Bussines – logika rada i manipulacija nad podacima

Data Access – snimanje i učitavanje, provera integriteta (čuvanje podataka)

Kod velikih aplikacija svaki od ovih paketa ima svoje podpakete koji se bave određenim delom u okviru samog paketa.

Kada je neka promenljiva ili metoda deklarisana bez upotrebe modifikatora vidljivosti to znači da ona ima default modifikator vidljivosti. Ovaj modifikator definiše da promenljivoj ili metodi može direktno da se pristupi iz drugih klasa u okviru istog paketa ali ne može da joj se pristupa direktno iz drugih paketa.

U primeru datom na slajdu klasa Magacin ima metodu stanjeZaRobu koja ima default modifikator vidljivosti. Ovoj metodi pristupamo iz klase Lager bez problema jer se nalaze u okviru istog paketa bussines. Međutim klase Prikaz i Snimanje nemogu da pristupe ovoj metodi jer ne pripadaju istom paketu.

Upotreba protected modifikatora vidljivosti

Protected modifikator vidljivost – nad klasa

Protected modifikator vidljivosti dolazi do izražaja kod hijerarhije klasa. Kada imamo neku nadklasu i kada imamo klasu koja je nasledila tu nadklasu. Primer upotrebe protected modifikatora vidljivosti je dat na slajdu kroz jednu nad klasu i jednu klasu koja nju nasleđuje. U datom primeru nad klasa se zove NadKlasa a klasa koja je nasleđuje se zove Roba.

U okviru klase NadKlasa postoji dva podatka koji imaju protected modifikator vidljivosti. To su podaci sifra i naziv. Nad klasa ima konstruktor koji prima ova dva podatka.

```
public class NadKlasa
{
protected String sifra;
```

```
protected String naziv;
public NadKlasa(String sifra, String naziv)
{
	this.sifra = sifra;
	this.naziv = naziv;
}
public NadKlasa() { }
public String getSifra() { return sifra; }
public void setSifra(String sifra) { this.sifra = sifra; }
public String getNaziv() { return naziv; }
public void setNaziv(String naziv) { this.naziv = naziv; }
@Override public String toString() { return sifra + " - " + naziv; }
}
```

Nad klasa ima, naravno i prazan konstruktor, kao i odgovarajuće setere i getere za protected podatke sifra i naziv. Kao što smo već učili svaka klasa bi trebalo da ima i Override metode toString pa je to napravljeno i u ovoj klasi. Ovim smo završili pripremu nad klase koja sadrži podatke sa protected modifikatorom vidljivosti.

Protected modifikator vidljivost – Konkretna klasa

```
public class Roba extends NadKlasa
{
	private double cena;
	public Roba(double cena) { this.cena = cena; }
	public Roba() { }
	public Roba(String sifra, String naziv, double cena)
	{
		super(sifra, naziv);
		this.cena = cena;
	}
	public double getCena() { return cena; }
	public void setCena(double cena) { this.cena = cena; }
	@ Override public String toString()
	{
		return "Roba: " + naziv + " ima cenu: " + cena;
	}
}
```

Kreiramo konkretnu klasu Roba koja će da koristi podatke iz klase NadKlasa. Modifikator protected dozvoljava direktan pristup samo klasama koje su nasledile klasu u kojoj se podaci nalaze. Iz tog razloga klasa Roba je nasledila klasu NadKlasa. Nasleđivanje se vrši pomoću ključne reči extends i o tome će biti više reči na jednom od sledećih predavanja. Klasa Roba treba pored sifre robe i naziva robe da ima još i podatak cena robe. U klasi Roba deklarisaće se samo promenljiva cena jer promenljive sifra i naziv već postoje u nad klasi i koristićemo ih odande. Promenljivoj cena je dodeljen modifikator vidljivosti private kako bi klasa bila pravilno enkapsulirana.

Po pravilima koje smo već učili klasa treba da ima tri konstruktora, prazan konstruktor, konstruktor sa podacima klase i kopi konstruktor. Sva tri konstruktora kao i definicije podatka cena su prikazani na slajdu.

U kopi konstruktoru se vidi poziv podacima sifra i naziv. Kao što se vidi u primeru ovim podacima pristupamo kao da su podaci same klase u kojoj se nalazimo. Ovo je moguće zato što su ovi podaci u nad klasi definisani pomoću modifikatora vidljivosti protected.

U metodi toString se takođe vidi poziv protected podatka naziv iz nad klase.

Upotreba klasa Roba i NadKlasa

```
Roba r = new Roba("0001", "Sveska", 56.6);
System.out.println("Roba = " + r);
r.setNaziv("Sveska sa tvrdim koricama");
System.out.println("Roba = " + r);
```

Roba = Roba: Sveska ima cenu: 56.6

Roba = Roba: Sveska sa tvrdim koricama ima cenu: 56.6

Klasa Roba je dobro enkapsulirana i korisnik ove klase koristi podatke sifra i naziv kao da su definisani u klasi Roba.

Upotrebljavamo klasu Roba koristeći podatke i iz klase NadKlasa i iz klase Roba. Kao što se vidi u ovom primeru modifikatro vidljivosti protected nam obezbeđuje raspodelu podataka kroz hierarhiju bez narušavanja encapsulacije ka korisnicima ovih klasa.

Domeni važenja promenljivih

Svaka promenljiva ima svoj životni vek. Pod životnim vekom promenljive podrazumevamo period od trenutka nastajanja te promenljive do trenutka kada ona prestaje da se koristi. Razlikujemo nekoliko domena važenja promenljivih. Promenljiva može da važi u okviru petlje, u okviru metoda, u okviru instance klase ili u okviru svih instanci jedne klase.

- u okviru petlje
- u okviru metoda
- u okviru instance klase
- u okviru klase

Primer važenja promenljive u okviru petlje

Ovde je prikazana jedna tipična **for** petlja. U okviru ove petlje prolazimo kroz niz, i punimo ga sa vrednošću proizvoda indeksa tog niza. U okviru petlje postoji promenljiva i. Ova promenljiva i se inicijalizuje u prvom delu **for** petlje, u delu naredbe **int** i = 0. Tokom postojanja petlje ta promenljiva postoji, i ona se koristi za računanje, moze se uvećavati, menjati. U trenutku kada se napusti **for** petlja, ta promenljiva više ne postoji. Pod komentarom je napisan kod i = 8. Taj deo koda se može izvršiti zato što promenljiva i više ne postoji, prestaje njen domen važenja.

```
for (int i = 0; i < niz.length; i++)
{
      niz[i] = i * i;
}
// i = 8; //nemože jer i više ne postoji
```

Primer važenja promenljive u okviru metode

U primeru je prikazana metoda izračunavanja koja, kao parametar, prima dve promenljive a i b, tipa **int**. U telu metode je deklarisana nova lokalna promenljiva c. Sve tri promenljive (a, b i c) su lokalne, i njihov domen važenja je u okviru ove metode, odnosno dokle god postoji ova metoda, ove promenljive postoje. U trenutku kada napuštanja ove metode, ni jedna od ove tri promenljive neće postojati.

```
public int izracunavanje(int a, int b)
{
	int c = 0;
	a++;
	c = a+b;
	return a * b + c;
}
```

Primer važenja promenljive u okviru instance klase

Na primeru se vidi klasa Programer koja je nastala nasleđivanjem klase RadnoMesto. U okviru klase Programer postoje dve promenljive brojSati i cenaPoSatu. Prva je tipa **int**, a druga je tipa **duble**. Obe ove promenljive imaju domen važenja na nivou instance klasa. Konstruktor klase Programer prima brojSati i cenaPoSatu kao parametre tog konstruktora. U ovom konstruktoru brojSati i cenaPoSatu su lokalne promenljive. Lokalna promenljiva uvek ima veću važnost od promenljive instance, tako da, kada u konstruktoru Programer pomenemo broj sati, misli se na lokalnu brojSati, a ne na promenljivu brojSati koja pripada instanci klasa. Lokalne promenljive uvek preklapaju promenljive instance klase. Da bi moglo da se referencira (poziva) na promenljivu instance klase, mora se koristiti naredba **this**. Naredbom **This**.brojSati – se vrši referenciranje na promenljivu brojSati iz klase, i dodeljuje joj se vrednost lokalne promenljive brojSati. Identičan princip važi i za promenljivu cenaPoSatu. Pomoću naredbe **this**.cenaPoSatu referencira se na promenljivu instance klase cenaPoSatu, i dodaje joj se vrednost lokalne promenljive cenaPoSatu. U konstruktoru klase Programer postoji poziv promenljive instance klase brojSati i promenljive instance klase cenaPoSatu. Takođe im se pristupa pomoću ključne reči **this**, i koriste se u okviru ove metode. U okviru ove metode ove dve promenljive su vidljive i postoje. U nastavku klase su prikazani **seter** i **geter** za brojSati. BrojSati je promenljiva koja važi u okviru instance klase, i ovde su prikazani **seter** i **geter** za tu promenljivu i samim tim se pristupa toj promenljivoj. Slede **seter** i **geter** za cenaPoSatu. Promenljiva cenaPoSatu je promenljiva koja važi na nivou instance klase, i u okviru ovih metoda se pristupa toj promenljivoj i barata sa njom. Potom ide metoda izračunajPlatu. Metoda izračunajPlatu takođe ima pristup promenljivama instance klase cenaPoSatu i brojSati i računa vrednost promenljive Zarada koja pripada nadklasi.

```
public class Programer extends RadnoMesto
{
	private int brojSati;
	private double cenaPoSatu;
	public Programer() { }
	public Programer(int brojSati, double cenaPoSatu)
	{
		this.brojSati = brojSati;
		this.cenaPoSatu = cenaPoSatu;
	}
	public Programer(Programer programer)
	{
		this.brojSati = programer.getBrojSati();
```

```
            this.cenaPoSatu = programer.getCenaPoSatu();
      }
      public int getBrojSati() { return brojSati; }
      public void setBrojSati(int brojSati)
      {
            this.brojSati = brojSati;
            izracunajPlatu();
      }
      public double getCenaPoSatu() { return cenaPoSatu; }
      public void setCenaPoSatu(double cenaPoSatu)
      {
            this.cenaPoSatu = cenaPoSatu; izracunajPlatu();
      }
      protected void izracunajPlatu() { zarada = cenaPoSatu * brojSati; }
}
```

Modifikator static

Modifikator **static** obezbeđuje promenu domena važenja metode i podatka na koje se odnosi.

Može da se primeni na metode i podatke:

- upotreba static modifikatora na polja klase
- upotreba static modifikatora na metode klase

Svaka nabrojana upotreba **static** modifikatora obezbeđuje drugačije ponašanje.

Upotreba static modifikatora na polja klase

U primeru je prikazana klasa Neprijatelji. Ona ima nekoliko promenljivih: redniBroj (redni broj neprijatelja), zatim brNeprijatelja (predstavlja promenljivu koja će u sebi sadržati ukupan broj neprijatelja), i promenljive tipa **duble**: štit i energija (koje taj neprijatelj ima). RedniBroj se odnosi na svakog neprijatelja ponaosob, odnosno na svaku instancu Neprijatelja ponaosob, tako da svaka instanca ima svoj redni broj. Promenljive štit i energija takođe se odnose na svaku instancu klase Neprijatelj. Dakle, svaka pojedinačna instanca, svaki objekat, klase Neprijatelj mora da ima svoju jačinu štita i svoju jačinu energije. Promenljiva brojNeprijatelja treba da se odnosi na sve instance klase Neprijatelj, jer ona u sebi treba da sadrži koliko ukupno ima objekata tipa Neprijatelj. Iz tog razloga se stavlja modifikator **static** i menja podrazumevano ponašanje promenljive, tako da ona zadržava svoje stanje na nivou klase, odnosno sve instance klase tipa Neprijatelj imaju pristup jedinstvenoj promenljivoj brNeprijatelja. S' obzirom da promenljiva brNeprijatelja treba da predstavlja ukupnu sumu svih neprijatelja, ona mora da se povećava svaki put kada se novi neprijatelj kreira. Najbolje mesto za povećavanje ove promenljive je u konstruktoru klase Neprijatelji. Postoji više konstruktora klase Neprijatelji, i u svakom od njih promenljiva brNeprijatelja mora biti povećana za 1, jer se ne zna unapred, prilikom kreiranja klase, koji od konstruktora će biti pozvan. Samim tim ista stvar mora da se nalazi u svim konstruktorima. Prikazan je konstruktor koji povećava samo promenljivu brNeprijatelja, i postiji konstruktor koji prima štit i energiju kao parametre i, takođe, povećava statičku promenljivu brNeprijatelja. Zatim postoji **copy** konstruktor, i to je treći po redu konstruktor u okviru ove klase koji takođe mora da poveća statičku promenljivu brNeprijatelja (u slučaju da se novi neprijatelj pravi kao kopija postojećeg). Tada postoje dve **static** metode. To su **geter** i **seter** za **static** promenljivu brNeprijatelja. Ove dve metode su static, što znači da im se može pristupiti i pre nego što postoji bilo koja instanca ove klase. Naime, prilikom prve deklaracije tipa neprijatelj, u memoriji računara se generiše prostor za ove dve metode, kao i za našu promenljivu brNeprijatelja. Promenljivoj brNeprijatelja i **seter-u** i **geter-u** za nju, s' obzirom da svi imaju **static**

modifikator, moguće im je pristupiti odmah, pre nego što se napravi ijedna instanca tog tipa. Iz ovog razloga, mi ne moguće je u okviru **static** metoda da pristup ne-**static** podacima. Kompajler će da prijavi grešku. Obrnuto je moguće. Iz ne-**static** metoda je moguće pristupiti **static** promenljivama. Slede **seter** i **geter** za energiju i za redniBroj. Uvodimo novu metodu umanjiEnergiju. Parametar metode umanjiEnergiju je udarac (tipa **duble**), i u zavisnosti od veličine ove promenljive, biće umanjeni energija i štit samog neprijatelja. Energija se umanjuje za vrednost udarca, dok se štit umanjuje za dvostruku vrednost udarca. U slučaju da je energija ili štit veća od udarca, biće umanjena za jačinu udarca, odnosno za dva puta jačinu udarca,. U suprotnom: u slučaju da je štit manji od jačine udarca biće postavljen na nulu, a u slučaju da je energija manja od jačine udarca onda će biti pozvana metoda umro. U okviru ove metode nema poziva statičke promenljive. Zatim se uvodi metoda umro. Metoda umro poziva se samo u slučaju da neprijatelj izgubi svoju energiju. U tom slučaju neprijatelj umire, i ono što je neophodno da se uradi je da se umanji ukupan broj neprijatelja, tako što se umanjuje statička promenljiva brNeprijatelja. Treba obratiti pažnju da metoda umro nije **static**, odnosno nema modifikator **static**, ali ipak pristupa promenljivoj brNeprijatelja. Metoda umro je lokalna, odnosi se samo na tog jednog neprijatelja, na jednu instancu klase Neprijatelji, i zbog toga nije tipa **static**. Na kraju vidimo **override** metode **toString**, gde definišemo kako će na ekranu biti ispisano za svakog neprijatelja.

```
public class Neprijatelji
{
	int redniBroj = 0;
	static int brNeprijatelja = 0;
	private double stit;
	private double energija;
	public Neprijatelji()
	{
		brNeprijatelja++;
	}
	public Neprijatelji(double stit, double energija)
	{
		this.stit = stit;
		this.energija = energija;
		brNeprijatelja++;
	}
	public Neprijatelji(Neprijatelji neprijatelj)
	{
		stit = neprijatelj.getStit();
		energija = neprijatelj.getEnergija();
		brNeprijatelja++;
	}
	public static int getBrNeprijatelja() { return brNeprijatelja; }
	public static void setBrNeprijatelja(int brNeprijatelja) {
	Neprijatelji.brNeprijatelja = brNeprijatelja; }
	public double getStit() { return stit; }
	public void setStit(double stit) { this.stit = stit; }
	public double getEnergija() { return energija; }
```

```
	public void setEnergija(double energija) { this.energija = energija;
	}
	public int getRedniBroj() { return redniBroj; }
	public void setRedniBroj(int redniBroj) { this.redniBroj = redniBroj;
	}
	public void umanjiEnergiju(double udarac)
	{
		if (energija > udarac)
			energija -= udarac;
		else
			umro();
		if (stit > udarac)
			stit -= udarac * 2;
		else
			stit = 0;
	}
	private void umro()
	{
		brNeprijatelja--;
	}
	@Override public String toString()
	{
		return "Nerpijatelj: " +redniBroj +"/" +brNeprijatelja+
		"\tEnergija: " + energija + "\tStit: " +stit;
	}
}
```

Upotreba klase Neprijatelj

Kreira se niz tipa klasa Neprijatelji, od 10 neprijatelja (djavoli), i kroz jednu **for** petlju će se napuniti. U primeru je prikazana jedna **for** petlja koja se izvrshava. Dakle, postoji lokalna promenljiva brojač i, ona će da prođe onoliko puta kroz petlju koliko ima neprijatelja (u ovom slučaju 10). Puni se svaka instanca klase Neprijatelji slučajnom vrednošću štita i energije. To se radi pomoću matematičke metode mat.random koja vraća slučajni broj u intervalu od 0 do 1. Množenjem tog broja sa 100 dobijamo slučajni interval u rasponu od 0 do 100. Ovo se radi i za vrednost promenljive štit, i za vrednost promenljive energija. Na ovaj način se instanca klase Neprijatelj napunili sa slučajnim vrednostima štita i energije. Svaki napravljen neprijatelj dobija svoj redni broj. Pomoću metode setRedniBroj setuje se njegov broj. Polazi se od i+1, to znači da prvi neprijatelj ima redni broj 1, drugi redni broj 2, itd. Pomoću metode System.out.println ispisuje se na ekranu početno stanje svakog od neprijatelja. Zatim počinje **while** petlja koja se izvršava dok postoji ijedan neprijatelj. Ovo se vidi po tome što getBrNeprijatelja mora da bude veći od 0. Dakle, dok postoje neprijatelji, ova petlja će se izvršavati. Treba obratiti pažnju na specifičnost poziva metode getBrNeprijatelja, koja je, kao što je prikazano ranije, **static** tipa. Svim metodama se pristupa preko instance (dakle, naziv instance.metoda). U slučaju **static** metoda, njima se pristupa: naziv klase.naziv metode. Ovo je specifično za **static** metode, zbog toga što one pripadaju svim instancama klase, a ne samo jednoj pojedinačnoj instanci, i zbog toga je uobičajeno da se one pozivaju tako što se poziva preko naziva klase (naziv klase.naziv metode). Prikazan je deo koda kroz koji prolazimo dok postoji ijedan neprijatelj. Prvo ćemo da slučajno odaberemo nekakav indeks. Ovaj indeks se odnosi na redni broj praktičnog neprijatelja. Dakle, indeks se bira kao slučajna vrednost od 0 do 9. Ovo se postiže pomoću

matematičke metode Math.random čime se dobija slučajan broj od 0 do 1, množi se sa 10 (dobija slučajni broj od 0 do 10), i matematičkom metodom Math.floor zaokružuje na nižu vrednost (dakle, vrednost 9,99 je 9), iseca se ono iza decimalnog zareza pomoću kastovanja u int, i u indeksu se pojavljuje samo broj od 0 do 9. Kao što je ranije pokazano, đavoli su 10 neprijatelja koji imaju indekse od 0 do 9. Ovim dobijamo slučajni indeks. Zatim biramo jačinu udarca, ponovo po slučajnom odabiru. Dakle, Math.random nam vraća broj od 0 do 1, i množi se sa 30, tako da će udarac na neprijatelja biti u intervalu od 0 do 30. Poziva se slučajni neprijatelj, i umanjuje energija za jačinu udarca. Nakon toga ispisuje se na ekranu kolika je bila jačina udarca, koje je novo stanje tog neprijatelja, i koliko je ukupno neprijatelja još ostalo. Ovo se ponavlja dok ima neprijatelja. Udaranjem nekog neprijatelja udarcem koji je jači od energije koju on ima, rezultira time da on umire i umanjuje se broj neprijatelja, i tako se ova petlja vrti u krug sve dok ne dođe u situaciju da neprijatelja više nema. **Main** metoda poziva ovu našu klasu UpotrebaNeprijatelja kako bi započela celu aplikaciju.

```
public class UpotrebaNerpijatelja
{
	public UpotrebaNerpijatelja()
	{
		Neprijatelji[] djavoli = new Neprijatelji[10];
		for(int i = 0; i<djavoli.length; i++)
		{
			djavoli[i] = new Neprijatelji(Math.random()*100,
			Math.random()*100);
			djavoli[i].setRedniBroj(i+1);
			System.out.println(djavoli[i]);
		}
		while(Neprijatelji.getBrNeprijatelja() > 0)
		{
			int index = (int) Math.floor(Math.random()*10);
			double jačinaUdarca = Math.random()*30;
			djavoli[index].umanjiEnergiju(jačinaUdarca);
			System.out.println("JačinaUdarca = " + jačinaUdarca);
			System.out.println(djavoli[index]);
			System.out.println("Neprijatelji.brNeprijatelja = " +
			Neprijatelji.brNeprijatelja);
		}
	}
	public static void main(String[] string)
	{
		new UpotrebaNerpijatelja();
	}
}
```

Upotreba static modifikatora na metode klase

Ranije je prikazana upotreba modifikatora **static** nad metodama **seter** i **geter** za brNeprijatelja, a ovde vam pokazujemo jednu karakterističnu upotrebu **static** modifikatora nad metodom **main**. Naime, metoda **main** je startna metoda, i ona klasa koja sadrži metodu **main** se zove startna klasa. Prilikom započinjanja i završavanja Java aplikacije, spoljna mašina (odnosno operativni sistem preko Java

virtuelne mašine) pokreće aplikaciju tako što poziva ovu metodu. Ova metoda mora da postoji pre nego što postoji instanca klase u kojoj se ona nalazi. Zbog toga ona mora da bude **static**. Ovo je tipičan primer upotrebe **static** metode.

```
public static void main(String[] string)
{
      new UpotrebaNerpijatelja();
}
```

Modifikatori (poznati)

Spisak modifikatora koji su do sada korišćeni. Modifikatori vidljivosti: : **public**, **protected** i **private**. Pomoću modifikatora menjamo osnovno ponašanje promenljivih ili metoda, i menjamo njihovu vidljivost. Modifikator **static**, je obrađivan u ovoj lekciji. U nekim od dosadašnjih primera je korišćen modifikator **abstract**. Modifikator **abstract** nam obezbeđuje da neka metoda ne mora da bude implementirana u nekoj abstraktnoj klasi. Postavljanje modifikatora **abstract** ispred metode, označava da ta metoda ne mora da ima i neće imati telo u toj klasi. Takav način obavezuje da, u nekoj od nasleđenih klasa, postoji implementirana ta metoda. Ako u nekoj klasi postoji bar jedna **abstract** metoda, i sama klasa mora da ima modifikator **abstract**. Modifikator **final** obezbeđuje da neka promenljiva ne može da se menja tokom rada aplikacije. Koristi se za definisanje konstanti u okviru aplikacije, pogledati u zadatku geometrija, gde je u interfejsu dat PI broj kao **final**, odnosno kao konstanta. **Final** se koristi i na metodama i klasama, a u tom slučaju označava da metoda ne može da bude **override**-ovana u nekoj od klasa koje nasleđuju tu klasu, odnosno ako je **final** modifikator nad klasom, to znači da ta klasa ne može da se nasleđuje uopšte. **Abstract** i **final** modifikatori su na neki način suprotni. **Abstract** definiše da neka metoda mora da se implementira u nekoj od klase dece, dok nam modifikator **final** za metodu naređuje da ta metoda ne može da se **override**-uje, odnosno ne može da postoji takva metoda u nekoj od klasa dece. Takodje, ako je reč o modifikatorima nad klasama, modifikator **abstract** definiše da klasa mora da se nasledi, ne može da se koristi takva kakva jeste već mora da se nasledi da bi mogla da se napravi instanca te klase. **Final** modifikator sprečava nasleđivanje, odnosno blokira da ta klasa ne može da se nasleđuje.

- Modifikatori vidljivosti: public, protected, private
- static
- abstract
- final

Modifikatori (ne poznati)

Pored modifikatora koji su do sada predstavljeni postoje i četiri nova modifikatora. Prvi od njih, **synchronized**, se koristi isključivo nad metodama i obezbeđuje zaključavanje metoda od pristupa više niti u isto vreme. Ovo obezbeđuje veću stabilnost u višenitnom radu i jedna je od bitnih osobina Jave, zbog čega je Java pogodna za višenitno programiranje. Modifikator **native** se koristi u specijalnim situacijama, kada je potreban pristup naredbama operativnog sistema ili procesora, odnosno nekim metodama i nekom kodu koji nije Javin. Sprečavanje kompajlera da tumači ono što se nalazi u okviru te metode, postize se modifikatorom **native**, koji komunicira sa kompajlerom omogućavajući da unutra postoji kod koji nije Javin. Modifikator volatile upravlja virtuelnom mašinom tako da promenljiva na koju se odnosi ovaj modifikator može da promeni svoju vrednost u bilo kom trenutku. Uočava se razlika od klasičnih promenljivih koje menjaju svoju vrednost samo u trenutku kada se u programu, nekom naredbom promeni vrednost. U slučaju **volatile** promenljive, promena nastaje nekim spoljnim situacijama. Primer može da bude D/A konvertor gde se spoljni ulaz nekog merača menja usled promene, recimo, temperature, a ne zavisno od toka aplikacije. Modifikator **transient** je veoma bitan sigurnosni mehanizam, odnosi se isključivo na promenljive, i komunicira sa promenljivom koja ima modifikator **transient** tako da ona ne može da napusti virtuelnu mašinu. Bilo kakvo snimanje klase u kojoj se nalazi promenljiva tipa **transient**, ili puštanje kroz mrežu instance klase koja u sebi ima promenljivu **transient**, će prouzrokovati da ta promenljiva nestane iz klase, dakle neće više postojati, odnosno povećava sigurnost te promenljive. Promenljiva može da postoji samo u okviru virtuelne mašine i ne može da je napusti.

- synchronized
- native
- volatile
- transient

Primer - Pogrešan

```
public class Godine {
public Godine() {
	}
		String brojGodina(int godiste, String ime){
		Calendar tekuci = Calendar.getInstance();
		return ime + " ima " + (tekuci.get(Calendar.YEAR) - godiste) +
		" godina";
	}
}
```

Upotreba

```
Godine pera = new Godine();
String s = pera.brojGodina(1980, "Pera");
System.out.println(s);
```

Nedostaci

Ne enkapsulirani podaci

Metoda radi sa parametrima umesto sa atributima klase

Klasa je prazna, odnosno bez zadužnja

Ne fleksibilno rešenje

Podaci se ne čuvaju, nakon poziva metoda nestaju

Ispravno rešenje

```
public class Godine {
	private String ime;
	private int godište;
	public Godine() { }
	public Godine(String ime, int godište) {
		this.ime = ime;
		this.godište = godište;
	}
	public Godine(Godine g) {
		this.ime = g.getIme();
		this.godište = g.getGodište();
	}
```

```
	public int getGodište() { return godište; }
	public void setGodište(int godište) {
		this.godište = godište;
	}
	public String getIme() { return ime; }
	public void setIme(String ime) {this.ime = ime; }
	private int getBrojGodina() {
		Calendar tekuci = Calendar.getInstance();
		return tekuci.get(Calendar.YEAR) - getGodište();
	}
	@Override public String toString(){
		return getIme() + " ima " + getBrojGodina() + " godina.";
	}
}
```

Upotreba

```
Godine pera = new Godine("Pera", 1980);
Godine mika = new Godine(pera);
mika.setIme("Mika");
System.out.println(pera);
System.out.println(mika);
```

Prednosti

Enkapsulirani podaci

Možemo da kreiramo klasu na više načina kao što je pokazano u upotrebi

Ispis teksta može da se prilagodi raznim potrebama

Lakose proširuje klasa sa dodatnim podacima

Podaci se pamte

OO pristup rešavanju problema

Prosleđivanje prostih tipova

```
public void metoda(int a, String b, char c, double k, boolean b)
```

Svi podaci dolaze u metodu kao kopije originalnih podataka

Ovo se odnosi i na String koji je specijalan tip objekta

Izmene nad podacima u metodi ne utiču na originalne podatke

Prosleđivanje objekata

```
public void metod(Student s)
```

Podaci u metodi su reference na originalne podatke

Svaka izmena nad podatkom će izmeniti i originalni objekat

Proslediti objekat kao kopiju

Potrebno je poslati objekat u metodu ali da izmene nad objektom ne utiču na originalni objekat

```
metod(new Student(s));
```

Koristi se kopi konstruktor

Nema izmena nad originalnom metodom, može da se koristi po potrebi.

Proslediti prost tip kao instancu

Potrebno je proslediti prost tip kao instancu kako bi izmene u metodi uticale na originalni objekat

```
public void metod(int[] a){ ... }
```

Nedostaci: mora da se menja originalna metoda, mora da se radi sa nizom.

Varijabilni broj parametara

Postoji potreba da imamo metodu koja prima varijabilni broj parametara.

Može da se implementira pomoću primljenog niza.

Broj elemenata u nizu može da varira

Primer metode koja štampa osobe u grupi

Štampanje niza

```
public void stampa(Student[] s){
	String ispis="";
	for(Student stud : s)
	{
		ispis += stud.getIme() + "\t";
		ispis += stud.getPrezime() + "\t";
		ispis += stud.prosek() + "\n";
	}
	System.out.println("Spisak studenata \n" + ispis);
}
```

Štampanje jednog objekta

```
public void stampa(Student stud) {
	String ispis = "";
	ispis += stud.getIme() + "\t";
	ispis += stud.getPrezime() + "\t";
	ispis += stud.prosek() + "\n";
	System.out.println("Spisak studenata \n" + ispis);
}
```

Upotreba

Da bi poslali podatke u metodu koja prima niz, mora prvo da se pripremi niz pre poziva metode

Metodi koja prima jedan podatak može odmah da se prosledi podatak bez pripreme, ali je neupotrebljiva kad treba poslati više podataka.

Idealno rešenje

Treba nam metoda koja može da primi jedan podatak, ali po potrebi da primi dva ili tri.

Bilo bi poželjno da ista metoda može da primi i niz.

Ovo se u Javi implementira upotrebom varijabilnih parametara koji se prosleđuju metodi

Primer metode sa varijabilnim parametrima

```
public void stampa(Student ... s) {
	String ispis = "";
	for (Student stud : s) {
		ispis += stud.getIme() + "\t";
		ispis += stud.getPrezime() + "\t";
		ispis += stud.prosek() + "\n";
	}
	System.out.println("Spisak studenata \n" + ispis);
}
```

Upotreba

Slanje jednog studenta u metodu

```
stampa(stud1);
```

Slanje dva studenta u metodu

```
stampa(stud1, stud2);
```

Slanje tri studenta u metodu

```
stampa(stud1, stud2, sutd3);
```

Slanje niza studenata u metodu

```
stampa(nizStudenata);
```

Prednosti

Laka upotreba

Čitljivo

Višestuka upotrebljivost iste metode

Jednostavno programiranje