Polimorfizam

Autori:
Goran Savić
Milan Segedinac

1. Interfejsi

Ranije smo objasnili da API predstavlja skup funkcionalnosti koje neka programska komponenta pruža spoljnjem korisniku. Konkretno za klasu, API klase je predstavljen spolja dostupnim metodama. Zaglavlja metoda (naziv, parametre i povratni tip) možemo posmatrati kao specifikaciju API-ja klase. Posmatrajući zaglavlja metoda, saznajemo šta objekat klase može da uradi. Osim specifikacije **šta** objekat može da uradi, klasa definiše i **kako** je specificirano ponašanje realizovano. Ovo je definisano u telu metoda kroz programski kod koji metode sadrže i naziva se implementacija API-ja. Dakle, klasa definiše specifikaciju i implementaciju svog API-ja, tj. formalno opisuje šta objekat može da uradi i na koji način to radi.

U Javi je moguće formalno opisati i samo specifikaciju ponašanja bez implementacije. Dakle, moguće je definisati određeno ponašanje bez navođenja kako je ovo ponašanje realizovano. Realizaciju ovog koncepta ćemo objasniti kroz jedan primer. Već smo se upoznali sa osnovnim strukturama podataka. Jedan tip ovih struktura su sekvencijalne strukture podataka. Videli smo da sekvencijalna struktura podataka može biti realizovana na različite načine. Na primer, putem niza i spregnute liste. Obe pomenute strukture omogućuju skladištenje elemenata tako da je između njih definisan redosled. Nad ovako uskladištenim elementima moguće je izvršiti dodavanje, uklanjanje, izmenu i pronalaženje elementa u odnosu na poziciju elementa u listi. Dakle, specifikacija ove dve strukture je ista, samo je način realizacije, tj. njihova implementacija drugačija.

Pogledajmo kako možemo ovu specifikaciju formalno opisati. Objasnićemo to na primeru realizacije pomenute dve strukture podataka u Java biblioteci klasa. Skladištenje elemenata u niz realizovano je putem klase ArrayList, a u spegnutu listu putem klase LinkedList. Pogledajmo najvažnije metode koje obe klase sadrže.

Zaglavlje metode	Opis
`add(E e)`	Dodaje novi element na kraj liste
`add(int index, E e)`	Dodaje novi element na specificiranu poziciju u listi
`contains(Object o)`	Utvrđuje da li lista sadrži prosleđeni objekat
`get(int index)`	Vraća element na specificiranoj poziciji
`indexOf(Object o)`	Vraća poziciju na kojoj se prosleđeni element nalazi u listi
`remove(int index)`	Uklanja element na specificiranoj poziciji u listi
`remove(Object o)`	Uklanja specificirani objekat iz liste
`set(int index, E e)`	Zamenjuje element na specificiranoj poziciji u listi specificiranim elementom
`size()`	Vraća broj elemenata u listi
`toArray()`	Vraća Java niz koji sadrži elemente iz liste

Obe klase sadrže metode sa prikazanim zaglavljima. Ove metode obavljaju gore pomenute operacije dodavanja, izmene, uklanjanja i pronalaženja elemenata iz liste. Dakle, u ovom delu, specifikacija API-ja ove dve klase je ista. Jasno, realizacija tela metoda se razlikuje, tj. API je na jedan način implementiran u klasi ArrayList, a na drugi u klasi LinkedList.

Prikazani API možemo i formalno definisati u programskom jeziku Java. Za formalnu specifikaciju ponašanja određenog objekta, Java koristi koncept pod nazivom **interfejs**. Terminološki gledano, u računarstvu se pod interfejsom smatraju tačke kroz koje sistem pruža pristup svojim resursima. U

tom smislu, obzirom da API klase definiše metode kao tačke pristupa objektu klase, API klase se često naziva i interfejs klase. Zato se za formalnu specifikaciju API-ja klase koristi koncept interfejsa.

Sintaksno gledano, u programskom jeziku Java, interfejs je spisak zaglavlja metoda i definiše se ključnom rečju *interface*. U nastavku je prikazan interfejs koji opisuje API iz prethodne tabele. Ovaj API specificira ponašanje koje lista treba da ima, bez obzira na to kako je implementirana.

```
public interface List<E>  {
      boolean add(E e);
      void add(int index, E e);
      boolean contains(Object o);
      E get(int index);
      int indexOf(Object o);
      E remove(int index);
      boolean remove(Object o);
      E set(int index, E e);
      int size();
      Object[] toArray();
}
```

Vidimo da i interfejs, kao i klasa, ima naziv (List u ovom slučaju). Takođe, kao što klasa može biti generički realizovana tako da radi sa bilo kojim tipom podatka, tako je i interfejs iz primera generički, obzirom da se ponašanje koje specificira može primeniti na bilo koji tip. Treba primetiti da metode u interfejsu nemaju telo, već samo zaglavlje. Razlog je taj što interfejs samo treba da specificira određeno ponašanje, ali se u pravilu ne bavi implementacijom tog ponašanja. Takođe, nema modifikatora pristupa ispred metoda jer se podrazumeva da su metode javne (*public*).

Kada postoji ovako formalno definisan API u formi interfejsa, klasa može da realizuje metode definisane interfejsom. Ovo se naziva implementacija interfejsa. Implementacijom interfejsa klasa na sebi svojstven način realizuje ponašanje specificirano interfejsom. Na sintaksnom nivou u Javi, u deklaraciji klase se navodi koje interfejse klasa implementira. Na ovaj način, klasa nosi informaciju da objekti te klase mogu da realizuju određeno ponašanje. Prethodno prikazani interfejs (sa izostavljenim delovima koji nam sada nisu važni) je deo *Java Collections Frameworka* Java biblioteke klasa. Klase ArrayList i LinkedList se obavezuju da imaju ponašanje liste time što implementiraju prikazani List interfejs. U nastavku je na primeru deklaracije klase ArrayList prikazana sintaksa kojom klasa navodi da implementira određeni interfejs.

```
public class ArrayList<E> implements List<E> {
...
}
```

Na sličan način i klasa LinkedList implementira interfejs List. Implementacijom interfejsa, klasa se obavezala da realizuje određeno ponašanje. Sintaksno, realizacija ponašanja se vrši u metodama. Klasa je dužna da implementira svaku metodu koju sadrži interfejs kojeg klasa implementira. Metode moraju imati istu deklaraciju (povratni tip, naziv i tipove parametara) kao i metode iz interfejsa. Nema prepreke da klasa uvede i implementira dodatne metode koje interfejs nije specificirao. Dakle, API klase je uvek nadskup API-ja interfejsa koji klasa implementira. Tako da klasa sigurno sadrži sve metode koje sadrži interfejs kojeg implementira. Što se tiče našeg primera, klase ArrayList i

LinkedList sadrže implementirane metode koje specificira interfejs List, jer implementiraju ovaj interfejs.

Naglasimo još da ne postoji prepreka da klasa implementira više interfejsa. Tada je klasa dužna da implementira metode koje specificiraju svi ovi interfejsi. Pomenimo još da od Java verzije 8, interfejs može da definiše podrazumevanu (eng. *default*) implementaciju metode. Tada metoda interfejsa sadrži i telo, pa klasa nije dužna da obezbedi svoju implementaciju ove metode ukoliko podrazumevana implementacija radi na način koji je u skladu sa ponašanjem klase.

Sada ćemo da analiziramo kako možemo da iskoristimo činjenicu da klasa implementira određeni interfejs da bismo dobili programski kod koji je lakše izmenjiv i proširiv. Vratimo se na primer sa List interfejsom i klasama ArrayList i LinkedList koje ga implementiraju. Posmatrajmo programski kod iz sledećeg primera. Kod definiše jednu listu predstavljenu ArrayList objektom u koju smo ubacili tri objekta klase Drzava. Država je definisana oznakom i nazivom. Ovi podaci se mogu proslediti pri inicijalizaciji objekta kao parametri konstruktora, a preuzimaju se get metodama. Za ispis podataka o državama koristi se prikazana metoda ispis().

```
ArrayList<Drzava> drzave = new ArrayList<Drzava>();
drzave.add(new Drzava("sr", "Srbija"));
drzave.add(new Drzava("fr", "Francuska"));
drzave.add(new Drzava("it", "Italija"));

ispis(drzave);

...

void ispis(ArrayList<Drzava> drzave) {
      for (int i = 0; i < drzave.size(); i++) {
            Drzava d = drzave.get(i);
            System.out.println(d.getOznaka() + " " + d.getNaziv());
      }
}
```

Postavlja se pitanje koliko je prikazani kod jednostavan za održavanje. Analizirajmo šta bi trebalo u kodu izmeniti ako bi se u nekom trenutku zaključilo da je bolji izbor podatke o državama skladištiti u spregnutoj listi predstavljenoj klasom LinkedList. Očigledno, na svim mestima u kodu potrebno je umesto klase ArrayList postaviti klasu LinkedList.

Moguće je kod realizovati i tako da bude lakše izmenljiv. Ako klasa implementira određeni interfejs, Java nam pruža mogućnost da kada radimo sa tom klasom, kao oznaku tipa koristimo interfejs koji klasa implementira. Pogledajmo u narednom primeru kod koji ima istu funkcionalnost kao kod u prethodnom primeru, ali omogućuje lakšu izmenljivost.

```
List<Drzava> drzave = new ArrayList<>();
drzave.add(new Drzava("sr", "Srbija"));
drzave.add(new Drzava("fr", "Francuska"));
drzave.add(new Drzava("it", "Italija"));

ispis(drzave);
...

private void ispis(List<Drzava> drzave) {
      for (int i = 0; i < drzave.size(); i++) {
            Drzava d = drzave.get(i);
```

```
			System.out.println(d.getOznaka() + " " + d.getNaziv());
		}
}
```

Primetimo da je sada objekat *drzave* deklarisan da je tipa List, iako je on očigledno tipa ArrayList što se vidi po konstruktoru koji je pozvan. Ovo je dozvoljeno jer klasa ArrayList implementira List interfejs. Svaki ArrayList objekat ima i metode koje specificira interfejs List. Vidimo da je dalji kod generičan i da koristi samo metode koje List interfejs specificira. Dakle, taj kod je ispravan i za objekat bilo koje druge klase koja implementira List interfejs. Takođe, metoda ispis() sada kao tip objekta prima List interfejs, tako da se metodi može proslediti objekat bilo koje klase koja implementira List interfejs. Ovim metoda ispis() ne mora da pretrpi nikakve izmene u slučaju da je umesto klase ArrayList potrebno koristiti neku drugu klasu koja implementira List interfejs. Ako bi prikazani kod trebao da koristi klasu LinkedList za skladištenje država, jedina izmena bi bila u prvom redu gde bi se instancirao objekat klase LinkedList umesto objekta klase ArrayList.

2. Nasleđivanje i polimorfizam

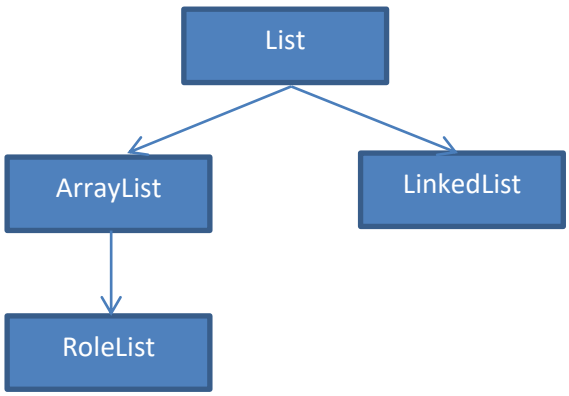
U prethodnom poglavlju videli smo kako objekat određene klase možemo da posmatramo kao da je tipa interfejsa koji klasa implementira. Generalno, ovo važi u svakom slučaju u kojem imamo vezu podtip – nadtip. Veza podtip-nadtip se naziva veza **nasleđivanja** (eng. *inheritance*). Klasa može da nasledi podatke i ponašanje druge klase. U tom smislu, klasu koja se nasleđuje zovemo predak ili roditelj (eng. *parent*), a klasu koja je nasleđuje zovemo potomak (eng. *child*). Klasa naslednica će sadržati sve atribute i metode klase koja je njen predak. Klasa potomak je podtip klase koja joj je predak, a predak se smatra nadtipom klase koja mu je potomak. Kaže se da je klasa podtip specijalizacija klase nadtip, jer predstavlja specifičnu varijantu entiteta predstavljenog klasom koja je nadtip. Na sintaksnom nivou, nasleđivanje se realizuje tako što se u zaglavlju klase navodi ključna reč extends i klasa koja se nasleđuje. Dat je primer nasleđivanja.

```
public class Osoba {
	protected String ime;
	protected String prezime;

	public String getIme() {
		return ime;
	}
	public void setIme(String ime) {
		this.ime = ime;
	}
	public String getPrezime() {
		return prezime;
	}
	public void setPrezime(String prezime) {
		this.prezime = prezime;
	}
}


public class Student extends Osoba {
	private String brIndeksa;

	public String getBrIndeksa() {
		return brIndeksa;
	}
```

```
        public void setBrIndeksa(String indeks) {
              this.indeks = indeks;
        }

        ...
}
```

U prikazanom primeru klasa *Student* nasleđuje klasu Osoba. Klasa *Student* nasleđuje sve podatke i metode definisane u klasi Osoba. Treba primetiti modifikator pristupa *protected* korišćen u klasi *Osoba*. Atributi označeni ovim modifikatorom pristupa su dostupni iz klasa naslednica i klasa koje su istom paketu, ali nisu iz ostalih klasa.

Interfejs je takođe tip podatka i klasa koja implementira interfejs se smatra podtipom tog interfejsa. Moguće je definisati i više nivoa u hijearhiji tipova. Napominjemo da klasa može da implementira više interfejsa, ali ne može da nasledi više od jedne klase. Takođe, validno je i da istovremeno implementira interfejse i nasleđuje određenu klasu. Dat je primer hijerarhije tipova ilustrovan na klasama iz standardne Java biblioteke. Prikazana klasa RoleList predstavlja proširenu realizaciju klase ArrayList i ovde je izabrana samo zbog ilustracije pa se nećemo baviti njenim funkcionalnostima.



Dakle, svaki objekat možemo posmatrati i kao objekat nekog od njegovih nadtipova. Zato kažemo da je objekat polimorfan (može se manifestovati u više tipova, tj. formi), a samu osobinu nazivamo **polimorfizam**.

U primeru sa listom država, parametar metode ispis() je objekat *drzave* koji je označen da je tipa List. List je samo interfejs i ne postoje objekti ovog tipa, pa je jasno da će metoda kao parametar uvek dobijati objekat nekog od podtipova koji implementiraju List interfejs. Pri pozivu metoda size() i get() pozivaće se programski kod implementiran u istoimenim metodama ovog podtipa.

Analizirajmo kako se polimorfizam manifestuje ako je i nadtip klasa, a ne interfejs. U tom slučaju i nadtip sadrži implementirane metode. Iako i nadtip sadrži implementaciju metoda, podtip može da sadrži svoju implementaciju nekih od nasleđenih metoda. Ovo je slučaj kada je ponašanje opisano određenom metodom u podtipu specifičnije u odnosu na ponašanje definisano u nadtipu. Analizirajmo ovaj slučaj na sledećem primeru. Ako posmatramo program za evidenciju prodaje artikala u prodavnici, stavku na računu možemo predstaviti klasom StavkaProdaje čiji je deo prikazan u nastavku.

```
public class StavkaProdaje {
	protected double jedCena;
	protected int kol;

	public double ukupnaCena() {
		return jedCena * kol;
	}
	...
}
```

Vidimo da je stavka predstavljena jediničnom cenom i količinom. Treba primetiti da je modifikator pristupa za ove atribute *protected* kako bi direktno bili dostupni i u klasama naslednicama. Ukupna cena se izračunava kao proizvod jedinične cene i količine. Ukoliko se prodavnica odluči da određene artikle stavi na akcijski popust, za ove stavke na računu ukupna cena se izračunava na specifičniji način. Tada se ovakva stavka prodaje može predstaviti posebnom klasom koja predstavlja specijalizaciju standardne stavke prodaje, tj. akcijska stavka je podtip klasične stavke prodaje. Dat je primer klase AkcijskaStavkaProdaje koja predstavlja stavku prodaje na akciji.

```
public class AkcijskaStavkaProdaje extends StavkaProdaje {

	protected double popust;
	protected int danaNaAkciji;

	public double ukupnaCena() {
		return (jedCena * kol) * (100 - popust - danaNaAkciji)/100;
	}
}
```

Vidimo da klasa AkcijskaStavkaProdaje nasleđuje klasu StavkaProdaje. Time klasa AkcijskaStavkaProdaje sadrži sve atribute i metode kao i klasa StavkaProdaje. Dodatno, vidimo da je akcijska stavka prodaje opisana sa još dva atributa. Atribut *popust* predstavlja inicijalni popust za taj artikal, a atribut *danaNaAkciji* označava broj dana koji su protekli od početka trajanja akcije. Pošto se za akcijsku stavku prodaje ukupna cena drugačije računa, primetimo da klasa AkcijskaStavkaProdaje sadrži novu implementaciju metode ukupnaCena(). Slučaj kada potomak implementira novo ponašanje za metodu nasleđenu od pretka se naziva **redefinisanje metode** (eng. *method overriding)*. Pri redefiniciji metode, metoda mora imati isto zaglavlje kao i metoda pretka (povratni tip, naziv i parametre), samo je telo metode drugačije. Redefinisanje metoda ne treba mešati sa terminom preklapanje metoda (eng. *method overloading*) koji se odnosi na definisanje više metoda istog naziva, a različitih tipova i/ili broja parametara.

Analizirajmo sada kako se polimorfizam može primeniti u slučaju kada imamo dve klase od kojih jedna nasleđuje drugu kao u prikazanom primeru sa stavkama prodaje. Pogledajmo sledeći kod koji radi sa instancama prikazanih klasa.

```
List<StavkaProdaje> stavke = new ArrayList<>();
stavke.add(new StavkaProdaje(76.38, 3));
stavke.add(new StavkaProdaje(129.99, 1));
stavke.add(new AkcijskaStavkaProdaje(158.44, 4, 20, 7));

for (int i = 0; i < stavke.size(); i++) {
	System.out.println("Ukupna cena stavke: " +
		stavke.get(i).ukupnaCena());
}
```

Primetimo da, kao i u primeru sa interfejsima, važi pravilo da se nadtip može deklarisati kao podtip. Tako smo deklarisali da lista prima objekte tipa StavkaProdaje, a u listu su ubačeni objekti ovog tipa, ali i objekat tipa AkcijskaStavkProdaje. Ovo je moguće jer je ovaj tip podtip klase StavkaProdaje. Konstruktori ove dve klase će biti kasnije prikazani i objašnjeni. Ključni deo pomenutog primera je poziv metode ukupnaCena() nad elementom liste u prikazanoj for petlji. Kao što smo rekli, kada je nadtip interfejs nema dileme da će pri pozivu metode biti pozvana metoda iz klase koja implementira interfejs, jer sam interfejs ne sadrži implementaciju metode. Kada su i podtip i nadtip klasa postavlja se pitanje da li će se pozvati metoda iz pretka ili naslednice. U prikazanom primeru, iako je lista deklarisana kao lista objekata StavkaProdaje, neće se uvek pozivati metoda iz te klase. Naime, koja metoda će se pozvati zavisi od toga kojeg je zaista tipa objekat nad kojim se metoda poziva, a ne od toga kako je objekat deklarisan. Konkretno, u prve dve iteracije prikazane for petlje pozvaće se metoda ukupnaCena() klase StavkaProdaje jer su prva dva elementa liste tog tipa. U trećoj iteraciji pozvaće se metoda ukupnaCena() klase AkcijskaStavkaProdaje jer je treći element objekat te klase.

Dakle, skok na instrukciju na kojoj počinje određena metoda nije upisan u kompajliranom programu, jer bi u takvoj realizaciji bila izabrana metoda na osnovu toga kako je objekat deklarisan. Umesto toga, odluka na koju se instrukciju prelazi se donosi u trenutku izvršavanja programa na osnovu tipa objekta na koji referenca na objekat pokazuje. Kažemo da se povezivanje koda ne vrši za vreme kompajliranja (eng. *compile-time*), nego za vreme izvršavanja (eng. *run-time*). Ova osobina dinamičkog izbora koja metoda se poziva se naziva **dinamičko povezivanje** (eng. *dynamic binding, late binding*).

U Javi za svaku redefinisanu metodu važi dinamičko povezivanje. To znači da će se dinamički tek za vreme izvršavanja utvrditi tip objekta nad kojim se metoda poziva, tj. da li se metoda poziva nad tipom kojim je promenljiva deklarisana ili nad nekim od njegovih podtipova. Treba imati u vidu da dinamičko povezivanje može negativno da utiče na performanse, jer se pre poziva metode najpre mora izvršiti kod koji utvrđuje koja konkretno metoda se poziva.

Prema teoriji objektnog programiranja, metode koje mogu biti redefinisane u klasi naslednici kako bi se obezbedilo polimorfno ponašanje, nazivaju se **virtualne metode**. U nekim programskim jezicima (npr. C++ i C#), za metodu je neophodno eksplicitno naznačiti da je virtualna da bismo dobili redefinisano ponašanje nad polimorfnim objektom. U Javi je podrazumevano svaka metoda virtualna i to se posebno ne naglašava u zaglavlju metode.

Kada je reč o polimorfizmu, analizirajmo ovaj koncept na još jednom primeru. Dat je primer koda u kojem se u objekat deklarisan kao predak upisuje objekat tipa naslednik, što je validno ponašanje.

```
StavkaProdaje sp = new AkcijskaStavkaProdaje(158.44, 4, 20, 7);
```

Ako se nad ovim objektom sada pozove metoda ukupnaCena(), kao što je prikazano u nastavku, pozvaće se metoda iz klase AkcijskaStavkaProdaje jer se vrši dinamičko povezivanje.

```
double cena = sp.ukupnaCena();
```

Ipak, treba napomenuti da, obzirom da je promenljiva deklarisana kao objekat klase predak, ona ima pristup samo članovima objekta definisanim u klasi predak. Ovo je ilustrovano u sledećem primeru koda.

```
double cena = sp.getJedCena();
double popust = sp.getPopust();
```

Greška pri kompajliranju

Ukoliko želimo da pristupimo članovima klase naslednice neophodno je da korišćenjem *cast* operatora naglasimo da je objekat tipa klase naslednice. U toku izvršavanja programa se vrši provera da li je prosleđeni objekat zaista naznačenog tipa. Ukoliko jeste, pristupa se objektu, a u suprotnom se pojavljuje izuzetak. Korišćenje *cast* operatora za objekat iz prethodnog primera je prikazano u nastavku.

```
double popust = ((AkcijskaStavkaProdaje) sp).getPopust();
```

3. Ključna reč *super*

Ranije smo pominjali da u Javi postoji ključna reč *this* koja u objektu predstavlja referencu na sam taj objekat. Slično, postoji i ključna reč *super* koja označava referencu na klasu koja je direktni predak. Dat je primer kako se ova ključna reč može koristiti.

```
public class AkcijskaStavkaProdaje extends StavkaProdaje {

	protected double popust;
	protected int danaNaAkciji;

	public double ukupnaCena() {
		return super.ukupnaCena() * (100 - popust - danaNaAkciji)/100;
	}
}
```

Vidimo da je korišćenjem ključne reči *super* pozvana metoda ukupnaCena() iz klase predak, tj. klase StavkaProdaje. Obzirom da ova metoda vraća proizvod jedinične cene i količine, prikazani kod klase AkcijskaStavkaProdaje ima istu funkcionalnost kao i ranije prikazani.

Pored reference na roditeljsku klasu, ključnu reč *super* je moguće iskoristiti i za poziv konstruktora pretka. Prikazan je konstruktor klase StavkaProdaje, a zatim i konstruktor klase AkcijskaStavkaProdaje koji poziva konstruktor pretka.

```
public class StavkaProdaje {
...

	public StavkaProdaje(double jedinicnaCena, int kolicina) {
		this.jedCena = jedinicnaCena;
		this.kol = kolicina;
	}

...

}

public class AkcijskaStavkaProdaje extends StavkaProdaje {

...

	public AkcijskaStavkaProdaje(double jedCena, int kol,
			double popust, int danaNaAkciji) {
		super(jedCena, kol);
		this.popust = popust;
```

```
            this.danaNaAkciji = danaNaAkciji;
        }
...
}
```

Vidimo da se u klasi AkcijskaStavkaProdaje inicijalizacija jedinične cene i količine vrši u konstruktoru pretka i da naslednica poziva ova konstruktor korišćenjem ključne reči *super*. Na sličan način se iz jednog konstruktora klase može pozvati drugi konstruktor iste klase korišćenjem ključne reči *this*. Kao i kod reči *super,* u zagradi se navode vrednosti parametara konstruktora koji se poziva.

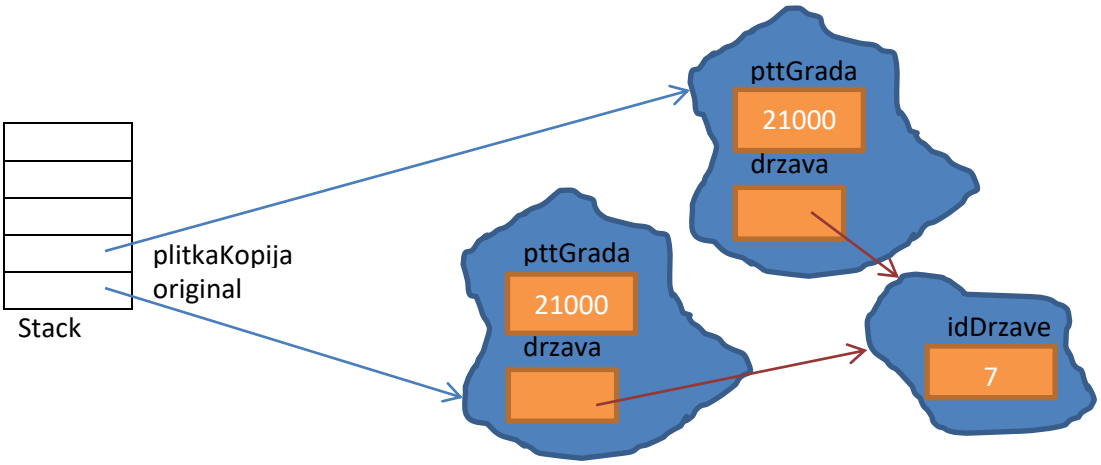
Treba imati u vidu da Java kompajler automatski dodaje naredbu super() u prvi red konstruktora naslednika ukoliko taj konstruktor sam nije pozvao neki konstruktor pretka ili neki drugi svoj konstruktor (korišćenjem naredbe this()). Naredba super() će pozvati konstruktor bez parametara u pretku, što znači da je neophodno da takav konstruktor postoji definisan u pretku ukoliko želimo da klasa naslednica eksplicitno ne poziva konstruktor pretka. Ukoliko klasa naslednica u svom konstruktoru eksplicitno poziva konstruktor pretka, taj poziv mora biti u prvoj liniji konstruktora. Razlog zašto se poziv konstruktora pretka uvek stavlja kao prvi red konstruktora (bilo automatski od strane kompajlera bilo eksplicitno od strane programera) je taj da budemo sigurni da je roditeljska klasa ispravno inicijalizovana pre nego pristupimo nekom od atributa ili metoda definisanih u njoj. Dakle, pri instanciranju objekta klase naslednice uvek se prvo poziva konstruktor klase pretka.

4. Klasa Object

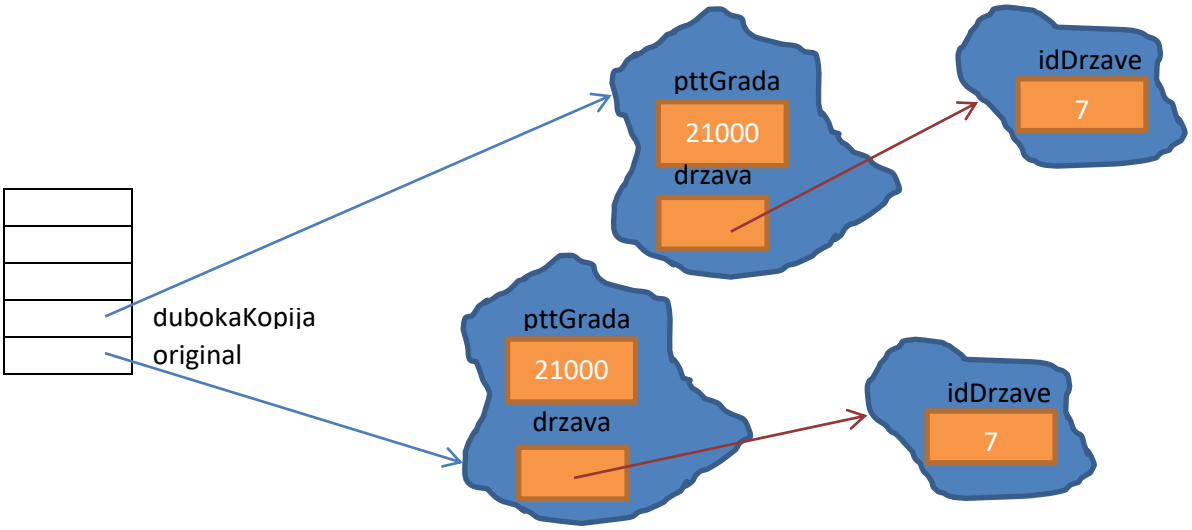
Ranije je pomenuto da Java biblioteka klasa sadrži klasu Object i da su podrazumevano sve klase podtip klase Object. Ovde ćemo detaljnije analizirati ovu klasu. Pogledajmo deo API-ja ove klase.

Naziv metode	Opis
`clone()`	Vraća kopiju objekta. To je novi objekat iste klase kao originalni objekat. U kopiji se svaki atribut inicijalizuje na istu vrednost kao u originalnom objektu. Metoda kreira takozvanu plitku kopiju objekta. To znači da se u kopiji za atribute koji su objekti ne kreira novi objekat, nego se samo upiše referenca iz odgovarajućeg atributa originalnog objekta.
`equals(Object obj)`	Poredi objekat sa prosleđenim objektom. Poređenje je implementirano tako da se smatra da su objekti jednaki samo ako predstavljaju referencu na isti objekat.
`getClass()`	Vraća objekat klase Class, koji predstavlja klasu čija je instanca objekat nad kojim pozivamo metodu. Između ostalog, iz objekta klase Class možemo da dobijemo informacije o nazivu i strukturi te klase.
`hashCode()`	Vraća celi broj koji predstavlja heš kod objekta.
`toString()`	Vraća string koji tekstualno reprezentuje objekat. Ova metoda je implementirana tako da vraća string koji sadrži naziv klase objekta i heš kod objekta u heksadecimalnom formatu.

Pomenuli smo metodu clone() koja kreira plitku kopiju objekta. Na slici je prikazan objekat klase Grad i njegova plitka kopija.



Vidimo da su original i kopija različiti objekti, ali objekat klase Drzava koji je atribut klase Grad nije kopiran, već atribut originala i kopije pokazuju na isti objekat. Naredna slika ilustruje kakav bi bio sadržaj memorije kada bi se kreirala duboka kopija objekta klase Grad.



Primetimo da je sada za atribute kopije napravljen novi objekat. Tako atribut *drzava* u kopiranom objektu pokazuje na novi objekat klase Drzava.

Kada govorimo o plitkim i dubokim kopijama objekta, iskoristićemo priliku da objasnimo još jedan pojam iz teorije objektnog programiranja. Reč je o **konstruktoru kopije** (eng. *copy constructor*). Konstruktor kopije je konstruktor koji kreira objekat klase inicijalizujući ga podacima postojećeg objekta iste klase. Novi objekat će imati iste podatke kao postojeći, pa se stoga ovakav konstruktor naziva konstruktor kopije. Sintaksno, konstruktor kopije je konstruktor klase koji kao parametar dobija objekat iste klase. Dat je primer konstruktora kopije klase Grad.

```
class Grad {
	int pttGrada;
	Drzava drzava;

	public Grad(Grad g) {
		pttGrada = g.pttGrada;
		drzava = g.drzava;
	}
}
```

Prikazani konstruktor kopije kreira plitku kopiju objekta, obzirom da se ne kreira kopija objekta država, već se samo referenca na isti objekat država upisuje u kopiju. Nema prepreke da se konstruktor kopije realizuje da kreira duboku kopiju, ako je to željeno ponašanje.

Obzirom da svaka klasa nasleđuje klasu Object, po potrebi klasa može da redefiniše neke od metoda nasleđenih iz klase Object. Na primer, spomenuli smo da je metoda toString() u klasi Object implementirana tako da vraća tekst sa informacijom o nazivu klase i heš kodu objekta. Obzirom da je namena ove metode da pruži informacije o objektu, dobra je praksa da naslednica redefiniše ovu metodu i da vrati string koji daje više informacija. Najčešće se metoda toString() redefiniše tako da vraća string koji sadrži nazive i vrednosti svih atributa objekta.

Takođe, metoda equals() je u klasi Object implementirana tako da samo uporedi da li su reference dva objekta jednake. Najčešće se i ova metoda redefiniše u naslednici kako bi uporedila objekte prema njihovom sadržaju, a ne adresi na kojoj se nalaze. Recimo, za klasu String je ova metoda redefinisana tako da se dva stringa porede na osnovu vrednosti svakog karaktera.

Što se tiče heš koda objekta koji se dobija pozivom metode hashCode(), dva jednaka objekta treba da imaju jednak heš kod. Iz tog razloga, kada god se redefiniše metoda equals(), pravilo je da se redefiniše i metoda hashCode(). Razlog je taj što metoda equals() određuje kada su dva objekta jednaka, pa u skladu sa tim se mora realizovati i metoda hashCode() kako bi bila ispoštovana logika da jednaki objekti imaju jednak heš kod. U nastavku je prikazana jedna realizacija redefinisanja metoda toString(), equals() i hashCode().

```
class Drzava {
	private int id;
	private String naziv;

	...

	public String toString() {
		return "Drzava [id=" + id + ", naziv=" + naziv + "]";
	}

	public boolean equals(Object o) {
		if (this == o) {
			return true;
		}
		if (o == null || getClass() != o.getClass()) {
			return false;
		}
		Drzava drzava = (Drzava) o;
```

```
            return id == drzava.id;
      }

      public int hashCode() {
            return id;
      }

}
```

Vidimo da je predstavljena klasa Drzava. Svaki objekat ima jedinstveni identifikator predstavljen atributom id. Metoda toString() je redefinisana tako da ispisuje vrednost identifikatora i naziva države. Identifikator je iskorišćen u metodi equals() da utvrdimo da li su dva objekta jednaka. Vidimo da dve države smatramo jednakim ako imaju isti identifikator. U skladu sa metodom equals(), i hashCode() metoda određuje heš kod na osnovu vrednosti identifikatora. Preciznije, sam identifikator je heš kod. Pomenimo i da Eclipse IDE ima podršku da automatski izgeneriše redefinisane metode toString(), equals() i hashCode().

5. Apstraktne klase i metode

U poglavlju o interfejsima smo videli da interfejs samo specificira ponašanje i samim tim njegove metode nemaju implementaciju. Moguće je i u Java klasi određene metode ostaviti bez implementacije i definisati samo njihovo zaglavlje. Metode koje nemaju implementaciju se nazivaju **apstraktne metode.** Nije moguće instancirati objekat klase koja sadrži apstraktne metode, jer takva instanca ne bi imala definisano ponašanje (za neke metode nije definisano šta rade). Slično, možemo definisati i klasu kao apstraktnu. **Apstraktna klasa** je klasa koju nije moguće instancirati. Klasa koja sadrži bar jednu apstraktnu metodu je sigurno **apstraktna klasa**. Postavlja se pitanje koji je scenario upotrebe klase za koju ne možemo kreirati objekat. Apstraktna klasa se koristi u nasleđivanju kao nadtip za druge klase. U apstraktnoj klasi se navedu podaci i ponašanje koje je zajedničko naslednicama, a putem apstraktnih metoda se predstavlja svako ponašanje koje nema smisla realizovati za taj apstraktni nadtip, nego se realizuje specifično za svaku naslednicu. Dakle, naslednik apstraktne klase je dužan da definiše ponašanje koje je u generalnom nadtipu ostavljeno apstraktno. Drugim rečima, klasa naslednica je dužna da implementira svaku apstraktnu metodu koju sadrži njen apstraktni predak. Izuzetak je ako je i sama klasa naslednica apstraktna. Tada i ona može da sadrži apstraktne metode (bilo one koje je sama uvela, bilo neke koje je nasledila od pretka).

Apstraktne klase se koriste kada je potrebno predstaviti podatke i ponašanje nekog apstraktnog entiteta, koji nema svoju manifestaciju, tj. u programu nije moguće napraviti objekat koji ga reprezentuje. Entitet se manifestuje u jednoj od svojih specijalizacija predstavljenih putem klasa naslednica. Ove klase naslednice je moguće instancirati. Podaci i ponašanje koji su zajednički različitim klasama naslednicama se mogu centralizovati u apstraktnu klasu koju svaka naslednica nasleđuje. Pošto će svaka naslednica imati isti API, tada je moguće korišćenjem polimorfizma svaku naslednicu posmatrati da je objekat tipa apstraktnog pretka i time uniformno upravljati svim naslednicama iako su one različitog tipa.

Pogledajmo primer evidencije komunalnih troškova. Svi troškovi su navedeni na priznanici koja ima različite stavke, npr. utrošak vode, utrošak grejanja i troškovi održavanja. Bez obzira na tip stavke, svaka stavka ima svoj naziv i metodu za računanje ukupnog iznosa stavke. Ovo ponašanje koje je zajedničko se može centralizovati u apstraktnu klasu StavkaPriznanice prikazanu u nastavku.

```
public abstract class StavkaPriznanice {
	protected String naziv;
	protected double iznos;
	...
	public abstract double izracunajIznos();
}
```

Vidimo da se apstraktna metoda definiše navođenjem ključne reči *abstract* i da takva metoda nema telo. Istom ključnom rečju se za klasu definiše da je apstraktna. Moguće je definisati klasu kao apstraktnu i ako ne sadrži neku apstraktnu metodu. Ako sadrži, mora se i klasa proglasiti apstraktnom navođenjem ključne reči *abstract* u zaglavlju klase. Prikazana klasa je apstraktna jer u priznanici u spisku stavki nikad neće postojati instanca ove klase. Priznanica sadrži konkretne stavke, npr. utrošak vode ili grejanja, ali nikad apstraktnu stavku predstavljenu klasom StavkaPriznanice. Konkretne stavke koje priznanica sadrži predstavljene su klasama koje nasleđuju apstraktnu klasu StavkaPriznanice. Naslednice su dužne da implementiraju nasleđenu apstraktnu metodu izracunajIznos() i mogu da uvedu nove atribute specifične za taj tip stavke. U nastavku je dat deo koda naslednica klase StavkaPriznanice koje predstavljaju stavku za utrošak vode, odnosno stavku za utrošak grejanja.

```
public class StavkaVoda extends StavkaPriznanice {

	private double jedinicnaCena;
	private int brojClanovaDomacinstva;
	private int brojStanaraUZgradi;
	private double potrosnjaUZgradi;

	public double izracunajIznos() {
		iznos =  (potrosnjaUZgradi / brojStanaraUZgradi) *
			brojClanovaDomacinstva * jedinicnaCena;
		return iznos;
	}

	...
}

public class StavkaGrejanje extends StavkaPriznanice {

	private double kvadraturaStana;
	private double jedinicnaCena;

	public double izracunajIznos() {
		iznos =  kvadraturaStana * jedinicnaCena;
		return iznos;
	}

	...
}
```

Vidimo da obe klase nasleđuju klasu StavkaPriznanice i na različit način implementiraju apstraktnu metodu nasleđenu iz pretka. Sada možemo da iskoristimo polimorfizam i da svim stavkama upravljamo jednako bez obzira na konkretan tip stavke. Pogledajmo sledeći izvod koda.

```
List<StavkaPriznanice> stavke = new ArrayList<>();
stavke.add(new StavkaVoda(14, 4, 54, 800));
stavke.add(new StavkaGrejanje(68, 32));
```

```
for (int i = 0; i < stavke.size(); i++) {
      StavkaPriznanice stavka = stavke.get(i);
      System.out.println("Iznos:" + stavka.izracunajIznos());
}
```

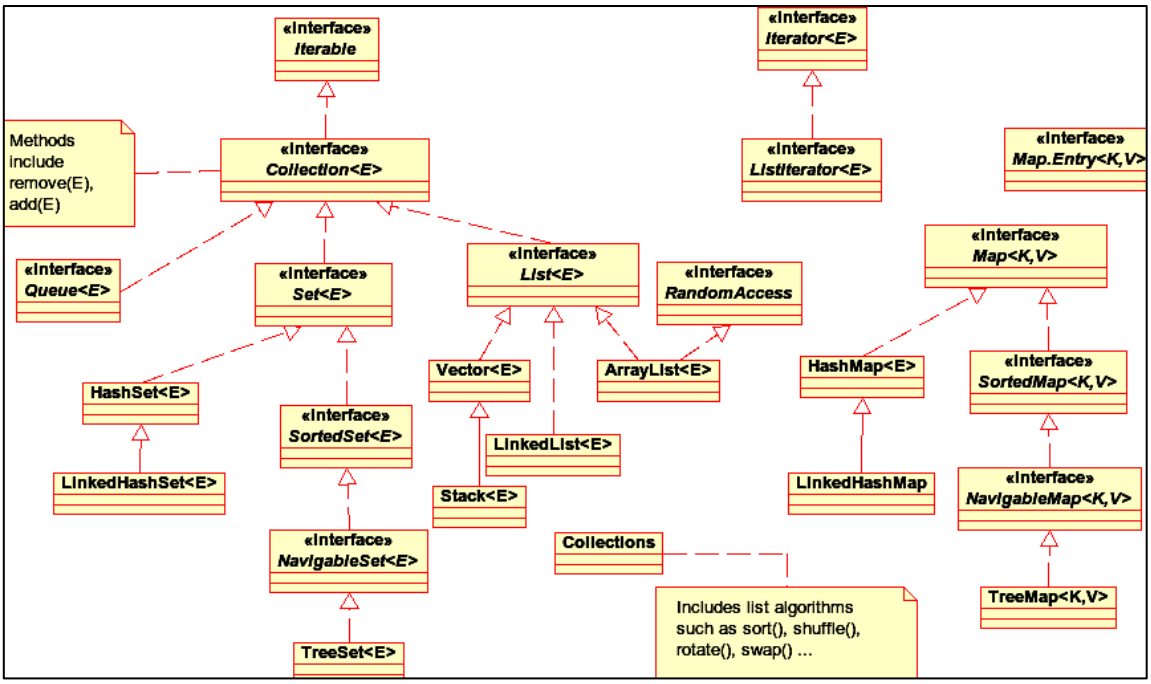
Kao i u ranijim primerima polimorfizma, i ovde vidimo da se objekti klasa naslednica tretiraju kao objekti pretka i da se u zavisnosti od konkretnog tipa stavke dinamički poziva metoda izracunajIznos() iz odgovarajuće naslednice.

6. Hijerarhija tipova u Java biblioteci klasa

Sada kada smo detaljnije upoznali koncept nasleđivanja, možemo se osvrnuti na ranije objašnjene pojmove koji se oslanjaju na ovaj koncept.

Java Collections Framework

Pogledajmo potpuniju hijerarhiju klasa iz Java Collections Framework skupa klasa koji predstavljaju različite strukture podataka.



* preuzeto sa http://hadooptutorial.info/short-notes-java-collections-framework/

Sa većinom klasa na slici smo se do sada susretali. Vidimo da postoji hijerarhija interfejsa koje klase implementiraju. Već smo objasnili da klase ArrayList i LinkedList implementiraju interfejs List. Ovaj interfejs nasleđuje interfejs Collection koji je u korenu hijerarhije jednog dela Java struktura podataka koji nazivamo kolekcije. Vidimo da su u drugoj grupi strukture koje realizuju mapu na različite načine i sve strukture ovog tipa implementiraju interfejs Map.

Vidimo da interfejs Collection nasleđuje interfejs Iterable. Ovaj interfejs je važan, jer kroz svaki objekat koji implementira ovaj interfejs možemo da iteriramo.

Iteriranje možemo vršiti i specijalnom varijantom for petlje koja je prikazana u nastavku na primeru iteriranja kroz listu objekata klase Drzava. Ova for petlja se naziva i unapređena for naredba (eng. *enhanced for statement*).

```
List<Drzava> drzave = new ArrayList<>();
drzave.add(new Drzava("sr", "Srbija"));
drzave.add(new Drzava("fr", "Francuska"));
drzave.add(new Drzava("it", "Italija"));

for (Drzava d: drzave) {
      System.out.println(d.getOznaka() + " " + d.getNaziv());
}

/* Ekvivalentan kod korišćenjem klasicne for petlje

for (int i = 0; i < drzave.size(); i++) {
      Drzava d = drzave.get(i);
      System.out.println(d.getOznaka() + " " + d.getNaziv());
} */
```

Prikazani tip for petlje možemo primeniti za prolazak kroz bilo koji objekat koji implementira interfejs Iterable. Takođe, ovaj tip for petlje moguće je koristiti i za prolazak kroz niz. Zaglavlje ovog tipa for petlje sadrži dva dela odvojena znakom dve tačke. Sa desne strane znaka se navodi naziv objekta kroz koji se iterira. Kao što smo pomenuli, ovaj objekat mora biti niz ili mora da implementira interfejs Iterable. Sa leve strane je lokalna promenljiva koja u svakoj iteraciji dobija vrednost sledećeg elementa iz kolekcije. Radi lakšeg razumevanja, u primeru smo u komentaru naveli kako bi isti kod izgledao korišćenjem klasične for petlje. Vidimo da promenljiva *d* u svakoj iteraciji ima vrednost trenutnog elementa u kolekciji.

Treba imati u vidu da ako koristimo ovaj tip for petlje, nije dozvoljeno vršiti izmene u kolekciji dok iteriramo. Ovo se odnosi samo na dodavanje ili izbacivanje elemenata iz kolekcije, dok je izmena sadržaja pojedinačnog elementa kolekcije dozvoljena. Dakle, sledeći kod će rezultovati izuzetkom pri izvršavanju.

```
for (Drzava d: drzave) {
      System.out.println(d.getOznaka() + " " + d.getNaziv());
      drzave.remove(d);
}
```

Korišćenjem klasične for petlje je dozvoljeno modifikovati kolekciju dok iteriramo, ali je problem što indeks označava poziciju elementa, a pozicije elementa se modifikacijom menjaju. Ovo može da dovede do grešaka pri izvršavanju programa, pa samim tim dolazimo do komplikovanijeg koda koji ima za cilj da se te greške otklone. Iz pomenutih razloga, ako je pri iteriranju potrebno vršiti modifikaciju kolekcije, preporučeni način za iteriranje je korišćenjem iteratora. Iterator je u Java Collections Frameworku predstavljen interfejsom Iterator. Svaki objekat koji implementira interfejs Iterable ima metodu iterator() koja vraća objekat tipa Iterator. U primeru je pokazano kako se kroz ranije prikazanu kolekciju država može iterirati korišćenjem iteratora uz istovremenu modifikaciju kolekcije. Konkretno, izbacuje se svaki element nakon što bude ispisan.

```
Iterator<Drzava> it = stavke.iterator();
while (it.hasNext()) {
      Drzava d = it.next();
```

```
        System.out.println(d.getOznaka() + " " + d.getNaziv());
        it.remove();
}
```

Vidimo da iterator ima metodu hasNext() koja vraća true ako postoji naredni element za iteriranje. Metodom next() se dobija naredni element. Izbacivanje poslednjeg elementa kojeg smo do tog trenutka dobili od iteratora se vrši metodom remove().

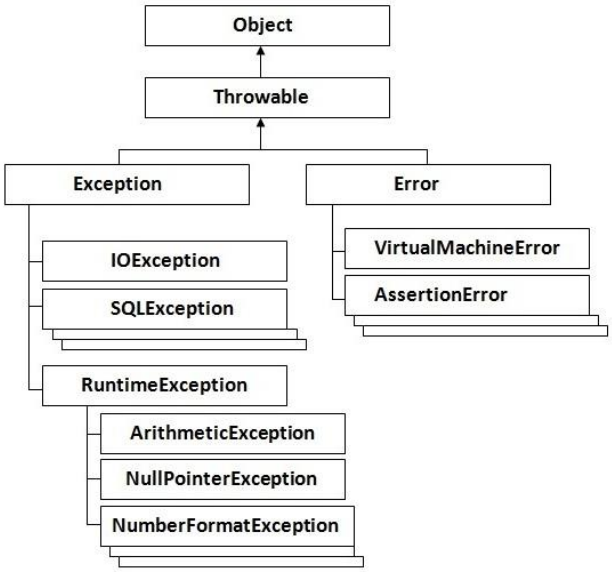
Ako obratimo pažnju na ranije prikazanu hijerarhiju klasa koje pripadaju Java Collections Frameworku, vidimo da postoji i klasa Collections. Ovo je pomoćna klasa koja putem statičkih metoda obezbeđuje često korišćene operacije nad kolekcijama. U tabeli su navedene neke od metoda ove klase.

Naziv metode	Opis
`binarySearch(List list, T key)`	Pretražuje specificiranu listu metodom binarne pretrage kako bi pronašao specificirani element key
`copy(List dest, List src)`	Kopira elemente iz jedne liste u drugu
`max(Collection coll)`	Vraća maksimalni element kolekcije
`min(Collection coll)`	Vraća minimalni element kolekcije
`reverse(List list)`	Menja listu tako da elementi imaju obrnut redosled
`shuffle(List list)`	Menja listu tako da redosled elemenata odredi na slučajan način
`sort(List list)`	Sortira rastuće specificiranu listu

Navedene metode max(), min() i sort() moraju da vrše poređenje elemenata da bi zaključile koji element je najveći ili najmanji, odnosno da bi sortirale elemente. Iz tog razloga, tip objekta koji lista skladišti mora da implementira interfejs Comparable. Implementacijom ovog interfejsa klasa se obavezuje da implementira metodu compareTo() koja poredi objekat klase sa nekim drugim objektom. Metode max(), min() i sort() pozivaju nad objektom metodu compareTo() da bi uporedile elemente.

Hijerarhija izuzetaka

Upoznavanje detalja nasleđivanja i polimorfizma nam omogućuje da pojasnimo još jedan konstrukt Java programskog jezika sa kojim smo se ranije upoznali. Reč je o izuzecima. Objasnili smo da je svaki tip izuzetka predstavljen odgovarajućom klasom. Između ovih klasa takođe postoji hijerarhija. Naredna slika ilustruje hijerarhiju izuzetaka u Java biblioteci klasa.



* preuzeto sa www.javatpoint.com

Klasa Throwable je predak svih grešaka i izuzetaka u Javi. Objekti ove klase mogu biti kreirani (izbačeni - eng. *throw*) kao reakcija na nepredviđeno ponašanje programa. Pri tome imamo dva tipa ovih reakcija. Prvi tip su greške predstavljene Error klasom i njenim naslednicama. Objekti ovih klasa predstavljaju ozbiljne probleme u radu aplikacije za koje nije predviđeno da budu obrađivani, već izazivaju prestanak rada aplikacije.

Drugi tip reakcije na nestandardni tok izvršavanja su izuzeci. Svaki specifičan izuzetak je podtip klase Exception. Kao što smo spominjali, specifični izuzeci mogu biti *checked* ili *unchecked.* Da podsetimo, za *checked* izuzetke mora da postoji obrada u kodu, pri čemu se za vreme kompajliranja proverava da li obrada postoji. Obrada može da bude hvatanje ili propagiranje izuzetka. Za *unchecked* izuzetke se ovakva provera da li je izuzetak obrađen ne vrši. Svi *unchecked* izuzeci nasleđuju klasu RuntimeException.

Dakle, definisanje novog tipa izuzetka podrazumeva definisanje nove klase koja nasleđuje klasu Exception. Ukoliko je reč o *unchecked* izuzetku, tada se nasleđuje klasa RuntimeException. U nastavku je prikazan primer definisanja novog tipa izuzetka. Izuzetak specificira da pri pretrazi studenata, student nije pronađen.

```
public class StudentNotFoundException
   extends RuntimeException {
      private long studentId;
      public StudentNotFoundException(long studentId) {
         this.studentId = studentId;
      }
      public long getStudentId() {
         return studentId;
      }
}
```

Prikazani izuzetak je *unchecked* izuzetak, jer nasleđuje klasu RuntimeException. Osim podataka koje je preuzeo od pretka (npr. poruka o grešci), ovaj izuzetak čuva i informaciju o identifikatoru studenta koji nije pronađen.

Kada smo se upoznali sa hijerarhijom izuzetaka, postaje jasniji mehanizam hvatanja izuzetaka. Ranije smo naveli da će *catch* naredba koja hvata izuzetak klase Exception, uhvatiti bilo koji tip izuzetka. Ovo se dešava zbog polimorfizma, tj. zbog toga što svaki izuzetak nasleđuje klasu Exception, pa je samim tim svaki izuzetak i tipa Exception. Generalno, izuzetak će biti uhvaćen u prvom *catch* bloku koji odgovara njegovom tipu. Izuzetak se naravno posmatra polimorfno, tako da je izuzetak objekat klase čija je instanca, ali i svakog nadtipa te klase. Na primer, gore prikazani izuzetak tipa StudentNotFoundException može biti uhvaćen u *catch* bloku koji hvata izuzetak tipa StudentNotFoundException, RuntimeException ili Exception.

7. Java anotacije

Da bismo ukazali na još jednu dobru praksu pri redefinisanju metoda, ukratko ćemo se upoznati sa još jednim konstruktom Java programskog jezika, a to su anotacije. **Anotacije** predstavljaju metapodatke u programu. Metapodaci su podaci o podacima. Ako govorimo o programu, programski kod koji se izvršava su sami podaci, a anotacije kao metapodaci daju dodatne informacije o kodu. Ove informacije može da koristi kompajler, drugi softverski alati ili sam program pri izvršavanju. Sintaksu i svrhu anotacija ćemo prikazati kroz primer korišćenja anotacija pri redefinisanju metoda.

Vratimo se na ranije prikazanu klasu AkcijskaStavkaProdaje koja nasleđuje klasu StavkaProdaje. Klasa naslednica redefiniše metodu ukupnaCena(). Ponovno je prikazana ova klasa, s tim da je ovog puta metoda ukupnaCena() označena anotacijom.

```
public class AkcijskaStavkaProdaje extends StavkaProdaje {

	protected double popust;
	protected int danaNaAkciji;

	@Override
	public double ukupnaCena() {
		return (jedCena * kol) * (100 - popust - danaNaAkciji)/100;
	}
}
```

Svaka Java anotacija ima naziv, a pre naziva se stavlja karakter @. U prikazu koda vidimo da je metoda ukupnaCena() dodatno opisana anotacijom @Override. Ova anotacija je napomena kompajleru da je reč o metodi nasleđenoj od pretka koja se u ovoj klasi redefiniše.

Praksa je da se za redefinisane metode postavi ova anotacija. Objasnićemo razlog za ovu praksu. Samo postavljanje ove anotacije ne utiče na primenu dinamičkog povezivanja metoda. Dakle, metoda će se ponašati kao što je ranije opisano bez obzira da li smo postavili ovu anotaciju. Kao što smo rekli, anotacija je samo napomena kompajleru da je reč o redefinisanoj metodi. Zašto dajemo kompajleru ovu informaciju? Zato jer postoji mogućnost da napravimo grešku u zaglavlju metode i da ne napišemo potpuno isto zaglavlje kao u metodi pretka. Podsetimo da redefinisana metoda mora da ima isto zaglavlje kao metoda pretka koju redefiniše. Na primer, moguće je napraviti slovnu

grešku tako da se naziv metode neznatno razlikuje od naziva metode pretka. Tada to više ne bi bila redefinisana metoda nego potpuno nova metoda, što je validno ponašanje i kompajler ne bi prijavio nikakvu grešku. Sa druge strane, dinamičko povezivanje ne bi radilo, jer naslednik ni ne sadrži redefinisanu metodu. Ovakvu grešku je relativno teško uočiti u programu, pa je iz tog razloga praksa da se stavlja anotacija @Override kojom se kompajleru naglašava da sledi metoda koja predstavlja redefiniciju metode pretka. U ovom slučaju, ako se napravi greška u zaglavlju metode, kompajler će prijaviti grešku i ukazaće nam da ispravimo zaglavlje metode.