PANEVROPSKI UNIVERZITET APEIRON

FAKULTET INFORMACIONIH TEHNOLOGIJA

BANJA LUKA

Seminarski rad

NEKI PRIMJERI IZ PROGRAMSKOG JEZIKA “JAVA"

Nastavni predmet: ***Principi programiranja***

Predmetni nastavnik/asistent:
Prof. dr Zoran Ž. Avramović
Igor Lastrić

Student:
Siniša Božić
192-20/RITP

Banja Luka, 2021.

SADRŽAJ

UVOD

Sredinom 1990ih godina, u praskozorje Internet revolucije, nastao je programski jezik Java. U to vrijeme na tržištu su dominirali C i C++, sa više od tri četvrtine tržišnog učešća. Korisnici su vrlo brzo prihvatili Javu, koja je u roku od dvije godine od predstavljanja postala jedan od pet najpopularnijih jezika opšte namjene. Do današnjih dana, to je jedan od tri najkorištenija programska jezika koji ne samo da je snažno uticao na tržišna kretanja, nego se može slobodno tvrditi da je izvršio revoluciju u programiranju i široko uticao na svijet oko nas. Java je nastala sa primarnom namjenom da bude univerzalni programski jezik koji je moguće pokrenuti ne samo na računarima, već i na mnogim ostalim platformama, kao što su razne vrste nezavisnih uređaja (kontrolera, bankomata, elektroničkih uređaja u pokućstvu, industriji itd.). S druge strane, u doba predstavljanja Jave, Internet je postajao sve moćniji medij i Java je iskoristila proliferaciju Interneta kako sa korisničke, a naročito sa serverske strane što je vrlo brzo dovelo do velikog rasta popularnosti ovog programskog jezika, a sa druge strane, Java je učinila Internet sigurnijim mjestom.

Treba pomenuti da su već pomenuti programski jezici C i C++ najbliži srodnici Jave. Od C, Java je preuzela sintaksu, a objektni model od C++[1]. Također, Microsoft je kreirao sopstveni programski jezik C# koji je vrlo blizak Javi što je potvrda pravog smjera kojim je Java krenula.

Ključne riječi koje opisuju Javu su:

Jednostavnost
Sigurnost
Prenosivost
Objektna-orjentisanost
Robusnost
Višestruko izvršavanje
Neutralna arhitektura
Intrepretiranost
Visoke performanse
Distribuiranost
Dinamičnost

Naročito treba izdvojiti orjentaciju Jave kao objektivno-orjentisanom programiranju, budući da je to vjerovatno najpopularnija programska paradigma korištena u zadnjih 20 godina. U Javi svaki element je objekt i svaki program sadrži barem jednu klasu.

[1]Kao i od programskog jezika Smalltalk.

1 PETLJE

1.1. "FOR“ PETLJA

Sintaksni oblik "for" petlje u Java programskom jeziku je[2]:

```
for (inicijalizacija; uslov; iteracija)
{
neki kôd
}
```

Prilikom definisanja ove petlje, potrebno je navesti i inicijalizovati varijablu koja će kontrolisati petlju, odnosno služiti kao njen brojač. Uslov petlje je logički operator koji određuje da li će se petlja ponavljati. Iteracija definiše kako će brojač da se mijenja. Ova petlja će se izvršavati sve dok je uslov `istinit`, i tek kad on promijeni stanje u `nestinit` petlja prestaje i program se izvršava daljim redom odozgo na dole, s lijeva na desno.

Primjer:

```
//ispisuje kvadratni korijen brojeva od 0-99
class KvadratniKorijen {
public static void main(String args[]) {
        int broj;
        double korijen;

        for (broj=1; broj<100; broj++) {
            korijen=Math.sqrt(broj);
               System.out.println("Kvadratni korijen broja "+ broj + " je
"+korijen);
            }
      }
}
```

Napomena: definisana je varijabla tipa `int` i varijabla tipa `double` (budući da je korijen decimalni broj). Petlja počinje iteraciju od broja 1 zaključno sa brojem 99, za svaki broj računa kvadratni korijen (pozivajući klasu `Math` i njen metod `sqrt()`), te se vraća na početak, povećava brojač za jedan (izraz broj++ znači *post* povećanje varijable broj za jedan[3]). Kada dosegne broj 99 petlja završava.

"For" petlja ima više varijacija i mogućih upotreba.

Na primjer, moguće je da se petlja kreće u pozitivnom ili negativnom smjeru, sa proizvoljno velikom iteracijom. Slijedeći program ispisuje brojeve od 100 do -95 u koracima od po 5:

[2]Za detaljan opis pogledati Oracle Java Documentation:
https://docs.oracle.com/javase/specs/jls/se15/html/jls-14.html#jls-14.14.1

[3]Npr. izraz ++broj prvo uvećava broj za jedan pa ga onda ispisuje (bez obzira na neku drugu operaciju koju treba izvršiti sa tim brojem), a izraz broj++ prvo uzima u obzir vrijednost varijable pa je onda povećava.

```
// "for" petlja koja se krece u negativnom smjeru
class negativnaPetlja {
public static void main(String[] args) {
        int broj;
        for (broj=100; broj>-100; broj-=5)
            System.out.println(broj);
      }
}
```

Napomena: kada se radi o jednostavnim izrazima, vitičaste zagrade bloka petlje se mogu izostaviti. Ova petlja počinje od broja 100, uz uslov da je broj veći od -100, te iteriše u dekrementalnim koracima od po 5. Ispisuju se brojevi od 100 do -95 u koracima od 5. Izraz `broj-=5` skraćeno znači: `broj=broj-5`.

"For" petlja testira uslov pri početku, ukoliko bi se desilo da je uslov `neistinit`, petlja se ne bi izvršila i kompajler bi prijavio grešku, kao na slijedećem primjeru:

```
for (broj=10; broj<5; broj++) {
neki kôd
}
```

Ova petlja se uopšte ne bi izvršila jer je početni uslov netačan, odnosno broj 10 nije manji od 5.

"For" petlja omogućava korištenje više brojača i kontrolnih varijabli. Na primjer:

```
// koristenje vise varijabli u "for" petlji
class viseVarijabli {
public static void main(String args[]) {
    int a,b;

    for (a=0, b=10; a<b; a++, b--)
        System.out.println("a i b: " + a + " " + b);
      }
}
```

Napomena: ovo je specifičan i rjeđe korišten slučaj, u kome petlju kontrolišu dve varijable (moguće je i više od toga ali takve petlje postaju komplikovane za razumjevanje). Budući da petlja "for" omogućava inicijalizaciju varijabli, moguće je bilo izbaciti red `int a,b;` :

```
for (int a=0, b=10; a<b; a++, b--)
```

Petlja postavlja inicijalne vrijednosti varijabli `a` i `b`, respektivno, na 0 i 10, postavlja uslov da je `a` manje od `b`, te potom iterira uvećavajući za jedan varijablu `a` te istovremeno smanjujući za jedan varijablu `b`. Program iterira sve dok je `a` manje od `b` te ispisuje:

```
a i b: 0 10
a i b: 1 9
a i b: 2 8
a i b: 3 7
a i b: 4 6
```

Postoji i skraćena verzija "for" petlje koja se naziva `enhanced for` petlja, odnosno unaprijeđena petlja. Ona se koristi za iteriranje kolekcije objekata, kao što je niz. Ova petlja će se obraditi u tački 2. ovog rada.

1.2. "WHILE" PETLJA

"While" petlja se u programskom Jeziku java deklariše kao[4]:

```
while (uslov)
{
neki kôd;
}
```

Pri tome, uslov je logički iskaz koji može biti `tačan` ili `netačan`. Dok je uslov `tačan`, petlja se izvršava, kad uslov postane `netačan`, petlja prestaje.

Idući primjer pokazuje kako se jednostavno mogu ispisati slova abecede korištenjem ove petlje.

```
// primjer "while" petlje
class WhilePrimjer {
public static void main(String args[]) {
    char karakter;
    karakter= 'a';
        while (karakter<'z') {
            System.out.print(karakter);
            karakter++;
            }
      }
}
```

Napomena: definisana je varijabla tipa `char` (koja se inicijalizuje jednostrukim navodnim znacima). Petlja je tako inicijalizovana da je varijabla karakter, početnog stanja `'a'`, manja od krajnjeg karaktera abecede, odnosno `'z'`. Sve dok je taj uslov tačan, petlja vrši iteraciju i ispisuje znak po znak. Nakon prve iteracije, varijabla karakter se uvećava za jedan, odnosno sa karaktera `'a'` se povećava na `'b'` i tako redom.

Suštinski, petlje "for" i "while" predstavljaju jednu te istu stvar iskazanu na različite načine. Petlja "for" omogućava i zahtjeva deklarisanje i inicijalizaciju varijable tokom deklarisanja same petlje, dok kod "while" petlje se deklarisanje i inicijalizacija kontrolne varijable obavlja

[4]Za detaljan opis pogledati Oracle Java Documentation:
https://docs.oracle.com/javase/specs/jls/se15/html/jls-14.html#jls-14.12

prije same petlje. Obadvije petlje koriste logički iskaz za verifikaciju iteracije, dok se brojač u "for" petlji također obično deklariše tokom deklarisanja same petlje[5], dok se u "while" petlji brojač deklariše na kraju izvršenja kôda unutar bloka petlje.

Podvarijanta "while" petlje je "do-while" petlja, čija je osobenost da će se izvršiti barem jednom, bez obzira na istinitost logičkog uslova, i ova petlja se obično rjeđe koristi.

Sumarizovano dajemo pregled ove tri petlje:

Poređenje	**For petlja**	**While petlja**	**Do while petlja**
osnovno	kontrola toka programa koja iteriše blok programa na osnovu logičkog iskaza	kontrola toka programa koja iteriše blok programa na osnovu logičkog iskaza	Kontrola toka programa koja izvršava dio programa bar jednom, bez obzira na stanje logičkog iskaza, a dalji dok izvršavanja (iteracije) zavisi od logičkog iskaza
korištenje	Ako je broj iteracija poznat	Kada je broj iteracija nepoznat ili nije fiksno zadat	Ako se ne zna tačan broj iteracija a postoji potreba da se dio programa izvrši bar jednom
sintaksa	`For(inicijalizacija ; uslov; iteracija) { neki kôd }`	`While (uslov) { neki kôd }`	`Do { neki kôd } while (uslov);`
primjer	`For (int i=1; i<=10; i++) { System.out.println( i); }`	`Int i=1; while (i<=10) { System.out.println(i ); i++; }`	`Int i=1; do { System.out.println(i) ; i++; while (i<=10) { }`

Tabela 1: Sumarni pregled karakteristika petlji u programskom jeziku Java

1.3 BREAK, CONTINUE kao načini kontrole izvršavanja petlje

Izraz break u bloku petlje prekida njeno izvršavanje, bez obzira na stanje logičkog uslova, i kontrolu izvršavanja programa usmjerava na kôd koji se nalazi nakon petlje. Na primjer:

```
// koristenje break za prekid petlje
class Break {
public static void main(String args[]) {
        for (int i = 0; i < 10; i++){
```

[5]"for" petlja se može definisati i kao:
for (int i=0; i <10) {
neki kôd;
i++
}

```
            if (i == 5)
                break;
            System.out.println("i: " + i);
            }
System.out.println("Petlja zavrsena");
      }
}
```

Napomena: program ispisuje brojeve od 0-9. Ova petlja ispisuje brojeve počev od 0, iteracija povećava korak po korak vrijednost brojača. Kada brojač bude imao vrijednost 5, `break` prekida petlju. Program ispisuje vrijednosti varijable i od 0 – 4. Ukoliko bi umjesto `break` unijeli `continue`, program bi nastavio ispisivanje sve do broja 9. Dakle, za razliku od `break`, continue ne izlazi iz petlje već nastavlja njenu iteraciju sve do momenta kada logički uslov sa početka petlje postane neistinit.

2 NIZOVI (ARRAYS)

Niz predstavlja skup varijabli istog tipa, kojima je dodjeljeno isto ime. U programskom jeziku Java, nizovi mogu biti višedimenzionalni[6], ali se najčešće koriste jednodimenzionalni. Nizovi rješavaju problem dodjele više vrijednosti jednoj varijabli. Npr. niz može da sadrži spisak isplaćenih plata, listu dnevnih temperatura, dobitaka na lutriji itd. U Javi, niz predstavlja objekat (za razliku od običnih, tzv. primitivnih varijabli), što pravi određenu razliku u manipulaciji nizovima a naprednijim korisnicima može donijeti i određene benefite[7].

Opšta sintaksa za kreiranje niza je[8]:

```
tip ime niza [] = new tip[veličina]
```

Pošto su nizovi objekti, prilikom kreiranja niza prvo se definiše tip niza i naziv varijable niza, a onda (kao i svi drugi objekti), koristi se `new` čime se alocira memorija. Na primjer, idući niz kreira niz brojeva sa 10 elemenata i dodjeljuje ih varijabli `primjer`:

```
int primjer[] = new int[10];
```

Niz deklarisan na ovaj način je stvoren kao prazan objekat, te ga je moguće popuniti iteriranjem i dodjelom pojedinačnih vrijednosti njegovim indeksnim brojevima:

```
// deklarisanje, inicijalizacija, popunjavanje i ispis jednodimenzionalnog niza

class Niz {

public static void main(String args[]) {

int primjer[]=new int[10];


            for (int i = 0; i < 10; i++)

            primjer[i]=i;


            for (int i = 0; i < 10; i++)

                  System.out.println("Element [" +i + "]: "+primjer[i]);

      }

}
```

Napomena: iz navedenog primjera se vidi da se niz popunjava iteracijom, kao da se iteracijom i ispisuje njegov sadržaj.

Indeksi niza počinju od 0, dakle ako niz ima veličinu 10, indeksi se kreću od 0-9. To je tako u većini programskih jezika pa i u Javi.

---

[6]Npr. dvodimenzionalni niz je skup nizova, niz koji sadrži više nizova.
[7]Npr. prazan niz je garbage-collected.
[8]Za detaljan opis pogledati Oracle Java Documentation:
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Arrays.html

Niz je moguće popuniti i odmah, pri deklarisanju i inicijalizaciji:

```
int primjer [] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
```

U ovom slučaju nije potrebno navoditi veličinu niza, jer je veličina očigledna iz samog seta unesenih podataka.

Veličina niza je fiksna u Javi, što znači da se jednom kreiran niz ne može povećavati ili smanjivati. Naravno, neodgovarajuća iteracija niza (npr. ako iterator prelazi veličinu niza) predstavlja grešku u izvršavanju takvog programa:

```
primjer [11]=22; //ispisuje gresku


for (int i=0; i<15; i++) {
neki kôd
} // iteracija preko velicine niza takodjer ispisuje gresku
```

Ako bismo htjeli promjeniti određeni element niza, to se jednostavno postiže pristupanjem njegovom indeksnom elementu i dodjeli nove vrijednosti[9]:

```
primjer [0]=1;
```

Iteracija niza se može postići i "for-each" petljom, koja predstavlja skraćeni oblik "for" petlje i pogodna je za korištenje radi svoje kompaktnosti:

```
String[] mobiteli = {"Samsung", "Apple", "Xiaomi", "Motorola"};
for (String i : mobiteli) {
  System.out.println(i);
}
```

[9]Naravno, ako bi kreirali i manipulisali nizom stringova, koristili bi duple navodnike za svaku takvu manipulaciju. Npr. primjer[0]="Test";

3 ARRAYLIST

ArrayList je klasa u Javi[10], koja je nastala kako bi se prevazišla ograničenja koja postoje kada se koriste nizovi. Može se reći da je ArrayList zapravo dinamički niz jer mu veličina nije fiksno zadata i lako se može mijenjati, po potrebi. Po ostalim karakteristikama, ArrayList služi istoj svrsi kao i niz, odnosno sadrži više vrijednosti varijabli istog tipa.

Opšti način kreiranja ArrayList je:

```
ArrayList<tip> imeVarijable=new ArrayList<tip>();
```

Ako bi uporedili niz i ArrayList, operacije koje se mogu na obadva izvršiti su:

	Niz	ArrayList
Kreiranje	`String lista [] = new String[10];`	`ArrayList<String> lista=new ArrayList<String>();`
Veličina	`int i=lista.length;`	`int i=lista.size();`
Ispis elementa	`String s=lista[3];`	`String s=lista.get(3);`
Upis elementa[11]	`lista[3]=s;`	`lista.set(3, s);`
Ispis	`"For" ili "for-each" petlja`	`"For" ili "for-each" petlja`

Tabela 2: Uporedne karakteristike struktura Niz i ArrayList

Ako ove obadve strukture podržavaju iste mogućnosti, postavlja se pitanje zašto postoji ArrayList i u čemu je njegov značaj. Da bi ovo spoznali, potrebno je navesti razlike:

	Niz	**ArrayList**
Dodati element na kraju	Ne podržava	`lista.add(s);`
Dodati element u sredini	Ne podržava	`lista.add(indeksSredina, s);`
Dodati element na početku	Ne podržava	`lista.add=lista.add(0,s);`
Izbrisati element	Ne podržava[12]	`lista.remove(3);`

Tabela 3: Razlike između struktura Niz i ArrayList

[10]Za detaljan opis pogledati Oracle Java Documentation:
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/ArrayList.html
[11]Ukoliko se dodaje predefinisana varijabla, dupli navodnici nisu potrebni.
[12]Ako bi unijeli list[3]=null; , time bi faktički izbrisali vrijednost datog elementa, ali ta null vrijednost bi ostala u strukturi niza i pri iteraciji bila prisutna. S druge strane u ArrayList, remove potpuno briše dati element.

4 HASHMAP

HashMap je takva struktura[13] koja omogućava skladištenje podataka u parovima. Svaki par ima svoj ključ, koji je jedinstven, i njemu pridruženu vrijednost. Na primjer, ako bi htjeli skladištiti ovako uređen set podataka:

Grana nauke: disciplina		**Šifra proizvoda: naziv artikla**
Pravne nauke: Međunarodno pravo Pravne nauke: Obligaciono pravo Medicina: Anatomija Medicina: Molekularna biologija Informatika: Računarske mreže Informatika: Sistemski softver	ili	6846865: Aktuator turbine 9684684: Servo pumpa 7485664: Filter motornog ulja 8768468: Zadnji izduvni lonac

Tabela 4: Primjer tipa podataka pogodnog za skladištenje u HashMap

niti niz, niti ArrayList ne bi bili pogodni, jer oni čuvaju samo elemente istog tipa, koji nisu povezani jedni sa drugima. Iz tog razloga se HashMap (klasa u Javi, a u drugim programskim jezicima poznat kao Dictionary) naziva i asocijativni niz, zato što jedinstvenom ključu asocira neku vrijednost. Kao i kod nizova i ArrayList, i ključ i vrijednost u HashMap mogu biti bilo primitivne varijable[14], bilo objekti.

Sintaksa za kreiranje HashMap glasi:

```
HashMap<tip ključa, tip vrijednosti> primjerHashMap = new HashMap<>();
```

Jednostavan primjer deklarisanja i upisa u HashMap slijedi:

```
import java.util.HashMap;
public class Main {
public static void main(String[] args)  {

 HashMap<Integer[15], String> licniDokumenti = new HashMap<>();
        licniDokumenti.put (212133, "Jovan Jovanovic");
        licniDokumenti.put (162348, "Marko Markovic");
        licniDokumenti.put(8082771, "Marijana Marijanovic");
        System.out.println(licniDokumenti);
        }
      }
```

Ispis:

[13]Za detaljan opis pogledati Oracle Java Documentation:
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/HashMap.html
[14]Pogledati detaljan spisak primitivnih vrijednosti:
https://docs.oracle.com/en/java/javase/15/docs/api/jdk.jdi/com/sun/jdi/PrimitiveValue.html
[15]Kolekcije kao što je HashMap za tip varijable ne koriste primitivne varijable već tzv. generics.

```
{212133=Jovan Jovanovic, 8082771=Marijana Marijanovic, 162348=Marko
Markovic}
```

Ukoliko bi pokušali dodati novu vrijednost koristeći već korišteni ključ, ta bi vrijednost bila dodijeljena tom ključu a stara prepisana:

```
licniDokumenti.put (212133, "Dragan Draganovic");
```

Ako želimo ispisati određenu vrijednost, pristupamo joj putem ključa:

```
System.out.println(licniDokumenti.get(212133));
```

Kao što je vidljivo iz dokumentacije, HashMap sadrži još mnoštvo metoda kojima se može manipulisati sadržajem ove strukture. Još ćemo pomenuti način iteracije[16] koji je nešto komplikovaniji:

```
for (Map.Entry<Integer, String> entry: licniDokumenti.entrySet()) {
   System.out.println(entry);
}
```

Da bi iterirali HashMap moramo koristiti `Map.Entry` klasu, definisati varijablu i koristiti `entrySet()` metod, jer se HashMap sastoji od parova ključ i vrijednost definisanih kao `Map.Entry` parovi, tako da se iteriraju parovi a ne njihovi ključevi i vrijednosti.

[16]Iako se HashMap može ispisati korištenjem standardne System.out.println metode (za razliku od nizova i ArrayList), iteracija je podrazumijevani način kako ispisa tako i drugih operacija kojima se sadržaj ove strukture na neki način mijenja.

5 REKURZIJA (RECURSION)

Rekurzija je programska tehnika u kojoj metod[17] poziva sam sebe u cilju rješavanja nekog problema. Ovakav metod se zove rekurzivnim. Mnogi problemi u programiranju se mogu riješiti rekurzijom.

Klasični primjer korištenja rekurzije je računanje faktorijela. Faktorijel od bilo kog prirodnog broja n je proizvod svih brojeva od 1 do n. Dakle, faktorijel od 5 je 120, odnosno 5x4x3x2x1.

U Javi, za računanje faktorijela bi mogli napisati metod koji koristi iterativnu petlju, kao npr.

```
// racunanje faktorijela putem iteracije
class Faktorijel {


    public static void main(String[] args) {
        int n=5;
        int fact;
        fact=factorial(n);
        System.out.println("Faktorijel od n="+n+ " je "+ fact + ".");
    }


    private static int factorial(int n) {
        int f=1;
        for (int i=1; i<=n;i++)
            f = f*i;
        return f;
    }
}
```

Ispis: `Faktorijel od n=5 je 120.`
Napomena: logika izračuna faktorijela je smještena u metod factorial, koji koristi "for“ petlju za iteraciju od `1` do `n=5`. U prvoj iteraciji, varijabla `f=f*1`, odnosno `1=1*1`; u drugoj iteraciji brojač se povećava na `2` pa je `f=1*2`; u trećoj iteraciji imamo `f=2*3`; u četvrtoj `f=6*4`, i konačno, petoj; `f=24*5` što je jednako `120`. Metod vraća vrijednost varijable `f` koja se poziva u main bloku kao `factorial(n)` i dalje ispisuje.

Rekurzivno rješenje se razlikuje u metodu koji bi u tom slučaju glasio:

```
    private static int factorial(int n) {
        if (n==1)
            return 1;
        else
            return n * factorial(n-1);
    }
```

[17]Ono što je funkcija u ostalim programskim jezicima, to je metod u Javi.

Ispis ovakvog programa je potpuno isti, međutim način izračunavanja faktorijela je drugačiji. Ukoliko je `n` jednako `1`, tada metod također vraća `1`. Ukoliko je `n=2`, metod će vratiti `2* faktorijel(2-1)`, za `n=3` metod vraća `3*faktorijel(3-1)` odnosno `6`; za `n=4` metod vraća `4*faktorijel(4-1)` odnosno `24`, i konačno, za `n=5`, metod vraća `5*faktorijel(5-1)` što konačno iznosi `120`. Dakle metod poziva sam sebe počevši od najmanje vrijednosti varijable `n` do njene dodijeljene vrijednosti[18].

[18]Detaljnije o rekurzivnim algoritmima u Javi i internim načinima njhovog izračuna pogledati na: https://introcs.cs.princeton.edu/java/23recursion/

6 OBJEKTNO-ORJENTISANO PROGRAMIRANJE

Centralna paradigma u programskom jeziku Java je objektno-orjentisano programiranje[19] (OOP). Skoro svaki dio kôda u Javi je objekat[20] te je OOP tako integralni dio ovog programskog jezika. Zato je razumijevanje osnovnih postulata OOP vrlo korisno za svakog korisnika koji koristi ili ima namjeru da koristi ovaj programski jezik.

OOP koncept nije novina, riječ je o principima emuliranja stvarnosti fizičkog svijeta nastalih još 1960ih godina u Norveškoj, na Univerzitetu u Oslu, gdje je nastao prvi OOP programski jezik Simula[21]. Simula je prvi programski jezik koji je koristio klase i objekte, iz njega se razvio Smalltalk[22], te zajedno sa C++, može se reći da su to preteče programskog jezika Jave.

OOP je nastao kao odgovor narastajućoj kompleksnosti programa kreiranih u nekih drugim paradigmama, prije svega strukturalnom programiranju (LISP i C), jer takvi programi su se sastojali od uvećavajućeg broja linija (tipa hiljada pa i više). OOP je moćno oruđe kojim se kompleksni programi mogu bolje kontrolisati na način da se njihov kôd organizuje oko objekata, tako da se zapravo programski kôd pretvara u interakciju između objekata koji su tako definisani da simuliraju procese koji se događaju u stvarnosti, budući da svaki programski kôd, u većoj ili manjoj mjeri, je zasnovan ili postoji radi rješavanja nekog konkretnog realnog problema.

Klasa je bazična jedinica u OOP. To je apstraktni pojam koji se koristi u svrhu klasifikovanja podataka na osnovu njihovih zajedničkih karakteristika. Primjera radi, ukoliko pišemo program koji će obračunavati poreze neke firme, najneophodnije bazne klase bi mogle biti:

- Porezi (klasa bi sadržala sve poreze koje jedna firma mora da evidentira, npr. indirektni/direktni porezi),
- Sektori firme (dio firme odgovoran za date poslovne te administrativne aktivnosti,npr. proizvodnja, finansije, pravna služba, a koji imaju veze sa upravljanjem porezima),
- Eksterni organi (institucije izvan firme koje korespondiraju sa istom u vezi regulisanja poreza, npr. Poreska uprava, Uprava za indirektno oporezivanje, Ministarstvo finansija).

Ove bazne klase mogu da imaju svoje podklase (koje su već navedene za svaku navedenu stavku).

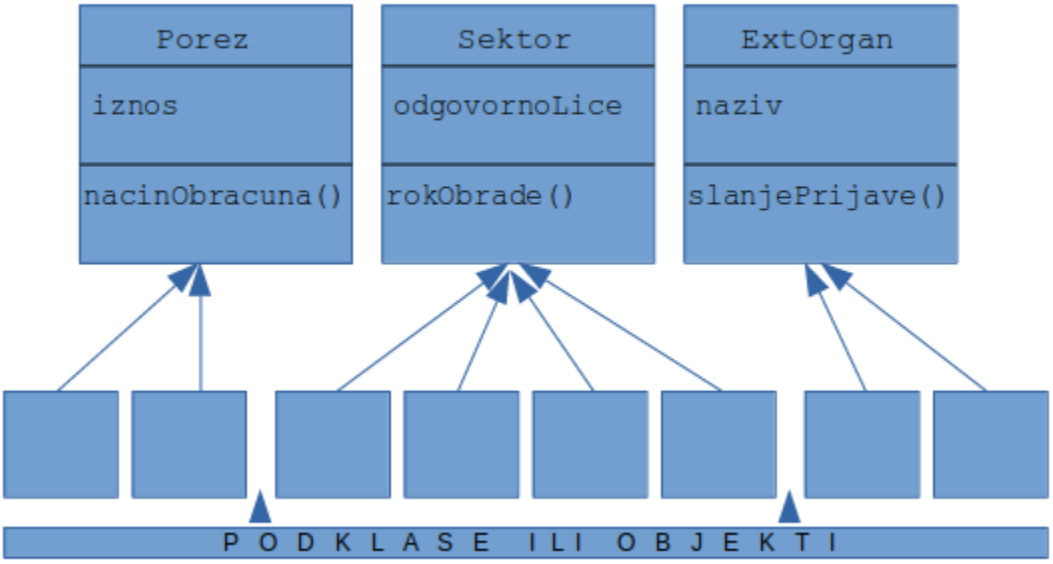
[19]Ipak, Java od verzije 8 podržava i funkcionalno programiranje tako da je zapravo riječ o tzv. multi-paradigm programskom jeziku.

[20]Vrlo trivijalan primjer je ispis, odnosno komanda System.out.print(), koja je u stvari poziv klase java.lang.System (koja nasljeđuje od krovne klase java.lang.Object) i koja poziva metod public static final PrintStream out.

[21]Skraćenica od SIMUlation LAnguage.

[22]Više o OOP istoriji na: Object-oriented programming: Some history, and challenges for the next fifty years https://www.sciencedirect.com/science/article/pii/S0890540113000795

Potrebno je razgraničiti pojmove klase i objekta. Dok su klase zajednički prostor, objekti su konkretni skupovi podataka koji se klasifikuju kao instance klase. Može se upotrijebiti analogija sa fajl-sistemom pa reći da ako je klasa direktorij ili folder na hard disku, objekat te klase je konkretan fajl smješten u tom direktorijumu.
Slijedeći dalje primjer programa koji proračunava poreske obaveze, moguće je koristeći UML dijagram predstaviti ovakvu organizaciju programa:



Prilog 1: Jedna od mogućih organizacija klasa ili objekata u programu za obračun poreskih obaveza

Definisane su tri klase[23], `Porez`, `Sektor`, `Extorgan`, unutar svake njih varijable, respektivno, iznos, odgovornoLice, naziv, te metodi, `nacinObracuna()`, `rokObrade()`, `slanjePrijave()`. Ukoliko želimo da dalje grupišemo program u sitnije cjeline, mogli bismo za svaku od ovih klasa stvarati podklase, ili, ako to ne želimo (ako nije neophodno za dalje rješavanje problema), onda možemo stvarati instance ovih klasa – objekte.
Ne ulazeći u ovom momentu u dalje razlaganje ovog problema, ovdje treba još jednom napomenuti zašto ovo radimo i kako bi mogli dizajnirati program a da ne koristimo OOP već recimo klasično proceduralno ili funkcionalno programiranje. Može se tvrditi da OOP

[23]Naravno ovo je samo jedna od mogućih interpretacija navedenog problema i za potrebe ovog rada je uproštena, izostavljajući brojne detalje.

zahtjeva i filozofski pristup rješavanju problema, jer se odmah na početku od programera očekuje da napravi virtuelni model problema i da uspostavi vezu sa realnim svijetom. Zato identifikujemo osnovne nosioce radnji, odnosno strukturu klasa i objekata, te definišemo za svaku klasu ono što ona 'zna' (iznos, `odgovornoLice`, naziv), te ono što čini (`nacinObracuna()`, `rokObrade()`, `slanjePrijave()`). Drugačije rečeno, `iznos`, `odgovornoLice`, naziv su varijable a `nacinObracuna()`, `rokObrade()`, `slanjePrijave()` metodi. Uz pomoć osnovnih postulata OOP, a to su apstrakcija, enkapsulacija, nasljeđivanje i polimorfizam, kontrolišemo odnose između klasa i objekata i na taj način kontrolišemo tokom programa, koji ovako kreiran ne ide nužno odozgo na dole. Ako bi primjenili klasični, proceduralni stil, jednostavno bi problem rješavali kreiranjem varijabli i po potrebi funkcija, pa bi, primjera radi, sve varijable koje su bile prethodno definisane u OOP slučaju u klasama i objektima, bile 'ogoljene' u ne-OOP varijanti, a metodi koji su se također nalazili u klasama i objektima, bi bili deklarisani u samom tijelu programa. Sa jedne strane, ne-OOP kôd bi bio jednostavniji u smislu da ne bi morali brinuti o odnosima definisanih klasa i objekata, a sa druge, vjerovatno bi sam kôd bio nepregledniji, uz višestruko korištenje jednog te istog kôda[24].

6.1 APSTRAKCIJA (ABSTRACTION)

Apstrakcija je jedan od ključnih elemenata OOP[25]. Apstrakcija je zapravo percepcija kompleksnosti nekog sistema, problema, objekta. Na primjer, automobil je sa stanovišta konstrukcije vrlo složeni objekat koji se sastoji od desetina hiljada dijelova. Čovjek kao korisnik automobila, u većini slučajeva pak ne razmišlja o toj kompleksnosti već je svodi na uprošćenu koncepciju koja kaže da je automobil sredstvo za transport. Korisnik automobila ignoriše njemu nepotrebnu kompleksnost, jer mu je to nepotreban teret[26]. Apstrakcija se može posmatrati kao složeni sistem od više slojeva klasifikacija. Na primjeru automobila, svana je to jedinstven objekat, međutim iznutra se sastoji od više podsistema kao što su upravljanje, pogon, električni, mehanički sistemi itd.

[24]Krajnji sud da li je OOP 'bolji' u smislu efikasnosti i smanjenja kompleksnosti od ne-OOP kôda nije moguće paušalno dati, već se to razlikuje u zavisnosti od tipa problema koji rješavamo, njegove kompleksnosti, kao i nivoa znanja programera. OOP je vjerovatno pogodniji za komplikovanije programe.

[25]Ovdje riječ o apstrakciji u širem smislu, u OOP konotaciji. Apstrakcija u Javi se odnosi na stvaranje apstraktne klase koja ima određene osobine i koja se koristi u određenim situacijama kako je pokazano na primjeru klase Oblik.

[26]Naravno, konstruktor automobila ima potrebe za uvećanom percepcijom ali i tada važi objektno-orjentisani pristup samo iz drugog ugla.

Ako to sada svedemo na probleme u programiranju, kompleksni program se može slojevito raščlaniti i svakom sloju pristupati posebno, kao na primjeru programa za obračun poreza koji smo ranije pominjali. Apstrakcijom smo se fokusirali na bitne stvari (definisali tri ključne klase, unutar njih varijable i metode), a za program nebitne stvari izostavili. Objekti klase `Porez` bi mogli biti Porez na dodatnu vrijednost i Porez na dobit; objekti klase `Sektor` recimo Proizvodnja, Računovodstvo, PravniSektor; objekti klase `ExtOrgan`, na primjer, PoreskaUprava, MinistarstvoFinansija.

Drugim riječima, apstrakcijom smo pomjerili fokus sa toga kako objekat nešto radi (nama nebitan podatak, npr. koji zakon primjenjuje) na to šta radi (bitan podatak npr. obračunava porez).

Klasa Oblik

```
// primjer apstrakcije u Javi
abstract class Oblik {
    String boja;
    // deklarisanje apstraktnih metoda
    abstract double povrsina();
    public abstract String toString();

    // apstraktna klasa moze imati
konstruktor
    public Oblik(String boja)
    {
        System.out.println("Poziv
konstruktora klase Oblik");
        this.boja = boja;
    }
    // 'get' metod
    public String getBoja()
    {
        return boja;
    }
}
```

Klasa Krug

```
class Krug extends Oblik {
    double poluprecnik;

    public Krug(String boja, double
poluprecnik)
    {
        // poziv konstruktora klase Oblik
        super(boja);
        System.out.println("Poziv
konstruktora klase Krug");
        this.poluprecnik = poluprecnik;
    }

    @Override
    double povrsina()
    {
        return Math.PI *
Math.pow(poluprecnik, 2);
    }

    @Override
    public String toString()
    {
        return "Boja kruga je "
                + super.boja
                + " a povrsina iznosi : "
                + povrsina();
    }
}
```

Klasa Pravougaonik

```
class Pravougaonik extends Oblik {

    double duzina;
    double sirina;

    public Pravougaonik(String color,
                        double duzina,
                        double sirina)
    {

        // poziv konstruktora klase Oblik
        super(color);
        System.out.println("Poziv
konstruktora klase Pravougaonik");
        this.duzina = duzina;
        this.sirina = sirina;
    }
```

Klasa Test

```
public class Test {

    public static void main(String[] args) {
        Oblik s1 = new Krug("crvena", 2.2);
        Oblik s2 = new Pravougaonik("zuta",
2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

```
    @Override
    double povrsina()
    {
        return duzina * sirina;
    }

    @Override
    public String toString()
    {
        return "Boja pravougaonika je "
                + super.boja
                + " a povrsina iznosi : "
                + povrsina();
    }
}
```

Tabela 5: Primjer apstrakcije u Javi

Ispis:

```
Poziv konstruktora klase Oblik
Poziv konstruktora klase Krug
Poziv konstruktora klase Oblik
Poziv konstruktora klase Pravougaonik
Boja kruga je crvena a povrsina iznosi : 15.205308443374602
Boja pravougaonika je zuta a povrsina iznosi : 8.0
```

Napomena: Kreirali smo četiri klase, `Oblik`, `Krug`, `Pravougaonik` i `Test`. Klasa `Oblik` je apstraktna[27], što znači da nije moguće stvoriti instancu te klase odnosno novi objekat. U toj klasi je definisana varijabla boja, metodi povrsina i `toString`, konstruktor `Oblik` i `'get'` metod. Apstrakcija se ogleda u tome da pošto nije moguće stvoriti novi objekat te klase, moguće je kreirati drugu klasu koja nasljeđuje apstraktnu klasu, a onda po potrebi pozivati varijable i metode iz apstraktne klase. Na taj način se dizajnom vrši skrivanje podataka i programeru daje sloboda izbora koji dio kôda (varijablu, metod) će koristiti u kojoj klasi.

Na osnovu rečenog, kreirane su dvije klase, `Krug` i `Pravougaonik`, koje nasljeđuju apstraktnu klasu `Oblik`, odnosno njene varijable i metode[28]. Klasa `Krug` definiše varijablu poluprecnik te metod `Krug`. Metod (konstruktor) `Krug` uzima dva argumenta, varijable boja i poluprecnik, putem naredbe `super`[29] preuzima vrijednost boje iz klase `Oblik`. Dalje, komandom `@override` se prepisuje metod povrsina klase `Oblik`, dodjeljuje mu se izraz koji računa površinu kruga, te tu vrijednost vraća. Analogno, drugi override prepisuje `toString` metod koji vraća tekst sa ispisom boje i površine. Klasa `Pravougaonik` radi sve što radi i klasa `Krug`, sa razlikom da, budući da je u pitanju pravougaonik, površina se

[27]Detaljnije o apstraktnoj klasi: https://www.javatpoint.com/abstract-class-in-java
[28]Konstruktori se ne nasljeđuju, kao ni private varijable.
[29]Detaljnije o komandi super: https://www.javatpoint.com/super-keyword

drugačije računa. Konačno, klasa `Test`, kao izvršna klasa, kreira objekte `Krug` i `Pravougaonik` (koristeći polimorfizam!), te poziva metod `toString()` za svaki od njih.

Apstrakcija se ogleda u tome da i klasa `Krug` i klasa `Pravougaonik` koriste isti kôd klase Oblik, dio njegovog kôda direktno preuzimaju (varijabla boja), a dio kôda prepisuju i nanovo deklarišu (metod `toString()`), i izvršavanjem programa dobijaju rezultate koji odgovaraju unijetim podacima (iako se baziraju na programskom kôdu iz klase koja ne može biti instancirana).

6.2 ENKAPSULACIJA (ENCAPSULATION)

Enkapsulacija je mehanizam koji skriva programski kôd od vanjskih uticaja i tako čini određen segment programa sigurnijim. U OOP, objekti unutar klase mogu biti organizovani tako da budu potpuno ili djelimično izolovani od drugih objekata u smislu pristupa i modifikacija. Osnovna jedinica enkapsulacije je klasa. Klasa definiše formu objekta, koji su instance klase. Objekte čini kôd (varijable i metodi), te podaci koji im mogu biti pridruženi. Unutar objekta, kôd, metodi i/ili podaci mogu biti sakriveni (privatni), ili javni (public). Vratimo se na primjer programa za izračun poreza i posmatrajmo klasu `Sektor`. Moguće je da podesimo kontrolu pristupa sadržaju te klase, tako da primjera radi varijabla `odgovornoLice` bude privatna, a metod `rokObrade()` javni. Recimo da smo stvorili novi objekat `Prodaja` koji pripada toj klasi. To bi značilo da je varijabli `odgovornoLice` moguće pristupiti samo unutar objekata koji pripadaju istoj klasi `Sektor`, odnosno da je ta varijabla skrivena za sve druge klase[30]. Analogno tome, javnom metodu `rokObrade()` bi bilo moguće nesmetano izvana kao i iznutra pristupati[31].

```
// primjer enkapsulacije u Javi
public class Enkapsulacija {
    /* deklarisane 'private' varijable
    // kojima je moguce pristupiti samo
    iz iste klase */
    private String ime;
    private int visina;
    private int godine;

    // 'get' metod varijable godine
    public int getGodine()
    {
```

[30]Tako bi se moglo onemogućiti da vrijednost te varijable (npr. "Marko Marković“) bude promijenjen od strane druge klase u "Pero Perić“.

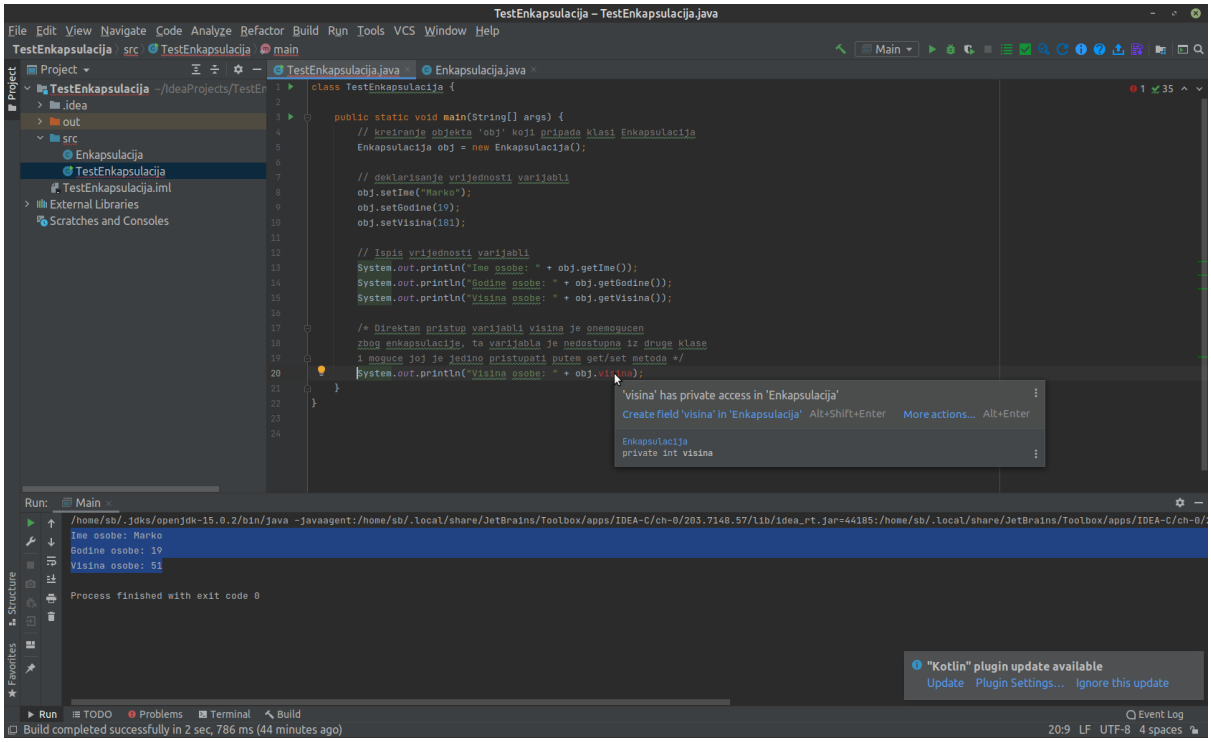
[31]Jer smo, recimo, odlučili da radi funkcionisanja programa ovaj metod mora da bude varijabilan, odnosno da mu je moguće pristupiti i po potrebi dodijeliti novu vrijednost i izvan klase u kojoj je definisan.

```
        return godine;
    }

    // 'get' metod varijable ime
    public String getIme()
    {
        return ime;
    }

    // 'get' metod varijable visina
    public int getVisina()
    {
        return visina;
    }

    // 'set' metod varijable godine
    public void setGodine(int newGodine)
    {
        godine = newGodine;
    }

    // 'set' metod varijable ime
    public void setIme(String newIme)
    {
        ime = newIme;
    }

    // 'set' metod varijable visina
    public void setVisina(int newVisina)
    {
        visina = newVisina;
    }
}

class TestEnkapsulacija {

    public static void main(String[] args) {
        // kreiranje objekta 'obj' koji pripada klasi Enkapsulacija
        Enkapsulacija obj = new Enkapsulacija();

        // deklarisanje vrijednosti varijabli
        obj.setIme("Marko");
        obj.setGodine(19);
        obj.setVisina(181);

        // Ispis vrijednosti varijabli
        System.out.println("Ime osobe: " + obj.getIme());
        System.out.println("Godine osobe: " + obj.getGodine());
        System.out.println("Visina osobe: " + obj.getVisina());

        /* Direktan pristup varijabli visina je onemogucen
          zbog enkapsulacije, ta varijabla je nedostupna iz druge klase i
moguce joj je jedino pristupati putem get/set metoda */
        //System.out.println("Visina osobe: " + obj.visina);
    }
}
```

Ispis programa:

```
Ime osobe: Marko
Godine osobe: 19
Visina osobe: 181
```

Napomena: Kreirali smo klasu `Enkapsulacija`, u njoj deklarisali privatne varijable kao i `get/set` metode. U klasi `TestEnkapsulacija` smo kreirali instancu ove klase, odnosno objekat `obj`, te dodijelili vrijednosti set varijablama i ispisali ih. Ukoliko pokušamo direktno pristupiti jednoj od privatnih varijabli, npr. ako kažemo ispiši `obj.visina`, program pri kompilaciji javlja grešku jer je to privatna varijabla u drugoj klasi kojoj se ne može pristupiti.



Prilog 2: Primjer upozorenja nemogućnosti pristupa privatnoj varijabli, IDE IntelliJ

Međutim očigledno je da se putem get/set može pristupiti privatnim varijablama, kao i dodijeliti im proizvoljna vrijednost, pa se postavlja pitanje u čemu je poenta skrivanja? Odgovor je da set metod omogućava postavljanje logike koja će spriječiti dodjeljivanje vrijednosti koje nam ne bi odgovarale, a ipak omogućiti pristup podacima iz drugih klasa[32]. Na primjer, varijable `godine` i `visina` su po prirodi stvari prirodni brojevi (cijeli brojevi veći od nule). Korištenjem `set` metoda bi mogli ubaciti logiku koja bi isključivala mogućnost da neko unese negativne vrijednosti ovih varijabli (jer je nemoguće imati negativan broj godina i negativnu visinu):

```
// izmijenjeni 'set' metod varijable godine
  public void setGodine(int newGodine)
```

[32]Mogli bismo vrlo lako zaključati čitav kôd neke klase, ali bi time OOP postalo besmisleno. Enkapsulacija omogućava parcijalnu kontrolu pristupa, omogućavajući pristup onim dijelovima kôda za koje je to potrebno, a drugima ograničavajući pristup.

```
{ if (newGodine>=0) {
    godine = newGodine;
} else {
    System.out.println("Greska! Godine ne mogu biti negativne");
    }
}
```

Ukoliko bi sada u klasi `TestEnkapsulacija` pokušali napisati:

```
// dodjela negativne vrijednosti varijabli setGodine
obj.setGodine(-19);
```

dobili bi slijedeći ispis:

```
Greska! Godine ne mogu biti negativne
Ime osobe: Marko
Godine osobe: 0
Visina osobe: 181
```

6.3 APSTRAKCIJA I ENKAPSULACIJA: SLIČNOSTI I RAZLIKE

Kako postoje očigledne sličnosti ali i značajne razlike između apstrakcije i enkapsulacije[33], sumiraćemo ih u idućoj tabeli:

	Apstrakcija	**Enkapsulacija**
1.	Apstrakcija govori o vanjštini objekta, na primjer mobilni telefon ima displej, auto ima točkove i kabinu itd.	Enkapsulacija govori o unutrašnjem sastavu, na primjer kako telefon ili auto funkcionišu, kako su njihovi podsistemi i dijelovi konektovani i šta tačno koji dio radi.
2.	Apstrakcija se koristi za skrivanje neželjenih/nepotrebnih podataka te za pružanje bitnih podataka.	Enkapsulacija znači sakrivanje podataka u određenu klasu radi njihove zaštite od pristupa i izmjene iz vanjskog okruženja (druge klase).
3.	Apstrakcija se fokusira na ono šta objekat radi umjesto onog kako to radi.	Enkapsulacija znači skrivanje internih detalja i mehanizama kako objekat vrši neku radnju.
4.	Apstrakcija rješava problem putem dizajna.	Enkapsulacija rješava problem putem implementacije.
5.	Apstrakcija se implementira korištenjem apstraktnih klasa i interfejsa.	Enkapsulacija se implementira korištenjem pristupnih modifikatora (private, public).

Tabela 6: Razlike između apstrakcije i enkapsulacije

6.4 NASLJEĐIVANJE (INHERITANCE)

U Javi, nasljeđivanje se odnosi na tehniku u OOP koja omogućava kreiranje klasa koje su izvedene iz drugih klasa. Ovo je sama srž OOP programskih jezika, pa je tako Java sastavljena od hijerarhije klasa i njima pripadajućih metoda. Klasa koja nasljeđuje obično se

[33]Ovo pitanje je često na intervjuima prilikom zapošljavanja Java programera. Za više informacija pogledati https://www.tonymarston.co.uk/php-mysql/abstraction.txt

naziva podklasa, a klasa iz koje se nasljeđuje bazna, ili super klasa. Naravno, nasljeđivanje u programskim jezicima nosi analogiju u nasljeđivanju u realnom svijetu, a to je prenos karakteristika i osobina obično sa roditelja na dijete. U tom smislu, podklasa automatski nasljeđuje atribute (odnosno varijable) i ponašanje (metode) od svoje bazne klase[34,35]. Klasa koja nasljeđuje može da deklariše svoje varijable i metode, te, može da mijenja ponašanje naslijeđenih karakteristika od bazne klase.

Na primjeru iz realnog svijeta, možemo kreirati baznu klasu `Vozila`. Klasa `Vozila` može da ima varijable brzina, težina, te metode voziti, zaustaviti se, skretati. Bazna klasa može da ima podklase `Automobili` i `Motocikli`. Te dvije klase nasljeđuju atribute bazne klase, a mogu da imaju i svoje posebne atribute (npr. metod auto parkiranje za automobil), ili da prepišu (`overload`) neke od baznih atributa (npr. Motocikl ima obično 2 točka, automobil pretežno 4).

Nasljeđivanje omogućava smanjenje kôda potrebnog za definisanje više klasa, budući da nove klase jednostavno preuzimaju već postojeći kôd. To je najveća prednost nasljeđivanja, smanjenje kôda i njegova ponovna upotreba[36]. Također se dobija jasniji kôd i postiže veća preglednost programa.

Kako smo već vidjeli, podklasa se u Javi kreira korištenjem komande `extends`.

Za jednostavnu demonstraciju nasljeđivanja koristićemo primjer kreiranja klasa za šahovski program. Za svaku figuru možemo kreirati klasu i definisati u svakoj od njih varijable `x`,`y`, te `vrijednost` (određuju poziciju figure na šahovskoj tabli kao i 'jačinu' figure), te metod za kretanje po tabli. Očigledno je da se kôd višestruko duplira.

Klasa Kralj	Klasa Kraljica	Klasa Pjesak

class Kralj
{
int x;
int y;
int vrijednost;
void kraljKretanje()
{
//kako se kralj krece
}
}
	class Kraljica
	{
	int x;
	int y;
	int vrijednost;
	void kraljicaKretanje()
	{
	//kako se kraljica krece
	}
	}
		class Pjesak
		{
		int x;
		int y;
		int vrijednost;
		void pjesakKretanje()
		{
		//kako se pjesak krece
		}
		}

[34]Java podržava nasljeđivanje od jedne bazne klase, odnosno, nasljeđivanje od više klasa (višestruko nasljeđivanje) nije direktno podržano (moguće je iznaći druge načine, kao npr. korištenje interfejsa).

[35]Vidjeti napomenu 26 za izuzetke od ovog pravila.

[36]Composition je tehnika koja pruža iste benefite, pogledati https://www.journaldev.com/1775/multiple-inheritance-in-java

Klasa Lovac	Klasa Top	Klasa Konj

class Lovac
{
int x;
int y;
int vrijednost;
void lovacKretanje()
{
//kako se lovac krece
}
}
	class Top
	{
	int x;
	int y;
	int vrijednost;
	void topKretanje()
	{
	//kako se top krece
	}
	}
		class Konj
		{
		int x;
		int y;
		int vrijednost;
		void konjKretanje()
		{
		//kako se konj krece
		}
		}

Tabela 7: Primjer višestrukog dupliranja kôda bez upotrebe nasljeđivanja

Koristeći nasljeđivanje, možemo dizajnirati baznu klasu ovako:

```
class baznaKlasaSah
{
int x;
int y;
int vrijednost;
      }
```

te prilagoditi podklase na ovaj način:

Klasa Kralj	Klasa Kraljica	Klasa Pjesak

class Kralj extends baznaKlasaSah
{
void kraljKretanje()
{
//kako se kralj krece
}
}
	class Kraljica extends baznaKlasaSah
	{
	void kraljicaKretanje()
	{
	//kako se kraljica krece
	}
	}
		class Pjesak extends baznaKlasaSah
		{
		void pjesakKretanje()
		{
		//kako se pjesak krece
		}
		}

Klasa Lovac	**Klasa Top**	**Klasa Konj**

class Lovac extends baznaKlasaSah
{
void lovacKretanje()
{
//kako se lovac krece
}
}
	class Top extends baznaKlasaSah
	{
	void topKretanje()
	{
	//kako se top krece
	}
	}
		class Konj extends baznaKlasaSah
		{
		void konjKretanje()
		{
		//kako se konj krece
		}
		}

Tabela 8: Primjer višestrukog iskorištavanja postojećeg kôda putem nasljeđivanja

Budući da podklase nasljeđuju od bazne klase, evidentno je da smo smanjili ponavljanje kôda jer smo na jednom mjestu deklarisali varijable `x`,`y` i `vrijednost`, a u podklasama smo definisali samo metod kretanja koji se mora definisati za svaku od podklasa ponaosob.

6.5 POLIMORFIZAM (POLYMORPHISM)

Polimorfizam je riječ koja u grčkom jeziku znači više formi, oblika. Jednostavan primjer polimorfizma iz realnog života bi bio funkcionisanje upravljača (volana) u automobilu. Bez

obzira kakav se koristi mehanizam upravljanja, on mora imati volan koji je isti za sve automobile. Dakle auto može da koristi različite tehnologije (servo upravljanje, manuelno upravljanje), upravljač će uvijek biti isti.

U programiranju, polimorfizam omogućava manipulaciju sa više objekata kao da su istog tipa. Pokažimo na primjeru o čemu se tu zapravo radi.

```
public class Zivotinja {

   public void govor() {

       System.out.println("Zdravo!");
   }
}

public class Pas extends Zivotinja {

   @Override
   public void govor() {
       System.out.println ("Vau-vau!");
   }
}

public class Macka extends Zivotinja {

   @Override
   public void govor() {
       System.out.println("Mjau!");
   }
}
```

Kreirali smo baznu klasu `Zivotinja` sa metodom `govor()`, te dvije podklase, `Pas` i `Macka`, koje prepisuju (`override`) metod `govor()` sa svojim prilagođenim vrijednostima.

Kreirajmo sada izvršni `main()` metod:

```
public class Main {

   public static void main(String[] args) {

       Zivotinja pas = new Pas();
       pas.govor();
   }
}
```

Ispis programa: `Vau-vau!`

Linijom kôda

```
Zivotinja pas = new Pas();
```

poziva se bazna klasa, kreira se novi objekat i dodjeljuju mu se atributi podklase `Pas`. Objekat `Pas` će koristiti metod definisan iz klase `Pas`, a ne iz klase `Zivotinja`, iako je kreiran kao instanca klase `Zivotinja`. Dakle, kreirali smo instancu klase `Zivotinja`,

koja koristi atribute druge, podklase `Pas`, odnosno uzima više formi. To je polimorfizam na djelu. Naravno, mogli smo kreirati novi objekat i ovako:

```
Pas pas = new Pas();
```

međutim ovako ne bi iskoristili polimorfizam.

Analogna stvar bi se desila i da smo kreirali objekat klase `Macka`:

```
public class Main {

   public static void main(String[] args) {

       Zivotinja macka = new Macka();
       macka.govor();
   }
}
```

Ispis programa: `Mjau!`

Da bi dalje pokazali ranije izrečenu tvrdnju 'polimorfizam omogućava manipulaciju sa više objekata kao da su istog tipa', pretpostavimo u primjeru dalje da je životinjama potreban veterinarski pregled. Stoga, kreirajmo novu klasu:

```
public class Veterinar {

   public void pregled(Zivotinja zivotinja) {

       System.out.println("Pregled je zavrsen!");
   }
}
```

U klasi `Veterinar` je deklarisan metod `pregled()`, kojeg zatim možemo proslijediti objektima klasa `Pas` i `Macka`:

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

   Macka macka = new Macka();
   Pas macka = new Pas();

   Veterinar veterinar = new Veterinar();

   veterinar.pregled(macka);
   veterinar.pregled(pas);
}
```

Kreirali smo instance klasa `Macka` i `Pas`, te `Veterinar`. Metod klase `Veterinar`, `pregled()`, funkcioniše sa objektima drugih klasa kao da su istog tipa, dok istovremeno, objekti macka i pas imaju različite atribute, metod `govor()` daje različite rezultate kako smo ranije vidjeli. To je primjer polimorfizma koji tretira objekte po tipu (za razliku od recimo nasljeđivanja koji tretira objekte po pripadnosti), omogućavajući njihovu manipulaciju kao da su istovjetni, iako pripadaju raznim tipovima (klasama).

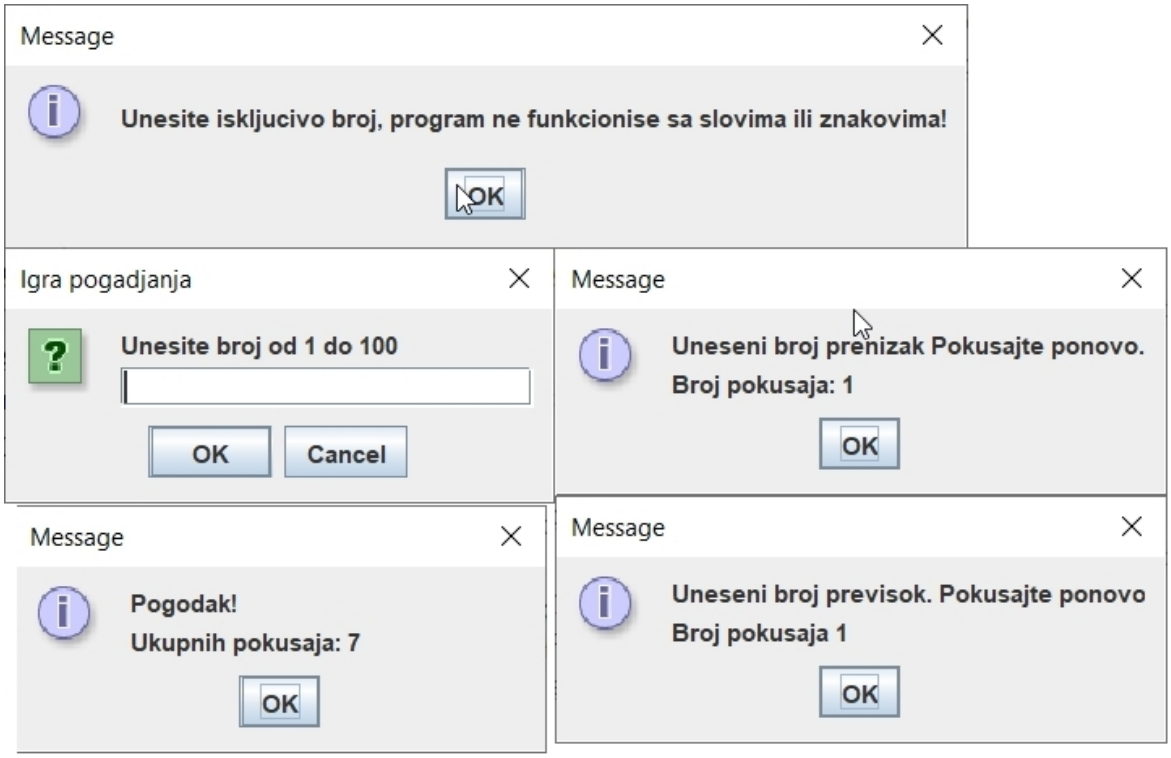
7 MINI PROJEKTI

7.1 Igra pogađanja

Ovaj jednostavni program se sastoji iz generisanja slučajnog broja, te unosa korisnika sa ciljem da se pogodi koji je broj generisan.

```
// Java program "Igra pogadjanja"

import javax.swing.*;

public class IgraPogadjanja {
    public static void main(String[] args) {
        int brojZaPogadjanje = (int) (Math.random()*100 + 1);
        int inputKorisnika = 0;
        // opciono prikazi generisani broj
        // System.out.println("Izabrani rezultat je " + brojZaPogadjanje);
        int brojPogadjanja = 1;


        try {
        while (inputKorisnika != brojZaPogadjanje)
        {
            String odgovor = JOptionPane.showInputDialog(null,
                    "Unesite broj od 1 do 100", "Igra pogadjanja", 3);
            inputKorisnika = Integer.parseInt(odgovor);
      JOptionPane.showMessageDialog(null, ""+
      odrediKorisnikovUnos(inputKorisnika, brojZaPogadjanje,
      brojPogadjanja));
            brojPogadjanja++;
            }
        }
        catch (NumberFormatException e) {
                String greskaPoruka="Unesite iskljucivo broj, program ne
funkcionise sa slovima ili znakovima!";
            JOptionPane.showMessageDialog(null, ""+ greskaPoruka) ;
        }
    }

      public static String odrediKorisnikovUnos(int odgovorKorisnika, int
generisaniBroj, int brojac){
        if (odgovorKorisnika <=0 || odgovorKorisnika >100) {
            return "Uneseni broj izvan zadatih granica";
        }
        else if (odgovorKorisnika == generisaniBroj ){
            return "Pogodak!\nUkupnih pokusaja: " + brojac;
        }
        else if (odgovorKorisnika > generisaniBroj) {
             return "Uneseni broj previsok. Pokusajte ponovo\nBroj pokusaja
" + brojac;
        }
        else if (odgovorKorisnika < generisaniBroj) {
            return "Uneseni broj prenizak Pokusajte ponovo.\nBroj pokusaja:
" + brojac;
        }
        else {
            return "Netacno!\nBroj pokusaja " + brojac;
        }
    }
}
```

Ispis programa:



Napomena: Ovaj program koristi grafički interfejs SWING API klase za ispis prozora za unos pogađanja. Pozivom `Math.random` metoda se generiše slučajan broj od 0-100 koji korisnik treba da pogodi. Evidentira se i zbraja broj pokušanih pogađanja. Na osnovu logičkog iskaza koji je uvijek istinit, "while“ petlja iteriše unos korisnika sve dok se ne pogodi generisani broj. Metod `odrediKorisnikovUnos` sadrži logiku koja poredi uneseni broj sa generisanim brojem i u skladu sa tim vraća određenu poruku. Glavni kôd programa je smješten u try blok, koji se pokušava izvršiti pod pretpostavkom da neće doći do neke greške. Ako do greške dođe, a ta greška se odnosi na to da korisnik unese neki drugi karakter osim broja, catch blok hvata tu grešku u Javi definisanu kao `NumberFormatException`, završava program obavještavajući korisnika prigodnom porukom da mora unijeti isključivo brojeve.

7.2 "IKS-OKS“

U pitanju je klasična igra "Iks-oks", izvedena u Javi unosom i ispisom iz konzole.

```
// Java program "Iks-oks"
import java.util.*;

public class IksOks {

    static ArrayList<Integer> pozicijeKorisnika =new ArrayList<Integer>();
    static ArrayList<Integer> pozicijeRacunara =new ArrayList<Integer>();

    public static void main(String[] args) {
        char [][] tabla={
                {' ', '|', ' ', '|', ' '},
                {'-', '+', '-', '+', '-'},
                {' ', '|', ' ', '|', ' '},
                {'-', '+', '-', '+', '-'},
                {' ', '|', ' ', '|', ' '}};

        ispisiTablu(tabla);
        try {
        while (true) {
            Scanner scan= new Scanner(System.in);
        System.out.println("Unesite poziciju (1-9):");
        int pozicijaKorisnika=scan.nextInt();
                    while(pozicijeKorisnika.contains(pozicijaKorisnika)||
pozicijeRacunara.contains(pozicijeKorisnika)) {
            System.out.println("Pozicija zauzeta! Unesite ponovo!");
            pozicijaKorisnika=scan.nextInt();
        }
        postaviUnos(tabla, pozicijaKorisnika, "korisnik");
            String rezultat= provjeriPobjednika();
            if(rezultat.length()>0) {
                System.out.println(rezultat);
                break;
            }


        Random rand=new Random();
        int pozicijaRacunara=rand.nextInt(9)+1;
                     while(pozicijeKorisnika.contains(pozicijaRacunara)||
pozicijeRacunara.contains(pozicijeRacunara)) {
                pozicijaRacunara=rand.nextInt(9)+1;
            }
        postaviUnos(tabla, pozicijaRacunara, "racunar");
        ispisiTablu(tabla);

        rezultat= provjeriPobjednika();
        if(rezultat.length()>0) {
            System.out.println(rezultat);
            break;
            }
        }
    }
    catch (InputMismatchException e) {
        System.out.println("Molimo unesite iskljucivo broj!");
        }
    }
```

```
    public static void ispisiTablu(char[][] tabla) {
        for(char[] red:tabla){
            for(char c:red) {
                System.out.print(c);
            }
            System.out.println();
        }
    }

      public static void postaviUnos(char[][] tabla, int pozicija, String
korisnik) {

        char karakter=' ';
        if (korisnik.equals("korisnik")) {
            karakter = 'X';
            pozicijeKorisnika.add(pozicija);
        } else if(korisnik.equals("racunar")) {
            karakter='O';
            pozicijeRacunara.add(pozicija);
        }

        switch (pozicija) {
            case 1:
                tabla[0][0]=karakter;
                break;
            case 2:
                tabla[0][2]=karakter;
                break;
            case 3:
                tabla[0][4]=karakter;
                break;
            case 4:
                tabla[2][0]=karakter;
                break;
            case 5:
                tabla[2][2]=karakter;
                break;
            case 6:
                tabla[2][4]=karakter;
                break;
            case 7:
                tabla[4][0]=karakter;
                break;
            case 8:
                tabla[4][2]=karakter;
                break;
            case 9:
                tabla[4][4]=karakter;
                break;
            default:
                break;
        }
    }

    public static String provjeriPobjednika() {
        List gorRed= Arrays.asList(1,2,3);
        List sredRed= Arrays.asList(4,5,6);
        List donRed= Arrays.asList(7,8,9);
        List lKol= Arrays.asList(1,4,7);
        List sKol= Arrays.asList(2,5,8);
```

```
        List dKol= Arrays.asList(3,6,9);
        List d1= Arrays.asList(1,5,8);
        List d2= Arrays.asList(7,5,3);

        ArrayList<List> pobjeda=new ArrayList<List>();
        pobjeda.add(gorRed);
        pobjeda.add(sredRed);
        pobjeda.add(donRed);
        pobjeda.add(lKol);
        pobjeda.add(sKol);
        pobjeda.add(dKol);
        pobjeda.add(d1);
        pobjeda.add(d2);

        for(List l:pobjeda) {
            if(pozicijeKorisnika.containsAll(l)) {
                return "Pobijedili ste, cestitka!";
            } else if(pozicijeRacunara.containsAll(l)) {
                return "Racunar je pobjedio!";
                                    }   else   if   (pozicijeKorisnika.size()+
pozicijeRacunara.size()==9) {
                return "Nerijeseno!";
            }
        }
        return "";
    }
}
```

Ispis programa:

```
Unesite poziciju (1-9):
1
X| |
-+-+-
 | |
-+-+-
 |0|
Unesite poziciju (1-9):
2
X|X|
-+-+-
 |0|
-+-+-
 |0|
Unesite poziciju (1-9):
3
Pobijedili ste, cestitka!
```

```
Unesite poziciju (1-9):
1
X| |
-+-+-
0| |
-+-+-
 | |
Unesite poziciju (1-9):
6
X| |
-+-+-
0| |X
-+-+-
 |0|
Unesite poziciju (1-9):
7
X| |
-+-+-
0|0|X
-+-+-
X|0|
Unesite poziciju (1-9):
9
X|0|
-+-+-
0|0|X
-+-+-
X|0|X
Racunar je pobjedio!
```

```
Unesite poziciju (1-9):
1
X| |
-+-+-
 |0|
-+-+-
 | |
Unesite poziciju (1-9):
3
X| |X
-+-+-
 |0|0
-+-+-
 | |
Unesite poziciju (1-9):
4
X|0|X
-+-+-
X|0|0
-+-+-
 | |
Unesite poziciju (1-9):
8
X|0|X
-+-+-
X|0|0
-+-+-
 |X|0
Unesite poziciju (1-9):
9
Nerijeseno!
```

Napomena:

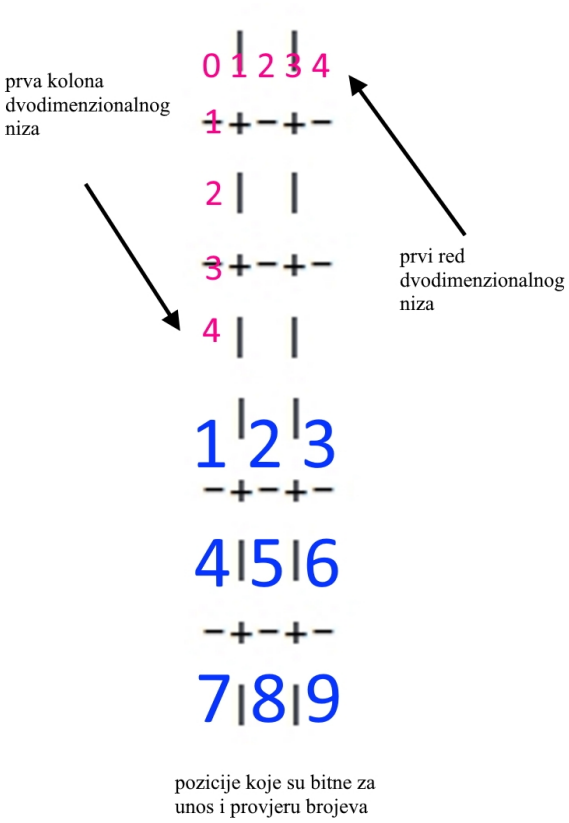
Ova igra se sastoji od table na kojoj dva korisnika unose pozicije. Cilj igre je ispuniti u nizu horizontalno, vertikalno ili dijagonalno navedena polja prije nego što to uradi drugi igrač.

Program u main metodu deklariše dvodimenzionalni niz tabla koji će služiti kao ispis table u konzoli. U tu svrhu kreiran je metod `ispisiTablu` koji dvostrukom petljom iteriše

navedeni niz. Iako se program mogao napraviti i bez ovih metoda, njihovim korištenjem `main` metod je mnogo jasniji i kraći. Kao i u prethodnom primjeru, izvršni kôd je obuhvaćen try/check petljom, koja služi da se uhvate eventualne greške nastale neodgovarajućim korisničkim unosom.

Za korisnički unos se koristi klasa `Scanner`.

Budući da se tabla sastoji iz tri reda i tri kolone (zapravo polja koja nas interesuju se sastoje od 3*3 table), nas interesuju pozicije niza `tabla` od 1-9. Metod `postaviUnos` sadrži logiku koja smješta korisnički unos u tabelu. U tu svrhu koristi se `switch` koji ima 9 slučajeva, svaki od tih slučajeva odgovara poljima 1-9 niza `tabla`. Na primjer, ukoliko korisnik unese broj 5, taj broj odgovara drugom redu, drugoj koloni niza:




```
case 5:
      tabla[2][2]=karakter;
            break;
```

Za unos koristimo "while" petlju koja je uvijek tačna, budući da nam treba neprekidni unos sve dok se tabla ne popuni, odnosno igra ne završi jednim od ishoda.

Metod `provjeriPobjednika` provjerava da li je neki od korisnika ispunio uslove igre za pobjedu. U tu svrhu, budući da igra ima 8 pobjedničkih ishoda (tri horizontalna, tri vertikalna i dva dijagonalna), možemo kreirati 8 lista, primjera radi ova linija:

```
List gorRed= Arrays.asList(1,2,3);
```

kreira listu iz već definisanog niza uzimajući pozicije `1,2,3` kao vrijednosti.

Budući da imamo dva igrača, korisnika i računarsku logiku, trebamo kreirati unos i za računar, te smjestiti odnosno zapamtiti pozicije gdje je koji korisnik igrao, da se te pozicije ne bi ponavljale. U tu svrhu, unutar već postojeće "while" petlje treba nam još jedna pod-petlja, koja će izbjeći dupliranje pozicija, odnosno provjeravati nakon unosa u slučaju korisnika

(odnosno nasumično izabranog broja u slučaju računara), da li je navedena pozicija već korištena. Ako nije, program će nastaviti sa daljim izvršavanjem, ako nije, ispisaće se prigodna poruka koja će u konzoli vratiti korisnika na ponovni unos druge pozicije. Već birane pozicije ćemo skladištiti u dvjema ArrayList, `pozicijeKorisnika` i `pozicijeRacunara`. Obadva ova objekta su kreirana kao `static`, iz razloga lakšeg pristupanja (ako je objekat `static`, svi metodi u klasi mu mogu pristupati bez potrebe da svaki put se pravi novi objekat u svakom metodu). U metodu `postaviUnos` ćemo dodati logiku koja skladišti korisnički input u jednu od ovih lista (za korisnika to je `pozicijeKorisnika`, za računar to je `pozicijeRacunara`).

Vratimo se natrag u metod `provjeriPobjednika`, u kome se nalazi 8 lista u kojima je zabilježen rezultat koji vodi do pobjede u igri. Budući da treba da provjerimo svaki ovaj mogući ishod da bi verifikovali da li je neki od korisnika pobijedio, to bi značilo puno manuelnog rada budući da bi morali iterisati svaku od tih lista. Zato je moguće dodati sve te liste u jednu listu nazvanu pobjeda i nad njom izvršiti iteraciju:

```
ArrayList<List> pobjeda=new ArrayList<List>();
```
ili
```
ArrayList<List> pobjeda=new ArrayList<List>();
```

Možemo deklarisati ArrayList kao objekat tipa List jer su oni povezani, List obuhvata klasu ArrayList.

Potom, iterišemo stvorenu listu `winning` provjeravajući da li u njoj imamo neki od pobjedničkih uslova, npr. linija:

```
if(pozicijeKorisnika.containsAll(l))
```

provjerava za svaku iteraciju, da li u listi `playerPositions` ima bilo koji od pobjedničkih uslova (metod `containsAll` to omogućava), odnosno da li je korisnik unio bilo koju od ovih kombinacija:

```
List gorRed= Arrays.asList(1,2,3);
List sredRed= Arrays.asList(4,5,6);
List donRed= Arrays.asList(7,8,9);
List lKol= Arrays.asList(1,4,7);
List sKol= Arrays.asList(2,5,8);
List dKol= Arrays.asList(3,6,9);
List d1= Arrays.asList(1,5,8);
List d2= Arrays.asList(7,5,3);
```

ZAKLJUČAK

U ovom radu smo predstavili neke od osnovnih tehnika programiranja u programskom jeziku Java. Petlje su neizostavni dio svakog programa i kao takve njihovo razumijevanje je esencijalno za rješavanje bilo kog programskog problema. Strukture kao što su nizovi, liste i mape takođe spadaju u osnovne alate kojima se programeri služe kako bi na što efikasniji način pojednostavili problem koji rješavaju, jer se njima može manipulisati pomoću uniformnih metoda ugrađenih u programski jezik.

Kako je Java objektno-orjentisana, u radu je pružen osnovni uvid u najznačajnije elemente ove programske paradigme i pokušali smo dati objašnjenja elementarnih principa OOP. Naravno, Java je širok i kompleksan programski jezik koji ostavlja programeru na volju izbor tehnika koje će koristiti u rješavanju datog problema.
Jave je, kao i C i C++ sintaksno 'težak' programski jezik, nemoguće je memorisati svaku komandu, već će programer u svom radu biti oslonjen na iščitavanje dokumentacije, korištenje drugih resursa kao što su knjige, naučni radovi i publikacije, te internet forumi i stranice. Upravo ova kolaboracija, koju je Internet omogućio je i dovela Javu do tržišnog uspjeha koji je evidentan nakon više od 25 godina od pojave ovog programskog jezika. Danas se od većine programera koji se bave opštim programskim jezicima očekuje da poznaju neki od jezika iz C-sintaksne grupe, a vjerovatno je najpoželjnije poznavanje Jave, kao univerzalnog jezika primjenjivog praktično u svim segmentima računarske tehnologije.

Po svim statistikama, Java je prvi ili drugi najtraženiji programski jezik po broju pretraga u Google pretraživaču, što je dovoljan pokazatelj aktuelnosti, atraktivnosti i traženosti znanja u ovom programskom jeziku. Znanja i iskustvo stečeno u programiranju u Javi će se sigurno moći iskoristiti na tržištu u dugom nizu godina, obzirom na ogromnu bazu softvera koja je nastala u zadnjih 25 godina.

LITERATURA

[1] Core Java, Volume I-Fundamentals, Eleventh Edition, Cay. S. Horstmann, Pearson, 2019

[2] Java A Begginer's Guide, Eight Edition, Herbert Schildt, McGraw-Hill, 2019

[3] Java All-In-One, 11th Edition, Doug Lowe, John Wiley&Sons, 2020

[4] Thinking in Java, Fourth Edition, Bruce Eckel, Prentice Hall 2006

[5] https://docs.oracle.com/javase/specs/jls/se15/html/index.html, The Java Language Specification, Java SE 15, pristupano 06.04.2021

[6] https://www.sciencedirect.com/science/article/pii/S0890540113000795, Object-oriented programming: Some history, and challenges for the next fifty years, pristupano 06.04.2021

[7] https://www.javatpoint.com/abstract-class-in-java, pristupano 06.04.2021

[8] https://www.javatpoint.com/super-keyword, pristupano 06.04.2021

[9] https://www.tonymarston.co.uk/php-mysql/abstraction.txt, Abstraction, Encapsulation, and Information Hiding, pristupano 06.04.2021

[10] https://www.journaldev.com/1775/multiple-inheritance-in-java, pristupano 06.04.2021