

# Podrška objektno orijentisanom programiranju u jezicima C++, Objective C, Java, C#, Ada i Ruby

Seminarski rad u okviru kursa  
Metodologija stručnog i naučnog rada  
Matematički fakultet

Katarina Popović, Dušan Pantelić, Dejan Bokić, Nikola Stojević  
ail7210@alas.matf.bg.ac.rs, pantelic.dusan@protonmail.com,  
mrdejan995@gmail.com, nikolastojevic@gmail.com

30. april 2019

## Sažetak

Kroz ovaj seminarski rad čitalac će biti upućen u objektno-orijentisano programiranje. Pre svega, kakvu i na koji način mu podršku pružaju programski jezici C++, Objective C, Java, C#, Ada i Ruby. Predstavljene su karakteristike i principi objektno-orijentisane paradigme, koji su najviše zaslužni za stečenu popularnost. Posebno je stavljen naglasak na enkapsulaciju, nasleđivanje, polimorfizam i apstrakciju. Objašnjeno je kako se ovi principi ostvaruju u navedenim programskim jezicima, naravno sa konkretnim primerima kako to izgleda u kodu.

## Sadržaj

|          |                    |           |
|----------|--------------------|-----------|
| <b>1</b> | <b>Uvod</b>        | <b>2</b>  |
| <b>2</b> | <b>C++</b>         | <b>3</b>  |
| <b>3</b> | <b>Objective C</b> | <b>4</b>  |
| <b>4</b> | <b>Java</b>        | <b>6</b>  |
| <b>5</b> | <b>C#</b>          | <b>7</b>  |
| <b>6</b> | <b>Ada</b>         | <b>9</b>  |
| <b>7</b> | <b>Ruby</b>        | <b>10</b> |
| <b>8</b> | <b>Zaključak</b>   | <b>12</b> |
|          | <b>Literatura</b>  | <b>12</b> |

# 1 Uvod

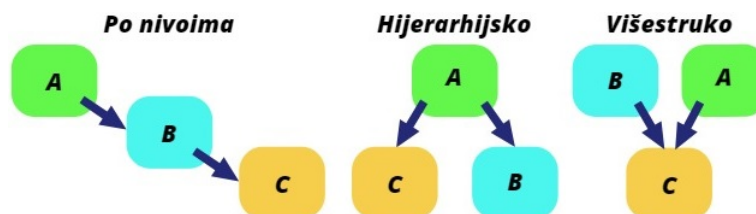
**Programska paradigma** je osnovni stil programiranja, način klasifikovanja programskih jezika na osnovu njihovih karakteristika. Programski jezici (u daljem tekstu jezici) često mogu da se klasifikuju u više od jedne paradigme.

Najčešće programske paradigme su: proceduralna (fokus stavlja na promenljive), funkcionalna (fokus na matematičkim funkcijama), logička (fokus na logičkim izrazima) i objektno-orijentisana paradigma (fokus je na **objektima**). Objekti imaju svoje interno stanje (atribute) i svoje ponašanje (metode). I metode i atributi mogu a ne moraju biti vidljivi spoljašnjem svetu. Srž objektno-orijentisanog programiranja je u slanju poruka i odgovaranju na njih. Ovu paradigmu najbolje opisuje tzv. *Tell dont ask* princip-

Nemoj da tražiš informaciju koja ti treba da bi uradio posao. Umesto toga, pitaj objekat koji ima tu informaciju da odradi taj posao za tebe. (*Allen Hollub- Hollub on Patterns*)

Osnovni principi objektno-orijentisane paradigme:

1. **Princip jedinstvene odgovornosti**- Svaka klasa treba da ima samo jedan zadatak. Čim klasa ima više od jednog zadatka, verovatno bi mogla da se podeli u 2 klase.
2. **Encapsulacija**- i *Tell dont ask* princip su usko vezani. Vidljivost podataka je osnovni način enkapsulacije. U većini programskih jezika, postoje 3 osnovna nivoa vidljivosti- public, protected i private.
3. **Nasledivanje**- podrazumeva da jedna klasa "naslediše metode i attribute svoje nadklase, sem kad su metode i atributi proglašeni privatnim. Uspostavlja odnos "jeste"između klasa- ako klasa A nasleđuje klasu B, kaže se da A "jeste"B. Klasa B se naziva nadklasom klase A. Vrste nasleđivanja prikazane su na slici 1.
4. **Polimorfizam**- označava da klase koje implementiraju isti interfejs mogu da rade različite stvari, dok god se drže principa tog interfejsa.
5. **Apstrakcija**- je termin koji se veže za *interfejse* i *apstraktne klase*. **Interfejsi** se koriste za opisivanje ponašanja klase, definišu javni API (Application programming interface) klase, odnosno metode koje klasa mora da implementira da bi program radio. Oni ne mogu biti instancirani, mogu imati samo potpise metoda, bez tela, ne mogu imati attribute i sve metode moraju biti javne. **Apstraktne klase** su slične interfejsima, ali za razliku od njih metode apstraktnih klasa mogu da imaju telo, mogu da imaju attribute i mogu imati sva moguća prava pristupa (public, protected, private).



Slika 1: Vrste nasleđivanja

## 2 C++

Jezici koji podržavaju sve 4 funkcionalnosti objektno orijentisane paradigme ali ne u potpunosti se obično nazivaju delimično objektno orijentisanim jezicima. Zbog sledećih karakteristika, C++ se smatra delimično objektno orijentisanim jezikom.

1. **Main funkcija je izvan klase :** U C++ može da se napiše validan, ispravan kod bez kreiranja ijednog objekta. Main funkcija je obavezna ali se ona nalazi izvan svake klase, što nije karakteristično za druge OOP jezike.
2. **Koncept globalne promenljive:** U C++ može da se kreira globalna promenljiva koja je dostupna svugde u kodu, dok u drugim OOP jezicima promenljive mogu biti deklarisanе samo u okviru klase gde mogu da se koriste modifikatori pristupa (private, protected, public).
3. **Postojanje friend funkcija:** Friend (prijateljska) funkcija može pristupiti privatnim poljima klase kojoj je deklarisanа kao prijateljska. Ovo je jedna veoma korisna karakteristika C++, ali i dalje narušava neke koncepte objektno orijentisane paradigme.

### 2.1 Enkapsulacija

U C++ ne mora eksplicitno za svaki atribut ili metod klase da bude nagašeno pravo pristupa, nego se prave sekcije, i na početku sekcije se stavi modifikator pristupa (podržani su private, public i protected, ne podržava package modifikator). Ukoliko se modifikator ne navede eksplicitno, kod klase se podrazumeva private, dok kod struktura se podrazumeva public (što je jedna od osnovnih razlika između struktura i klasa u C++).

```
1000 class Employee {
      private:
1002     int salary;
      public:
1004     Employee(int e_salary) { salary = e_salary;}
      int getSalary(){ return salary;}
1006     void setSalary(int newSalary) { salary = newSalary;}
      void display() {
1008         std::cout << "Hello I'm the employee!" << std::endl;
      }
1010 };
```

Listing 1: Primer deklarisanja klase sa enkapsulacijom

### 2.2 Nasleđivanje

U jeziku C++ i nasleđivanje može biti private, protected ili public. Ukoliko je nasleđivanje public, to znači da sva nasleđena polja ostaju javna, ukoliko je protected, tada će sva public polja postati protected, a ukoliko je nasleđivanje private, to znači da će sva public i protected polja postati private. Takođe, za razliku od drugih OOP jezika, u C++ je podržano i višestruko nasleđivanje, gde jedna klasa može da nasledi više od jedne klase. Zbog problema koje višestruko nasleđivanje može da uvede, u C++ je uvedena još jedna ključna reč prilikom nasleđivanja-vritual, koja sprečava tzv. dijamant strukturu.

```
1000 class Driver: public Employee {
      private:
```

```

1002     std::string truck = "FAP";
public:
1004     Driver(int salary, std::string truck)
        : Employee(salary), truck(truck)
1006     {};
        void display() {
1008         std::cout << "My truck is " << truck << "!" << std::endl; };
    };

```

Listing 2: Primer nasleđivanja klasa u C++

## 2.3 Polimorfizam

U C++ su podržana dva osnovna tipa polimorfizma- **polimorfizam u vreme kompilacije** i **polimorfizam u vreme izvršavanja**. Prvi obezbeđuje preopterećenje metoda i operatora. Preopterećenje operatora je isto jedna od C++ specifičnih mogućnosti, gde možemo sami da definišemo ponašanje operatora npr "+đok god ispunjava svoje osnovne karakteristike (da ima 2 argumenta). Drugi tip polimorfizma omogućuje premošćavanje metoda, tj. za metod se kaže da je premošten ukoliko izvedena klasa poseduje metod sa identičnim potpisom.

## 2.4 Apstrakcija

Za razliku od drugih OOP jezika, C++ ne poseduje ključnu rec *abstract*. Apstraktne klase se u C++ kreiraju tako što se napravi virtuelna metoda i u potpisu joj se dodeli 0. Takva klasa ne može biti instancirana, ali može biti napravljen pokazivač na nju. Takođe, apstraktna klasa može da ima konstruktor i destruktor. Klasa koje ne premosti virtuelnu metodu takođe postaje apstraktna klasa. Postoji nekoliko pravila koja moraju da važe za apstraktne klase:

1. Moraju biti proglašene javnim (inače potklasa ne može da ih premosti).
2. Virtuelne metode ne mogu biti static i ne mogu biti prijateljske metode neke druge klase.
3. Virtuelnim metodama se mora pristupati preko pokazivača na baznu klasu da bi se dobio pravi polimorfizam u vreme izvršavanja.
4. Potpis virtuelne metode mora biti identičan i u baznoj i izvedenoj klasi (povratna vrednost, tipovi argumenata, konstantnost argumenata,...).
5. Klase mogu imati virtuelni destruktor, ali ne mogu da imaju virtuelni konstruktor.

## 3 Objective C

Proces definisanja klasa se malo razlikuje kod jezika Objective C. Obavlja se u dve sekcije koje se označavaju sa ključnom reči **@interface** i **@implementation**, gde se vrši deklaracija i implementacija metoda klase. Obe sekcije se završavaju ključnom reči **@end**. Svaka klasa nasleđuje baznu klasu NSObject, koja sadrži konstruktor init. Ovaj konstruktor se može predefinisati. Moguće je kreirati i svoje konstruktore sa proizvoljnim argumentima, slično ostalim programskim jezicima i mogu se proizvoljno imenovati. Opširnije o ovome u dokumentaciji [2]. U primeru (3) vidimo definisanje klase zaposleni (detaljnije 3.1).

```

1000 @interface Employee : NSObject {
        double salary; @public int age;
1002 }
        @property(nonatomic, readwrite) double salary;
1004 - (void)display;
        @end
1006 # '-' za metode instance, '+' za klasne metode(static)
        @implementation Employee
1008 @synthesize salary;
        - (void)display { NSLog(@"Employee salary is %f", salary); }
1010 @end
        # (id) tip koji je kompaktilan svakom objektu
1012 @interface Driver : Employee {
        NSString* truck;}
1014 - (id)initWithTruck:(NSString*)model;
        @end
1016 # self oznacava tekuci objekat
        @implementation Driver
1018 - (id)initWithTruck:(NSString*)model {
        truck = model; return self;
1020 }
        - (void)display { NSLog(@"Driver salary is %f", salary); }
1022 @end
        int main(int argc, const char * argv[]) {
1024     NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
        Employee *empl = [[Driver alloc] initWithTruck:@"Mercedes"];
1026     empl.salary = 5.0;    empl->age = 33;
        [empl display];
1028     [pool drain];
        return 0;
1030 }}

```

Listing 3: Primer koda u Objective C jeziku

### 3.1 Enkapsulacija

Atributi klase su automatski privatni, što pogoduje enkapsulaciji. Njima se omogućuje pristup izvan klase preko ključne reči **@property**, uz implementaciju pristupnih metoda. Moguće automatsko generisanje metoda pristupa, navođenjem atributa uz ključnu reč **@synthesize**. Primer(3), main funkcija sadrži pozive pristupnim metodama pomoću operatora (**.**), dok se javnim atributima pristupa preko operatora (**->**).

### 3.2 Nasleđivanje

Ovaj koncept označavamo sa (**:**) i imenom klase koju nasleđujemo. Primer (3), klasa vozač nasleđuje klasu zaposleni. Postoji nasleđivanje u više nivoa i hijerarhijsko. Ako želimo iz izvedene klase da zovemo metode bazne, to radimo referisanjem na baznu klasu pomoću ključne reči **super**.

### 3.3 Polimorfizam

Višestruka upotrebljivost koda u vidu preopterećenosti (eng. *overloading*) nije omogućeno u Objective C jeziku, tako da se metodi moraju različito imenovati[4]. Koncept preklapanja metoda (eng. *overriding*) postoji, i prikazan je u primeru (3). Klase zaposleni i vozač imaju isti metod display i poziv [empl display] će izvršiti metod klase vozač, zato što promenljiva zaposleni empl referiše na objekat tipa vozač.

### 3.4 Apstrakcija

Jezik Objective C nema definisan koncept apstraktnih klasa [4]. Sličan efekat je moguće postići programerskom snalažljivošću i ne instancirati klasu koja bi trebalo biti apstraktna. Za interfejsse koje ovde nazivamo protokoli, vezana je ključna reč **@protocol** jer je **interface** rezervisana za klase. Sekcija protokola se završava sa **@end** i može sadžati dve podoblasti **@required**, za metode koji se obavezno moraju implementirati u klasi i **@optional**, za metode čija je implementacija opcionalna.

## 4 Java

Kako su klase u centru zbivanja krenućemo od njih. Definišemo ih pomoću ključne reči **class**. Sledeći primer (4) prikazuje kreiranje klase zaposleni, koja ima svoje atribute i metode (detaljnije 4.1). Objekte klase instanciramo pomoću metoda konstruktora, koji nema povratni tip i uvek se zove isto kao i klasa. Ako ne definišemo konstruktor, automatski se generiše podrazumevani konstruktor [7], koji inicijalizuje objekat na podrazumevane vrednosti.

```
1000 public class Employee {  
    private int salary;  
1002 #this je referenca na tekuci objekat  
    public Employee(int salary) { this.salary = salary;}  
1004 public int getSalary(){ return salary;}  
    public void setSalary(int newSalary) { salary = newSalary;}  
1006 public void display() {  
        System.out.println("Hello i'm employee!");  
1008 }  
    public static void main(String[] args) {  
1010        Employee Marko = new Driver(600, "Mercedes");  
        Marko.display();  
1012 }  
class Driver extends Employee {  
1014    String truck = "FAP";  
    #super vrsi poziv konstruktora bazne klase  
1016    public Driver(int salary, String truck) {  
        super(salary); this.truck = truck;}  
1018    public void display() {  
        System.out.println("My truck is "+truck+"!");  
1020    public void display(String x) {  
        System.out.println("My truck is "+truck+x+"!");  
1022 }  
}
```

Listing 4: Primer deklarisanja klase sa enkapsulacijom i nasleđivanjem

### 4.1 Enkapsulacija

Ograničavanje pristupa internim podacima klase postizemo navođenjem ključne reči **private** ispred deklaracije promenljive u klasi. Private modifikator pristupa znači da se podacima može pristupiti isključivo iz deklarisanе klase. Modifikatori i njihova vidljivost prikazani u tabeli 1. Ako su podaci ipak potrebni van klase, omogućuje se njihovo čitanje i menjanje, preko javnih metode pristupa (eng. *getters and setters*) [7]. U primeru (4), vrednosti privatnog atributa plata možemo pristupiti metodom `getSalary()` ili menjati sa `setSalary(newSalary)`.

### 4.2 Nasleđivanje

Za nasleđivanje koristimo ključnu reč **extends**. Obratiti pažnju na primer (4), gde klasa vozač nasleđuje svojstva klase zaposleni. U javi postoji

Tabela 1: Vidljivost različitih modifikatora pristupa.

| Modifikator   | Klasa | Paket | Podklasa | Svet |
|---------------|-------|-------|----------|------|
| public        | Da    | Da    | Da       | Da   |
| protected     | Da    | Da    | Da       | Ne   |
| podrazumevani | Da    | Da    | Ne       | Ne   |
| private       | Da    | Ne    | Ne       | Ne   |

nasleđivanje po nivoima i hijerarhijsko. Višestruko ne postoji direktno[8], već se implementira preko interfejsa (detaljnije 4.4). Ako želimo iz izvedene klase da zovemo metode bazne, to radimo referisanjem na baznu klasu pomoću ključne reči **super**.

### 4.3 Polimorfizam

Višestruka upotrebljivost koda, tj. pripadnost metoda objektu se obavlja u vreme izvršavanja (eng. *run time execution*) i predstavlja koncept preklapanja metoda (eng. *overriding*)[7]. U primeru koda 4, Marko.display(); pozvaće metod klase vozač, iako postoji isti metod bazne klase.

Koncept preopterećenosti metoda (eng. *overloading*)[7], određuje metode u vremenu kompajliranja (eng. *compile time*) na osnovu razlika u potpisu metode (različito ime metoda ili tipovi i broj parametara). U primeru koda [4], Marko.display(2); pozvaće metod display(int x) klase vozač.

### 4.4 Apstrakcija

Izdvajanje skupa metoda sa kojima spoljašnji korisnik komunicira, prema članku [8], vršimo pomoću apstraktnih klasa ili interfejsa. Za apstraktne klase navodimo ključnu reč **abstract** (kod 5). Ne mogu se instancirati, ali može biti tip promenljive. Sadrže apstraktne metode koje treba da predefiniše neka podklasa.

```
1000 public abstract class Employee {
      public abstract void display(); ...
}
```

Listing 5: Apstraktna klasa

Interfejs predstavlja nacrt klase. Sadrži apstraktne, statične, podrazumevane metode (mogu se predefinisati u klasi) i statičke promenljive. Da implementiramo interfejs navodimo ključnu reč **implements** (kod 6) i zatim ime interfejsa. Prednost interfejsa[7] je da klasa može implementirati više interfejsa, dok može da nasleđuje samo jednu klasu.

```
1000 interface Employee {
      public void display(); #podrazumevano apstraktna
1002 default void work(){System.out.println("Working"); }
```

Listing 6: Interfejs

## 5 C#

C# je jednostavan, moderan, objektno-orijentisan jezik, razvijen od strane Microsoft-a i odobren od strane ECMA-e (European Computer Manufacturers Association). Nudi punu podršku objektno orijentisanom pro-

gramiranju uključujući nasleđivanje, enkapsulaciju, apstrakciju, i polimorfizam.

Ne podržava druge paradigme ali koristi svoje imperativne strukture. Veoma je sličan Java programskom jeziku.

## 5.1 Enkapsulacija

Enkapsulacija je kada se grupa povezanih metoda, svojstava i ostalih članova tretira kao jedan isti objekat. C# podržava sledeće specifikatore pristupa:

- **Public:** Dopušta klasi da izloži attribute i metode članova drugim metodama i atributima. Svakom public članu se može pristupiti izvan klase.
- **Private:** Dopušta klasi da sakrije attribute i metode članova od ostalih funkcija i objekata. Samo metode iz iste klase mogu pristupiti svojim privatnim članovima. Čak i instanca klase ne može pristupiti svojim privatnim članovima.
- **Protected:** Dopušta podređenoj klasi da pristupi atributima i metodama članova svoje osnovne klase. Na taj način pomaže u implementiranju nasleđivanja.
- **Internal:** Dopušta klasi da izloži attribute i metode članova drugim metodama i atributima u trenutnom skupu. Tačnije svakom članu sa ovim specifikatorom pristupa se može pristupiti iz bilo koje klase ili metode definisane unutar aplikaciju u kojoj je definisan član.
- **Protected internal:** Dopušta klasi da sakrije attribute i metode članova od ostalih metoda i atributa klase, osim podređene klase unutar iste aplikacije. To se takođe koristi kod implementacije nasleđivanja.

## 5.2 Nasleđivanje

Nasleđivanje je, kao što samo ime kaže, mogućnost da “nasleđuje” metode i svojstva od postojećih klasa. Kod C# nasleđivanja, klasa čiji su članovi nasleđeni se zove bazična (roditeljska) klasa, a klasa koja nasleđuje članove bazične klase se zove izvedena (podređena) klasa.

C# Koristi sintaksu C++ za definisanje klasa. Nasleđena metoda od roditeljske klase može biti zamenjena u izvedenoj klasi tako što se definiše kao `new(novo)`. Verzija roditeljske klase se i dalje može zvati eksplicitnom sa prefiksom `base(baza)`: `base.Draw()`.

## 5.3 Polimorfizam

U C#-u, polimorfizam omogućava klasama da implementiraju različite metode koje se nazivaju istim imenom i omogućava pozivanje metoda izvedene klase kroz referencu bazične klase tokom izvođenja, na osnovu naših zahteva.

U C#-u imamo dve različite vrste polimorfizma a to su: Polimorfizam vremena kompiliranja i polimorfizam vremena izvođenja.

## 5.4 Apstrakcija

Apstrakcija je proces kod koga programer krije sve osim relevantnih podataka o datom objektu u cilju pojednostavljanja i povećanja efikasnosti. U C#-u možemo napraviti klasu sa potrebnim metodama i svojstvima.



Izlažemo samo potrebne metode i svojstva koristeći modifikatore pristupa na osnovu naših zahteva, videti primer (7).

```
1000 abstract class MobilePhone {
1001     public void Calling();
1002     public void SendSMS();
1003 }
1004 public class Nokia1400: MobilePhone {}
1005 public class Nokia2700: MobilePhone {
1006     public void FMRadio();
1007     public void MP3();
1008     public void Camera();
1009 }
1010 public class BlackBerry: MobilePhone {
1011     public void FMRadio();
1012     public void MP3();
1013     public void Camera();
1014     public void Recording();
1015     public void ReadAndSendEmails();
1016 }
```

Listing 7: Primer deklarisanja apstraktivne klase u C#-u

## 6 Ada

Ada sledi model klase zasnovan na odvojenim karakteristikama (tipovima(eng. *types*) deklarisanim u jedinicama paketa(eng. *package*)), a ne na jednom konstruktoru klase[5]. Klasa u Adi ima koncept klase tipa koja se sastoji od skupa tipova kreiranih unutar deklaracije osnovnog tipa(eng. *root type*) unutar paketa. Svakom tipu je pridružena oznaka(eng. *tag*) koja ga razlikuje od ostalih tipova dodavanjem ključne reči **tagged** u deklaraciju osnovnog tipa. Unutar paketa možemo imati procedure(nemaju povratnu vrednost) i funkcije(imaju povratnu vrednost). U nastavku možemo videti primer(8) OOP u Adi(u primeru su navedene samo deklaracije zbog kompaktnosti).

```
1000 package Employees is
1001     type Employee is tagged
1002         record
1003             Name: String;
1004         end record;
1005     procedure Set_Name(Obj: in out Employee; Name: String);
1006     function Print(E: Employee) return String;
1007 end Employees;
1008 package Drivers
1009     type Driver is new Employees.Employee with record
1010         Driver_ID: Integer;
1011     end record;
1012     procedure Set_D(Obj: in out Employee; Name: String);
1013     overriding function Print(D: Driver) return String;
1014 end Drivers;
```

Listing 8: Primer objektno orijentisanog programiranja u jeziku Ada.

### 6.1 Enkapsulacija

U Adi, enkapsulacija je malo drugačija od većine OOP jezika jer je privatnost generalno određena na nivou paketa[1]. Pomoću ključne reči **private** u deklaraciji baznog tipa obezbeđujemo da podacima unutar tipa pristupamo jedino pomoću funkcija i procedura. Još veći stepen privatnosti dobijamo dodavanjem ključne reči **limited** ispred ključne reči **private**. Procedure i funkcije unutar paketa deklariramo kao privatne ukoliko ispred niza deklaracija procedura i funkcija dodamo ključnu reč **private**.

## 6.2 Nasleđivanje

Nasleđivanje po nivoima i hijerarhijsko nasleđivanje je direktno podržano u Adi, dok je višestruko nasleđivanje moguće implementirati pomoću interfejsa(eng. *interface*) uz dodatni rad programera. U primeru(8) deklaracija tipa *Driver* je tipičan primer kreiranja izvedenog tipa. Izvedeni tip nasleđuje sve podatke unutar tipa, kao i sve procedure i funkcije roditeljskog tipa.

## 6.3 Polimorfizam

Korišćenjem ključne reči **overriding** ispred deklaracije funkcije ili procedure u izvedenom tipu postizemo polimorfizam pomoću nasleđivanja koji se najintenzivnije koristi. Polimorfizam takođe postizemo korišćenjem apstraktnih tipova gde za različite izvedene tipove različito implementiramo procedure i funkcije koje nisu definisane. Još jedan način za postizanje polimorfizma je pomoću podtipova(eng. *subtypes*) o kojima neće biti reči.

## 6.4 Apstrakcija

Ada pruža podršku apstraktnim tipovima čije je ponašanje skoro identično ponašanju apstraktnih klasi drugih OOP jezika. Tip postaje apstraktan dodavanjem ključne reči **abstract** u njegovoj deklaraciji. Apstraktni tip bez komponenti nazivamo interfejsom(eng. *interface*). Dodavanjem ključne reči **abstract** u deklaraciji procedura i funkcija omogućavamo da budu definisane u izvedenim tipovima

## 7 Ruby

Podršku u programskom jeziku Ruby ilustrućemo primerom(9) koji pokriva sve bitnije aspekte objektno orijentisanog programiranja. Standardni metod klase je **initialize**, on se poziva automatski prilikom kreiranja objekta i ponaša se skoro identično kao konstruktori u drugim programskim jezicima[3].

```
1000 class Employee
1001   attr_accessor :name
1002   def initialize(name)
1003     @name = name
1004     print()
1005   end
1006   def print
1007     puts "Employee: #{@name}."
1008   end
1009 end
1010 class Driver < Employee
1011   def initialize(name)
1012     @name = name
1013     print()
1014   end
1015   private
1016   def print
1017     puts "Driver: #{@name}."
1018   end
1019 end
1020 emp = Employee.new("John")
1022 drv = Driver.new("John")
```

Listing 9: Primer objektno orijentisanog programiranja u jeziku Ruby.

## 7.1 Enkapsulacija

Kako u samom jeziku ne postoji mogućnost direktnog pristupa podacima unutar klase (podaci su privatni), njima možemo pristupiti jedino pomoću metoda klase. Svi metodi klase su javni, osim ako nije eksplicitno naznačeno drugačije ključnim rečima **public**, **protected**, **private** neposredno pre definicije jednog ili više metoda. Ruby nam pruža mogućnost ugrađenih metoda za pristup (eng. *accessor methods*). U primeru (9) **attr\_accessor** omogućava čitanje i menjanje vrednosti promenljivih klase. Pomocu **attr\_reader** i **attr\_writer** možemo pojedinačno dopustiti samo čitanje odnosno samo menjanje vrednosti promenljivih.

## 7.2 Nasleđivanje

Kada nakon imena klase u njenoj definiciji dodamo znak `<` za kojim sledi ime već postojeće klase, dobijamo efekat nasleđivanja koji možemo videti u prethodnom primeru (9) gde klasa *Driver* nasleđuje klasu *Employee* (primetiti da u klasi *Driver* nismo implementirali **attr\_accessor** jer se nasleđuje). Nasleđivanje po nivoima i hijerarhijsko nasleđivanje je moguće dok višestruko nasleđivanje nije (više o tipovima nasleđivanja u 4.2).

## 7.3 Polimorfizam

Osnovni vid polimorfizma možemo postići nasleđivanjem tako što će različiti objekti odgovoriti različito na iste metode. U primeru (9) u klasi *Driver* smo definisali metod *print* koji je istog naziva kao i nasleđeni metod čime postizemo da instanca klase odgovori različito na metod od instance roditeljske klase.

Drugačiji vid polimorfizma postizemo takozvanim "pačijim kucanjem" (eng. *duck typing*) u kojem nisu bitni tipovi objekata već skup istoimenih metoda koje poseduju. Za primer uzmimo klasu *Duck* koja poseduje metod *quack* i funkciju koja za argument uzima objekat tipa *Duck* i poziva metod *quack*. U tom slučaju funkciji možemo proslediti bilo koji objekat koji poseduje metod naziva *quack* (sa istim ili različitim ponašanjem metoda) i gledati na njega kao da je tipa *Duck* bez obzira što on to nije.

## 7.4 Apstrakcija

Ruby nema direktnu podršku za apstrakciju klasa ali se sličan efekat može implementirati [6]. Apstrakciju je moguće implementirati pomoću nasleđivanja gde roditeljska klasa definiše apstraktne metode koji pokreću "NotImplementedError" grešku tako da se ne mogu instancirati, pa mora postojati dete klasa koja će pomocu gore opisanog polimorfizma (7.3) nasleđivanjem implementirati željene apstraktne metode. Metode koje su zajedničke implementiramo u roditeljskoj klasi. Biblioteka **abstraction** nam olakšava postizanje efekta apstrakcije.

## 8 Zaključak

U ovom radu su objašnjene najvažnije karakteristike objektno-orijentisanog programiranja, od teorijskog uvoda u principe, do konkretne podrške tih principa u jezicima. Uviđamo popularnost objektno-orijentisanog programiranja zbog osobina poput modularnosti, jednostavnog održavanja i modifikovanja postojećeg koda, višestruka upotreba koda. Primećuje se da različiti jezici na različite načine podržavaju određene koncepte, neke i ne podržavaju, dok za druge pružaju alternativne načine za njihovo ostvarivanje. Kroz izložene primere programski jezika ovaj rad pruža odličan uvod za upoznavanje sa objektno-orijentisanom paradigmom i podstrek čitaocu na dalje i dublje istraživanje obrađenih tema.

## Literatura

- [1] Introduction to Ada. on-line at: <https://learn.adacore.com/courses/intro-to-ada/index.html>.
- [2] Object C apple documentation. on-line at: <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjectiveC>.
- [3] Ruby - Object Oriented. on-line at: [https://www.tutorialspoint.com/ruby/ruby\\_object\\_oriented.htm](https://www.tutorialspoint.com/ruby/ruby_object_oriented.htm).
- [4] Gary Bennett, Brad Lees, and Mitchell Fisher. *Objective-C for Absolute Beginners: iPhone, iPad and Mac Programming Made Easy*. Apress, Berkely, CA, USA, 3rd edition, 2016.
- [5] AdaCore experts. *High-Integrity Object-Oriented Programming in Ada*. AdaCore([www.adacore.com](http://www.adacore.com)), 1.2 release edition, 2011. on-line at: <http://extranet.eu.adacore.com/articles/HighIntegrityAda.pdf>.
- [6] Hal Fulton. *The Ruby Way*. Sams Publishing, 2001.
- [7] Cay S Horstmann. *Core Java SE 9 for the Impatient*. Addison-Wesley Professional, 2017.
- [8] Aayushi Johari. Object Oriented Programming – Java OOPs Concepts With Examples, 2018. on-line at: <https://www.edureka.co/blog/object-oriented-programming/>.