

Objektno orijentisano programiranje

Statička polja. Apstraktne klase. Virtualne metode.
Operator ispisa

Statička polja

- Statičko polje je polje koje je deljeno između svih instanci/objekata jedne klase
- Ponašanje statičkih polja je slično kao kod globalnih promenljivih. Razlika je što se statička polja vezuju za klasu u kojoj su deklarisan
- Statička polja pripadaju klasi, ali ne i objektu
- Deklarišu se ključnom reči ***static***

Polja se deklarišu kao statička preko deklaracije unutar klase:

```
static tip ime_promenljive;
```

Statička polja - primer

```
class Komponenta{  
    private:  
        static double duzina;  
        static double visina;  
        int boja;  
};
```

- Klasa Komponenta se sastoji od tri atributa dužina, visina i boja.
- Dužina i visina su statička polja jer se ispred tipa polja nalazi ključna reč *static*
- Inicijalizacija statičkih polja nije moguća unutar klase u kojoj su deklarirana
- Inicijalizacija statičkog polja mora biti nakon deklaracije klase


```
double Komponenta::duzina = 0;  
double Komponenta::visina = 0;
```

← Postavljanje vrednosti polja na 0

Statička polja - primer

- Pristup statičkim poljima unutar klase je isti kao i kod normalnih polja

```
public:  
    Komponenta() {  
        duzina++;  
        visina++;  
    }
```



Povećanje dužine i visine za jedan. Ovim se postiže da će vrednosti (veličina komponente) rasti kako se bude kreiralo više objekata klase Komponenta. Ako bude napravljeno deset objekata tada će dužina i visina za sve objekte imati vrednost deset. Ako bude napravljeno tri objekta tada će vrednost polja za sva tri objekta biti tri.

Virtuelne metode

- Metoda se proglašava virtuelnom samo jedanput, i to u roditeljskoj klasi i tada u svim klasama naslednicama ta metoda zadržava tu osobinu
- Redefinisana virtuelna metoda mora imati istu deklaraciju kao originalna
- Definišu se ključnom reči ***virtual*** pomoću deklaracije:

```
virtual tip naziv_metode(lista_parametara)
{
    ///telo metode
}
```

- Metoda se deklariše kao virtuelna u osnovnoj klasi
- Virtuelne metode osiguravaju da se za objekat poziva ispravna metoda, bez obzira na vrstu reference (ili pokazivača) koji se koristi za poziv metode

Virtualne metode - primer

```
class Komponenta{  
    private:  
        double duzina;  
        double visina;  
        int boja;  
    public:  
        Komponenta() {}  
        virtual void specifikacije() {  
            cout<<duzina<<visina;  
        }  
}
```

← Klasa iz prethodnog primera je roditeljska klasa, i ne
sadrži statička polja više.

Klasa sadrži virtualnu metodu preko koje se na
konzoli ispisuju vrednosti polja dužina i visina.

Virtualne metode - primer

```
class Labela: Komponenta{  
    private:  
        string tekst;  
    public:  
        Labela:Komponenta() {}  
        void specifikacije() {  
            Komponenta::specifikacije();  
            cout<<tekst;  
        }  
};
```

Pristup metodi osnovne klase preko operatora pristupa ::

Ponašanje osnovne klase se može zameniti (redefinirati) u klasama naslednicama ako je to potrebno (nije neophodno)

Na primeru je prikazana klasa Labela koja nasleđuje klasu Komponenta. Labela proširuje nasleđenu klasu sa poljem tekst. Pri pozivu metode specifikacije pored vrednosti koje ima osnovna klasa ispisuje se i vrednost polja tekst.

Virtualne metode - primer

- Prilikom pozivanja metode, određuje se verzija metode koja će se pokrenuti na osnovu stvarnog tipa objekta.

```
Komponenta k;  
Labela l;  
Komponenta *kp;  
kp = &k;  
kp->specifikacije();  
kp = &l;  
kp->specifikacije();
```

Instanciranje objekata klase Komponenta i labela.

Pokazivač klase komponenta

U zavisnosti od stvarnog tipa objekta na koju pokazuje pokazivač, takav će biti i rezultat poziva metode specifikacije. U prvom primeru kada kp pokazuje na objekat komponente ispisaće se vrednosti polja visina i duzina. Ako pokazivač pokazuje na objekat klase labela, pored ispisa ove dve vrednosti ispisaće se i vrednost polja tekst.

Virtualne metode - primer

```
void specifikacijaKomponente(const Komponenta &komp) {  
    komp.specifikacije();  
}
```

U zavisnosti od stvarnog tipa objekta koji se prosleđuje slobodnoj funkciji preko reference, takav će biti i rezultat poziva metode specifikacije. Rezultat je isti kao i na prethodnom slajdu.

```
void specifikacijaKomponente(Komponenta *komp) {  
    komp->specifikacije();  
}
```

Dobija se sličan rezultat prilikom prenosa objekta preko pokazivača (tj. njegove adrese)

```
void specifikacijaKomponente(Komponenta komp) {  
    komp.specifikacije();  
}
```

Ako se objekat ne prosledi preko reference slobodnoj funkciji, tada će rezultat poziva metode specifikacije uvek biti isti bez obzira na tip objekta koji se šalje (pozvaće se metoda specifikacije iz osnovne klase).

Apstraktna klasa

- Apstraktna metoda je virtuelna metoda koja nema telo (tj. nema realizaciju). Virtuelna metoda se proglašava apstraktnom tako što se na kraju njenog prototipa napiše =0, odnosno prototip apstraktne metode izgleda ovako:

```
virtual Tip_metode ime_metode(lista_parametara)=0;
```

- Klasa koja ima bar jednu apstraktnu metodu naziva se apstraktna klasa.
- Apstraktna klasa se ne može instancirati, tj. ne može se kreirati objekat apstraktne klase.
- Apstraktna klasa može imati polja, obične metode i konstruktore.

Primer 1

U ovom primeru je potrebno napisati klasu **Artikal** koja ima polja naziv,cena i broj instanci koje je statičko polje i koje će predstavljati broj napravljenih instanci klase **Artikal**. Takođe, potrebno je preklopiti operator ispisa preko kojeg će se testirati napisana klasa.

(Detaljan tekst je dat u materijalima za ove vežbe)

Rešenje: 1.1. Deklaracija klase Artikal - polja i metode

```
#ifndef ARTIKAL_HPP_INCLUDED
#define ARTIKAL_HPP_INCLUDED

#include "dinstring.hpp"
#include <iomanip>

class Artikal{
    private:
        DinString naziv;
        double cena;
        static int instanci;
    public:
        Artikal(const char[],const double);
        Artikal(const DinString&, const double);
        Artikal(const Artikal&);
        ~Artikal();

        DinString getNaziv() const;
        double getCena() const;

        friend ostream& operator<<(ostream& ,const Artikal&);
};

#endif // ARTIKAL_HPP_INCLUDED
```

Napomena: Potrebno je uključiti dinstring fajl kao i biblioteku iomanip za manipulaciju izlazom.

Pored normlanih atributa se nalazi i polje *instanci* koje je static i koje će deliti objekti klase Artikal.

Rešenje: 1.2. Kreirati klasu Artikal - konstruktori

- Potrebno je kreirati konstruktor bez parametara, konstruktor sa parametrima i konstruktor kopije.
- Potrebno je inicijalizovati statičko polje klase.
- Preporuka je da se odmah nakon kreiranja konstruktora i ostalih metoda u fajlu *artikal.cpp* izvrši njihovo testiranje u fajlu *main.cpp* (kreiranjem objekta ili pozivom metode u *main* funkciji).

Rešenje: 1.2. Kreirati klasu Artikal - konstruktori

Pri pozivanju konstruktora povećava se broj instanci, dok se kod pozivanja destruktora broj instanci smanjuje. Omogućava se brojanje “živih” objekata klase Artikal, objekata koji su u datom trenutku napravljeni a nisu uništeni.

```
#include "artikal.hpp"

int Artikal::instanci = 0; // mora van klase

Artikal::Artikal(const char n[], const double c) : naziv(n), cena(c){
    instanci++;
}

Artikal::Artikal(const DinString& n, const double c) : naziv(n), cena(c){
    instanci++;
}

Artikal::Artikal(const Artikal& a) : naziv(a.naziv), cena(a.cena){
    instanci++;
}

Artikal::~~Artikal(){
    instanci--;
}
```

Inicijalizacija polja *instanci* klase Artikal. Kao što je napomenuto, inicijalizacija statičkih polja se mora obaviti van klase!

Rešenje: 1.3. Kreirati klasu Artikal - preklapanje metode za ispis

Napomena: Funkcija `setw` je deo biblioteke `iomanip` i koristi se da se zauzme željena “širina” ispisa na konzoli.

```
ostream& operator<<(ostream& os, const Artikal& a){
    os << "----- ARTIKAL -----" << endl;
    os << setw(10)<<"NAZIV: " << a.naziv << endl;
    os << setw(10)<<"CENA: " << a.cena << endl;
    os << setw(10)<<"INSTANCI: " << a.instanci << endl;
    os << "-----" << endl << endl;

    return os;
}
```

Pri pozivu ispisa na konzoli datog objekta klase `Artikal` ispisaće se sledeće informacije o objektu koje smo “upisali” u objekat klase `ostream os` koji smo vratili na kraju metode.

Rešenje: 1.4. Testiranje programa

Testiranje objekata klase artikal za različite vrednosti atributa i ispisivanje njihovih vrednosti preko preklopljenog izlaza. Obratiti pažnju na vrednost static polja (prikazuje koliko je objekata klase Artikal napravljeno). Testiranje destruktora je izvršeno preko objekta a3. Pri ispisu objekta a3 vrednost static polja *instanci* će biti tri. Kada se završi blok naredbi gde je napravljen objekat a3, završava se životni vek objekata i poziva se destruktor koji će smanjiti vrednost polja instanci na dva.

```
#include "artikal.hpp"

int main()
{
    Artikal a1("Milka sa lesnikom 150gr", 142.23);
    cout << a1 << endl;

    DinString dl("Next jabuka 11");

    Artikal a2(dl, 110);
    cout << a2 << endl;

    // uloga destruktora:
    // kreira se treci objekat
    {
        Artikal a3(a2);
        cout << a3 << endl;
    }
    // unisten je treci objekat

    cout << a2 << endl;

    return 0;
}
```


Primer 2

Napisati klasu **Osoba** koja ima polja ime i prezime. Iz klase **Osoba** izvesti klasu **Student** koja ima dodatno polje za broj indeksa. Iz klase **Osoba** izvesti i klasu **PhDStudent** koja dodatno ima polje `prosecnaOcena`. Napisati kratak test program.

(Detaljan tekst je dat u materijalima za ove vežbe)

Rešenje: 2.1. Kreirati klasu Osoba - polja

Napomena: polja ime i prezime su tipa DinString, moramo uključiti dinstring.hpp

```
#ifndef OSOBA_DEF
#define OSOBA_DEF

#include "dinstring.hpp"
#include <iostream>

using namespace std;

class Osoba
{
protected:
    DinString ime, prezime;
```

Rešenje: 2.2. Kreirati klasu Osoba - konstruktori

- Potrebno je kreirati konstruktor bez parametara, konstruktor sa parametrima i konstruktor kopije.
- Preporuka je da se odmah nakon kreiranja konstruktora i ostalih metoda u fajlu *osoba.hpp* izvrši njihovo testiranje u fajlu *main.cpp* (kreiranjem objekta ili pozivom metode u *main* funkciji).

Rešenje: 2.2. Kreirati klasu Osoba - konstruktori

```
public:  
    Osoba(const char* s1 = "", const char* s2 = "") : ime(s1), prezime(s2) { }  
  
    Osoba(const DinString& ds1, const DinString& ds2) : ime(ds1), prezime(ds2) { }  
  
    Osoba(const Osoba& osoba) : ime(osoba.ime), prezime(osoba.prezime) { }
```


Konstruktori sa parametrima kao parametre primaju ime i prezime koju su reprezentovani kao niz karaktera (u prvom konstruktoru) i kao objekti klase DinString (u drugom konstruktoru). Prvi konstruktor je konstruktor sa predefinisanim vrednostima. Ako se ne unesu vrednosti za polja ime i prezime, napraviće se objekat koji će za vrednosti polja ime i prezime imati "".

Napomena: Postavljanje vrednosti polja ime i prezime se vrši kompozitno jer su ime i prezime tipa DinString čiji se objekat mora napraviti.

Rešenje: 2.3. Kreirati klasu Osoba - virtualna metoda za ispis

- Metoda se proglašava virtualnom u roditeljskoj klasi tako što se ispred tipa metode navede rezervisana reč **virtual**

Pri pozivu metode ispisujemo vrednosti polja ime i prezime na konzoli. Metoda je konstantna, što znači da se ne mogu menjati polja unutar te metode (kompajler će javiti grešku).



```
virtual void predstaviSe() const
{
    cout << "Zovem se " << ime << " " << prezime << "." << endl;
}
```

Rešenje: 2.4 Kreirati klasu Student

- Kreirati *student.hpp* i u njemu pisati kod za klasu **Student**.
- Klasa **Student** nasleđuje klasu **Osoba** i proširuje je poljem brojIndeksa.
- U klasi **Student** potrebno je implementirati konstruktor bez parametara, konstruktor sa parametrima, konstruktor kopije i preklopiti metodu za ispis (*predstaviSe*).

Rešenje: 2.5. Kreirati klasu Student

Napomena: Poželjno je nakon svake implementirane metode istu pozvati u *main.cpp*. Ne zaboraviti uključiti *osoba.hpp* jer se tu nalazi klasa koju nasleđujemo.

```
#ifndef STUDENT_DEF
#define STUDENT_DEF

#include "osoba.hpp"

class Student : public Osoba
{
protected:
    int brojIndeksa;

public:
    Student(const char* s1 = "", const char* s2 = "", int i = 0) : Osoba(s1, s2), brojIndeksa(i) { }

    Student(const DinString& ds1, const DinString& ds2, int i) : Osoba(ds1, ds2), brojIndeksa(i) { }

    Student(const Osoba& os, int i) : Osoba(os), brojIndeksa(i) { }

    Student(const Student& s) : Osoba((Osoba)s), brojIndeksa(s.brojIndeksa) { }
```

Rešenje: 2.5. Kreirati klasu Student - konstruktori

```
Student(const char* s1 = "", const char* s2 = "", int i = 0) : Osoba(s1, s2), brojIndeksa(i) { }
```

```
Student(const DinString& ds1, const DinString& ds2, int i) : Osoba(ds1, ds2), brojIndeksa(i) { }
```

Konstruktori sa parametrima (prikazana dva) su slični konstruktorima klase Osoba. Razlika je što imamo novi parametar koji predstavlja brojIndeksa. Od parametara koji nam predstavljaju ime i prezime “pravimo” deo studenta koji predstavlja Osobu, dok polje broj indeksa postavljamo na prosleđenu vrednost. Broj indeksa smo mogli postaviti na zadatu vrednost i u telu konstruktora.

```
Student(const Osoba& os, int i) : Osoba(os), brojIndeksa(i) { }
```

```
Student(const Student& s) : Osoba((Osoba)s), brojIndeksa(s.brojIndeksa) { }
```

Konstruktor sa parametrima (treći) u ovom slučaju prima ceo objekat klase Osoba i broj indeksa kao parametre i pravi objekat klase Student za prosleđene parametre.


Kod konstruktora kopije da bi se napravio objekat klase Student moramo napraviti i deo studenta koji predstavlja Osobu, zato je potrebno *castovati* prosleđenog studenta i dobiti te informacije.

Rešenje: 2.6. Kreirati klasu Student - preklapanje metode

Pri preklapanju metode potrebno je ispisati osnovne podatke iz roditeljske klase. Takođe, želimo da dodamo nove podatke koje su specifične za ovu klasu.

Pozivanje metode iz roditeljske klase se vrši operatorom pristupa ::

```
void predstaviSe() const
{
    Osoba::predstaviSe();
    cout << "Broj mog indeksa je " << brojIndeksa << "." << endl;
}
```

A black arrow points from the text "Pozivanje metode iz roditeljske klase se vrši operatorom pristupa ::" to the `Osoba::predstaviSe();` line in the code block.

Rešenje: 2.7 Kreirati klasu PhdStudent

- Kreirati *phdstudent.hpp* i u njemu pisati kod za klasu **PhDStudent**
- Klasa **PhDStudent** nasleđuje klasu **Student** i proširuje je poljem prosečna ocena
- Uraditi korake kao za klasu **Student**
- Potrebno je preklopiti metodu i pored informacija koje nosi klasa Student ispisati i vrednost dodatnog polja
- Uključiti *phdstudent.hpp* u *main.cpp*-u i testirati ga

Rešenje: 2.7 Kreirati klasu PhdStudent - konstruktori

```
PhDStudent(const char* s1 = "", const char* s2 = "", int i = 0, double po = 0) : Student(s1, s2, i), prosecnaOcena(po) { }  
PhDStudent(const DinString& ds1, const DinString& ds2, int i, double po) : Student(ds1, ds2, i), prosecnaOcena(po) { }  
PhDStudent(const Osoba& os, int i, double po) : Student(os, i), prosecnaOcena(po) { }  
PhDStudent(const Student& s, double po) : Student(s), prosecnaOcena(po) { }  
PhDStudent(const PhDStudent& phds) : Student((Student)phds), prosecnaOcena(phds.prosecnaOcena) { }
```

Sve o čemu je bilo reč pri objašnjavanju konstruktora klase Student važi i ovde. Kako klasa nasleđuje Studenta tako se i prave objekti klase PhDStudent. Od PhDStudenta se “pravi” njegov Student deo, a od Studenta njegov Osoba deo, što je i prikazano na slajdu gde su objašnjeni konstruktori.

Rešenje: 2.8 Kreirati slobodne funkcije

Kod prve funkcije pravimo kopiju dok kod druge funkciju prosleđujemo objekat po referenci. Kod druge funkcije pozvaće se metoda predstaviSe u zavisnosti od sadržaja objekta koji smo poslali kao parametar.

```
#include "phdstudent.hpp"

void predstavljaj1(Osoba osoba)
{
    cout << "Predstavljajanje 1: ";
    osoba.predstaviSe();
}

void predstavljaj2(const Osoba &osoba)
{
    cout << "Predstavljajanje 2: ";
    osoba.predstaviSe();
}
```

Ponašanje slobodne funkcije smo objasnili kod primera virtualnih metoda sa početka prezentacije, gde smo prikazali ponašanje objekata u zavisnosti od objekta na koji pokazuje pokazivač. Ako prosledimo studenta kao parametar, pored informacija o imenu i prezimenu prikazaće se i broj indeksa. Ako prosledimo objekat klase PhdStudent pored navedenih informacija koje će se ispisati kod studenta ispisaće se i prosečna ocena.

Rešenje: 2.9 Testiranje programa

Testiranje ponašanja virtualnih metoda se vrši pomoću slobodnih funkcija sa prethodnog slajda. Takođe je testirana metoda predstaviSe zasebno.

```
cout << endl << endl;  
cout << "Predstavljanje objekata preko funkcije" << endl;  
cout << endl;
```

```
cout << "Osoba:" << endl;  
predstavljanje1(os1);  
predstavljanje2(os1);
```

```
cout << endl;  
cout << "Student:" << endl;  
predstavljanje1(st1);  
predstavljanje2(st1);
```

```
cout << endl;  
cout << "PhDStudent:" << endl;  
predstavljanje1(phds1);  
predstavljanje2(phds1);
```

```
// testirati i kad metod nije virtualan
```

```
return 0;
```

Primer 3

Napisati apstraktnu klasu **Figura**, koja sadrži apstraktne metode za izračunavanje obima i površine. Iz apstraktne klase **Figura** izvesti klase **Pravougaonik** i **Elipsa**. Iz klase **Pravougaonik** izvesti klasu **Kvadrat**. Iz klase **Elipsa** izvesti klasu **Krug**. Napisati klasu **Oblik** koja sadrži polja: A (Kvadrat) i B (Krug).

(Detaljan tekst je dat u materijalima za ove vežbe)

Rešenje 3.1 Kreirati klasu Figura

Klasa sadrži dve apstraktne metode što čini i klasu Figura apstraktnom.

Napomena: Klasa Figura se ne može instancirati

```
class Figura
{
    public:
        virtual double getO() const = 0;
        virtual double getP() const = 0;
};
```

Rešenje 3.2. Kreirati klasu Pravougaonik

- Kreirati *pravougaonik.hpp* i u njemu pisati kod za klasu **Pravougaonik**.
- Klasa **Pravougaonik** nasleđuje klasu **Figura**
- U klasi **Pravougaonik** potrebno je implementirati konstruktor bez parametara, konstruktor sa parametrima, konstruktor kopije i preklopiti metode za površinu i obim.

Rešenje 3.3 Kreirati klasu Pravougaonik - preklapanje metoda

Prilikom nasleđivanja apstraktne klase, sve apstraktne metode moraju biti implementirane u izvedenoj klasi kako izvedena klasa ne bi bila apstraktna. Vrednosti koje vraćaju ove metode se dobijaju putem osnovnih formula za računanje obima i površine pravougaonika.

```
double getO() const
{
    return 2 * (a + b);
}
```

```
double getP() const
{
    return a * b;
}
```

Rešenje 3.4 Kreirati klasu Elipsa

- Kreirati *elipsa.hpp* i u njemu pisati kod za klasu **Elipsa**.
- Klasa **Elipsa** nasleđuje klasu **Figura**
- U klasi **Elipsa** potrebno je implementirati konstruktor bez parametara, konstruktor sa parametrima, konstruktor kopije i preklopiti metode za površinu i obim.
- Klasa je veoma slična klasi Pravougaonik. Razlika je u formulama za računanje obima i površine.

Rešenje 3.5 Kreirati klasu Krug

- Kreirati *krug.hpp* i u njemu pisati kod za klasu **Krug**.
- Klasa **Krug** nasleđuje klasu **Elipsa**

```
class Krug : public Elipsa {  
  
    public:  
        Krug() : Elipsa(2,2) {}  
        Krug(double xx) : Elipsa(xx,xx) {}  
        Krug(const Krug& k) : Elipsa((Elipsa)k) {}  
  
};
```

Rešenje 3.6 Kreirati klasu Kvadrat

- Kreirati *kvadrat.hpp* i u njemu pisati kod za klasu **Kvadrat**.
- Klasa **Kvadrat** nasleđuje klasu **Pravougaonik**

```
class Kvadrat : public Pravougaonik
{
    public:
        Kvadrat() : Pravougaonik(3,3) {}
        Kvadrat(double aa) : Pravougaonik(aa,aa) {}
        Kvadrat(const Kvadrat& k) : Pravougaonik((Pravougaonik)k) {}
};
```

Rešenje 3.7 Kreirati klasu Oblik

- Kreirati *oblik.hpp* i u njemu pisati kod za klasu **Oblik**.
- Klasa **Oblik** se sastoji od objekta klase Kvadrat kao i objekta klase Krug.
- U klasi **Oblik** potrebno je implementirati konstruktor bez parametara, konstruktor sa parametrima, konstruktor kopije kao i metode za računanje obima i površine.

Rešenje 3.8 Kreirati klasu Oblik - metode za računanje površine i obima

Kod računanja površine datog oblika oduzimamo površinu kvadrata od površine kruga, dok se kod računanja obima vrednosti sabiraju.

```
double getP() const
{
    return A.getP() - B.getP();
}
```

```
double getO() const
{
    return B.getO() + A.getO();
}
```

Rešenje 3.9. Testiranje programa

Testirane su metode i konstruktori svih napisanih klasa i ispitane su vrednosti koje se dobijaju kod poziva metoda obima i površine. Na slici je dat primer za testiranje klase pravougaonik.

```
int main()
{
    cout << "Pravougaonik p1:" << endl;
    Pravougaonik p1;
    cout << "Povrsina: " << p1.getP() << endl;
    cout << "Obim: " << p1.getO() << endl << endl;

    cout << "Pravougaonik p2:" << endl;
    Pravougaonik p2(3,5);
    cout << "Povrsina: " << p2.getP() << endl;
    cout << "Obim: " << p2.getO() << endl;
    cout << " " << endl;

    cout << "Pravougaonik p3:" << endl;
    Pravougaonik p3(p2);
    cout << "Povrsina: " << p3.getP() << endl;
    cout << "Obim: " << p3.getO() << endl << endl;
```

Literatura

1. Kupusinac A. : Zbirka rešenih zadataka iz programskog jezika C++. Novi Sad: FTN, 2011.