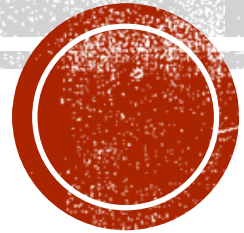
ОБЈЕКТНО ОРИЈЕНТИСАНО ПРОГРАМИРАЊЕ

ПРОГРАМСКИ ЈЕЗИК ЈАВА – 1

Напредни рад са класама и објектима



АПСТРАКТНЕ КЛАСЕ

- Када постоји заједничка надкласа за више класа, али је та надкласа веома општа, тада се та надкласа може прогласити апстрактном.
- Притом се захтева да свака од потклаcа мора да има своју конкретну реализацију тог општег понашања надкласе.
- **Пример.**
  - Претпоставимо да треба реализовати класу **GeometrijskiObjekat** у равни.
  - Постоје различите врсте геометријских објеката:
    троугао, квадрат, круг и сл.
  - Геометријски објекти имају методе попут обима и површине.
    које се различито дефинишу за различите објекте.
  - Треба обавезати сваку потклaсу **GeometrijskiObjekat** да дефинише ове методе.

ДЕФИНИСАЊЕ АПСТРАКТНЕ КЛАСЕ

- Апстрактни метод се означава кључном речи **abstract**.
- Такав апстрактан метод нема тело, тј. иза декларације следи само тачка-зарез.

```
public abstract double povrsina();
```

- Ако нека класа садржи апстрактан метод, тада она мора бити апстрактна.
  - Мада могу постојати апстрактне класе које не садрже апстрактне методе.
  - Иста кључна реч **abstract** служи и за означавање да је дата класа апстрактна.

```
public abstract class GeometrijskiObjekat {
…
        public abstract double povrsina();
}
```

ПРИМЕР 1

- Креирати Јава класу која представља геометријски објекат у равни, тако да иста оогући једноставну надоградњу и прошитивање.
- Сваки такав геометријског објекат у перспективи треба да обезбеди могућност за:
  - проверу конвексности,
  - проверу ограничености
  - и мерење обима и површине.

ПРИМЕР 1 (2)

```
public abstract class GeometrijskiObjekat {

  private String oznaka;

  public GeometrijskiObjekat(String oznaka) {
      this.oznaka = oznaka;
  }

  public String getOznaka() {
      return oznaka;
  }

  public void setOznaka(String oznaka) {
      this.oznaka = oznaka;

  }

  public abstract boolean jeKonveksan();

  public abstract boolean jeOgranicen();

  public abstract double obim();

  public abstract double povrsina();
}
```

ПРИМЕР 1 (3)

- Приметити да постоје и не-апстрактни методи.
  - Методи за узимање и постављање ознаке раде потпуно исто па нису апстрактни.
- Методи за проверу конвексности и ограничености, те за одређивање обима и површине геометријског објекта су превише општи.
  - Нама никавог смисленог начина њихове реализације на овом нивоу општости.
  - Стога је њихова реализација делегирана подкласама ове класе, па ће они овде бити дефинисани као апстрактни методи.
  - То даље значи да свака класа изведена из ове класе, а која сама није апстрактна, има обавезу да реализује сва четири апстрактна метода

АПСТРАКТНЕ КЛАСЕ НАПОМЕНЕ

- Апстрактна класа се не може директно инстанцирати, тј. не може се направити применом оператора **new**.
  - Нема смисла креирати нешто што није још завршено.
  - Међутим, ако се зна да је сваки примерак неке класе истовремено и примерак сваке од наткласа те класе, јасно да се креирани примерак конкретне подкласе неке апстрактне класе може посматрати као примерак те апстракне класе.

НАСЛЕЂИВАЊЕ ИЗМЕЂУ АПСТРАКТНИХ И КОНКРЕТНИХ КЛАСА

- Синтаксa наслеђивања је иста као код обичних класа – кључна реч **extends.**
- Поткласа апстрактне класе:

1. може реализовати све апстракнте методе
   и у том случају она постаје конкретна класа;
2. не мора реализовати апстрактне методе
   и у том случају и поткласа остаје апстрактна.

ПРИМЕР 2

- Написати Јава класе за геометријске објекте у равни, који представљају тачку, дуж и праву.
- Тачка поред апстрактних геометријских метода треба да реализује и метод за рачунање удаљности до задате тачке.
- Дуж треба да реализује методе за рачунање дужине и проверу да ли дуж садржи задату тачку.
- Права треба да реализује методе за проверу да ли права садржи задату тачку као и методе за проверу да ли су две прослеђене тачке са истих или различитих страна праве.

ПРИМЕР 2 (1)

- Решење је предугачко за слајдове, па приказујемо само реализацију класе **Tacka**.

```
import java.util.Objects;

public class Tacka extends GeometrijskiObjekat {

  private double x;
  private double y;

  public Tacka(String oznaka, double x, double y) {
      super(oznaka);
      this.x = x;
      this.y = y;
  }

  public Tacka(double x, double y) { this("", x, y);  }

  public Tacka(String oznaka) { this(oznaka, 0, 0); }

  public Tacka() { this("O", 0, 0); }

  public Tacka(final Tacka t) { this(t.getOznaka(), t.x, t.y); }
```

ПРИМЕР 2 (2)

```
public double uzmiX() { return x; }

public void postaviX(double x) { this.x = x; }

public double uzmiY() { return y; }

public void postaviY(double y) { this.y = y; }

public double rastojanje(Tacka t) {
    return (Math.sqrt(Math.pow(t.x - x, 2) + Math.pow(t.y - y, 2)));
}


 @Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || !(o instanceof Tacka)) return false;
    Tacka tacka = (Tacka) o;
    return Double.compare(tacka.x, x) == 0 && Double.compare(tacka.y, y) == 0;
}
```

ПРИМЕР 2 (3)

```
  @Override
  public int hashCode() { return Objects.hash(x, y); }

  @Override
  public String toString() { return getOznaka() + "(" + x + "," + y + ")"; }

  @Override
  public boolean jeKonveksan() { return true; }

  @Override
  public boolean jeOgranicen() { return true; }

  @Override
  public double obim() { return 0; }

  @Override
  public double povrsina() { return 0; }
}
```

ПРИМЕР 3

- Написати Јава класе за геометријске објекте у равни, који се односе на троугао, четвороугао и круг.
- Сваки од ових објеката треба да реализује
  и метод за испитивање да ли садржи задату тачку.

ПРИМЕР 3 (2)

- Решење је предугачко за слајдове па приказујемо само реализацију класе **Krug**.

```
package rs.math.oop.g09.p01.geometrijskiObjekti;


import java.util.Objects;
import static java.lang.Math.*;

public class Krug extends GeometrijskiObjekat {
  private Tacka o;
  private double r;

  public Krug(String oznaka, Tacka o, double r) {
      super(oznaka);
      this.o = new Tacka(o);
      this.r = r;
  }
```

ПРИМЕР 3 (3)

```
public Krug(Tacka o, double r) { this("", o, r); }

  public Krug(final Krug kr) { this(kr.getOznaka(), kr.o, kr.r); }

  @Override
  public boolean equals(Object o1) {
      if (this == o1) return true;
      if (o1 == null || !(o instanceof Krug)) return false;
      Krug krug = (Krug) o1;
      return o.equals(krug.o) && Double.compare(krug.r, r) == 0;
  }

  @Override
  public int hashCode() { return Objects.hash(o, r); }

  @Override
  public String toString() { return getOznaka() + ":[" + o + ";" + r + "]";  }


  public boolean sadrzi(Tacka t) { return t.rastojanje(o) <= r; }
```

ПРИМЕР 3 (4)

```
  @Override
  public boolean jeKonveksan() { return true; }

  @Override
  public boolean jeOgranicen() { return true; }

  @Override
  public double obim() { return 2 * r * PI; }

  @Override
  public double povrsina() { return pow(r, 2) * PI; }
}
```

ИНТЕРФЕЈСИ

- У развоју софтвера је често важно да се различите групе програмера договоре око “уговора“ о интеракцији приликом заједничког рада.
- Свака од тих група треба да буде у могућности да напише свој део кода, а да при томе нема информације како је писан код друге стране.
- Програмски језик Јава у ту сврху користи интерфејсе.
- Интерфејс се може посматрати као нека врта уговора између класе која га користи и класе која га имплементира.
- Интерфејси представљају уговоре (они описује понашање), а класе које их имплеметирају и користе представљају уговорне стране

ИНТЕРФЕЈСИ И АПСТРАКТНЕ КЛАСЕ

- Слично апстрактним класама, интерфејси обезбеђују шаблоне за неко понашање, а које ће друге класе користити.
- За разлику од апстрактних класа која шаблоне за апстрактно понашање прослеђују само подкласама, код интерфејса се шаблони се могу проследити било коме.
- Интерфејс је референтни тип, сличан класи.
  - Или прецизније речено тотално апстрактној класи где су сви методи апстрактни тј. без реализације.
- Дакле, у оквиру интерфејса се по правилу могу наћи само заглавља метода и дефиниције константи.
  - Иако је у Јави почев од од верзије 8 допуштен изузетак од правила (није покривено овде).

ДЕФИНИСАЊЕ ИНТЕРФЕЈСА

Интерфејс се дефинише слично класи,
само што се уместо кључне речи **class** користи кључна реч **interface.**

```
public interface Radoznao {
	void prikaziUpit();
	String tekstUpita();
}
```

- Уочава се да, методи интерфејса немају тело.
- Даље, с обзиром да су по правилу методи интерфејса апстрактни,
  то нема потребе да се та чињеница додатно наглашава кључном речју **abstract.**
- Слично као код апстрактних класа, иако постоје променљиве типа интерфејса,
  није могуће директно кеирати примерак интерфејса помоћу оператора **new.**

ИМПЛЕМЕНТАЦИЈА ИНТЕРФЕЈСА

- Дефинисани интерфејси се имплементирају од стране Јава класа.

```
public class Strucnjak implements Radoznao {

  @Override
  public void prikaziUpit() {
      System.out.println(

            "Реализација метода prikaziUpit() интерфејса Radoznao у класи Strucnjak");
  }

  @Override
  public String tekstUpita() {
      return "Реализација метода tekstUpita() интерфејса Radoznao у класи Strucnjak";
  }

  public String prikaziUpit2(){
      return "Реализација метода prikaziUpit2() у класи Strucnjak";
  }

}
```

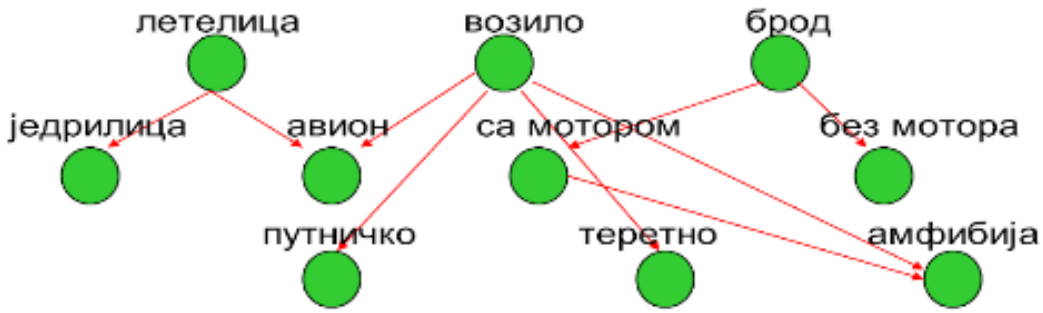
КОНВЕРЗИЈА КЛАСЕ У ИНТЕРФЕЈС

- Као што је то био случај са обичним класама које су у релацији наслеђивања, могуће је вршити конверзију објекта класе у интерфејс који класа имплементира.

```
public class Pitanja {

    public static void main(String[] arg)
    {
        Strucnjak p1 = new Strucnjak();
        p1.prikaziUpit();
        System.out.println(p1.tekstUpita());
        System.out.println(p1.prikaziUpit2());

        Radoznao p2 = new Strucnjak();
        p2.prikaziUpit();
        System.out.println(p2.tekstUpita());
    }
}
```

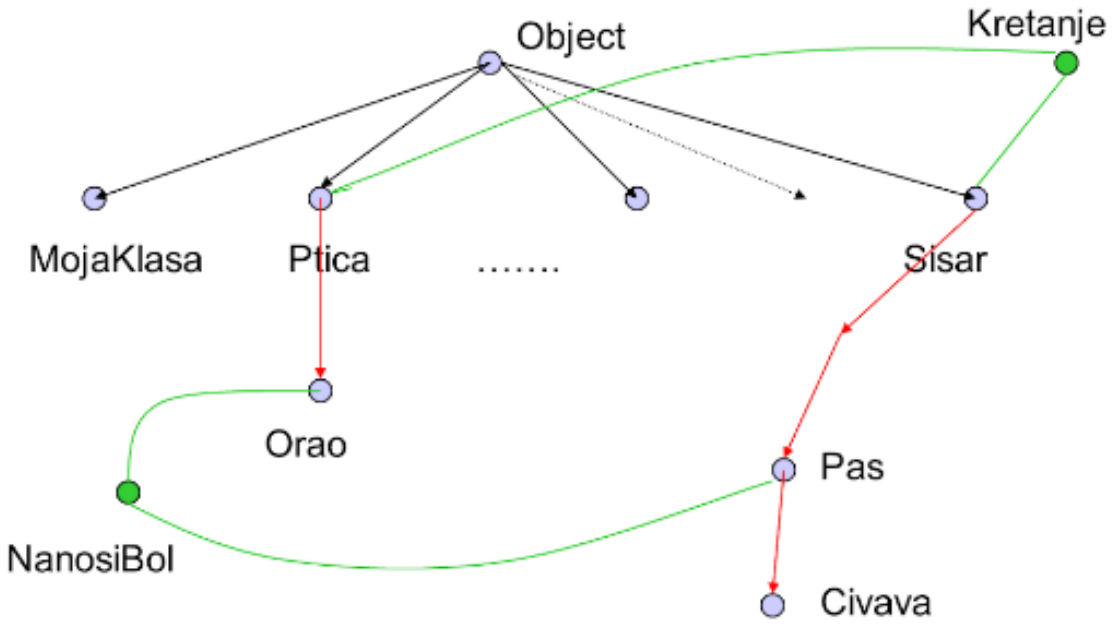
ВИШЕСТРУКО НАСЛЕЂИВАЊЕ И ИНТЕРФЕЈСИ

- Јави свака класа има тачно једну (било конкретну, било апстрактну) надкласу, али да може да имплементира више од једног интерфејса.
- У Јави је једноструко (дрво), у C++ је вишеструко (усмерени ациклички граф).



ВИШЕСТРУКО НАСЛЕЂИВАЊЕ – КОМПРОМИС СА ИНТЕРФЕЈСОМ

- Интерфејси у Јави омогућавају да се делимично компензује непостојање вишеструког наслеђивања - једна класа може имплементирати више интерфејса, а један интерфејс може проширивати један или више интерфејса.



ПРИМЕР 7

- Нека нам је на располагању већ развијен интерфејс **Radoznao.**
- Написати интерфејс **Razuman** са методама **razmortiCinjenice()** и **definisiHipotezu()**, Јава класу **Naucnik** са ниска-атрибутом **ime** која имплементира интерфејсе **Radoznao** и **Razuman,** као и програм који користи имплементиране методе преко промењљивих типа интерфејса.

ПРИМЕР 7 (2)

```
public interface Razuman
{
      void razmortiCinjenice();

      void definisiHipotezu();
}
```

```
Naucnik n = new Naucnik("Марковић");

Radoznao rd = n;
rd.prikaziUpit();
System.out.println(rd.tekstUpita());

Razuman rz = n;
rz.razmortiCinjenice();
rz.definisiHipotezu();
```

```
public class Naucnik implements Radoznao, Razuman {
  private String ime;

  public Naucnik(String ime) {
      this.ime = ime;
  }

  public String getIme() {
      return ime;
  }
  @Override
  public void prikaziUpit() { … }
  @Override
  public String tekstUpita() { … }
  @Override
  public void razmortiCinjenice() { … }
  @Override
  public void definisiHipotezu() { … }
```

ПРОШИРИВАЊЕ (НАСЛЕЂИВАЊЕ) ИНТЕРФЕЈСА

- Може се успоставити и однос проширивања између итерфејса, па интерфејс може проширити било један интерфејс, било већи број интерфејса.
- Однос проширивања се дефинише помоћу кључне речи **extends** – исте која се користи за дефинисање односа наслеђивања између класа.
- У телу интерфејса који проширује друге интерфејсе може, али не мора бити додатних метода.
- Класа која имплементира проширени интерфејс мора садржавати реализацију свих наведених метода, а ако се то не учини, класа се мора прогласити апстрактном.

ПРИМЕР 8

- Нека су нам на располагању већ развијени интерфејси **Radoznao** и **Razuman**, као и претодно развијена класа **Naucnik.**
- Креирати нови интерфејс **Eksperimentator** који проширује интерфејсе **Radoznao** и **Razuman** и који садржи метод **realizujeEksperimente()**.
- Креирати Јава класу **Istrazivac** изведену из класе **Naucnik** која имплементира интерфејс **Eksperimentator,** са ниска-атрибутом **probojUOblasti** и са превазиђеним методом за дефинисање хипотезе.
- Написати Јава програм који користи имплементиране методе преко промењљивих типа интерфејса.

ПРИМЕР 8 (2)

```
public interface Eksperimentator extends Radoznao, Razuman{
  void realizujeEksperimente();
}

public class Istrazivac extends Naucnik implements Eksperimentator{

  String probojUOblasti;

  public Istrazivac(String ime, String probojUOblasti) {
      super(ime);
      this.probojUOblasti = probojUOblasti;
  }

  @Override
  public void definisiHipotezu() { … }

  @Override
  public void realizujeEksperimente() { … }
}
```

ПРИМЕР 8 (3)

```
Eksperimentator eksp = new Istrazivac("Петровић", "Молекуларна Биологија");
eksp.prikaziUpit();
eksp.razmortiCinjenice();
eksp.definisiHipotezu();
eksp.realizujeEksperimente();

Radoznao rdz = eksp;
rdz.prikaziUpit();

Razuman rzm = eksp;
rzm.razmortiCinjenice();
rzm.definisiHipotezu();
```

ПРИМЕР 9

- Креирати Јава програм за рад са геометријским објектима у равни, тако да програм допусти једноставну надоградњу и проширивање.
- Треба обезебедити проверу конвексности, проверу ограничености, проверу припадности тачке објекту и мерење обима и површине.
- Треба реализовати класе за геометријске објекте који представљају тачку, дуж, праву, троугао, четвороугао и круг.
- На крају треба креирати разноврсне геометријске објекте, приказати их, срачунати њихов укупни обим и укупну површину,
  а потом за тачку чије се координате учитавају са стандардног улаза одредити који је од креираних геометријских објеката садрже, а који не.

ПРИМЕР 9 (2)

- Решење је предугачко за слајдове. Погледати у књизи.

ИНТЕРФЕЈСИ У JDK

- Као што је истакнуто, интерфејси бивају имплементирају од стране Јава класа.
- Могу се имплементрати интерфејси које је програмер осмислио,
  али и интерфејси које су осмислили креатори Јаве
  и који су у оквиру JDK испоручени заједно са програмским окружењем Јава.
- Размотрићемо следећа три, често коришћена интерфејса:
  - **Comparable**
  - **Comparator**
  - **Cloneable**

СОРТИРАЊЕ, COMPARABLE

- Обично испоручилац услуге тврди:
  Ако класа имплементира конкретни интерфејс, ја ћу онда пружити услугу.
- Таква је ситуација са методом **sort()** у класи **Arrays**.
- Метод **sort()** ће сортирати низ ниски, али под једним условом
  - елементи у низу морају сами знати како да се упореде.
- Одговорност за упоређивање, дакле, није на методу **sort()** класе **Arrays**,
  већ на конкретној класи чији низ објеката се сортира.
- У оквиру метода **sort()** ће се тај начин уређивања дефинисан уговором користити.
- Уговор је дефинисан интерфејсом **Comparable**.

```
public interface Comparable {
   int compareTo(Object other);
}
```

ПРИМЕР 12

- Написати Јава програм за сортирање низа тачака.
- Дата тачка треба да буде пре друге тачке ако је ближа координатном почетку, или ако су им растојања до кооридната почетка иста, а она има мању **y** координату тј. мању ординату.

ПРИМЕР 12 (2)

```
public class Tacka extends GeometrijskiObjekat implements Comparable{
  private double x;
  private double y;

  @Override
  public int compareTo(Object obj) {
        if (!(obj instanceof Tacka))

              return -1;
        Tacka t = (Tacka) obj;
        Tacka o = new Tacka(0, 0);
        double razlika = rastojanje(o) - t.rastojanje(o);
        if (razlika < 0) return -1;
        if (razlika > 0) return 1;
        return (int)(y - t.y);
  }
}
```

ВИШЕКРИТЕРИЈУМСКО СОРТИРАЊЕ - COMPARATOR

- Веома често је потребно да у оквиру истог софтверског система колекције и низови елемената буду сортирани према различитим критеријумима.
- Претходно описани механизам сортирања допушта сортирање према једном критеријуму, где се користи статички метод **sort()** класе **Arrays.**
- Међутим, метод **sort()** је преоптерећен, па у JDK постоји и варијанта овог статичког метода са два параметра, где је:
  - први параметар низ,
  - а други параметар је објекат типа **Comparator** који описује поређење два елемената низа.
- Ако се користи овај потпис метода **sort()**, подразумевани начин поређења, евентуално дефинисан интерфејсом **Comparable**, занемарује се.

```
public interface Comparator {
   int compare(Object o1, Object o2);
}
```

ПРИМЕР 13

- Написати Јава програм за сортирање низа тачака по више критеријума:
  - по локацији од ближих према даљим,
  - по ознаци,
  - као и по локацији од даљих према ближим.

ПРИМЕР 13 (2)

- Решење је предугачко за слајдове. Погледати у књизи.

КЛОНИРАЊЕ - CLONEABLE

- Јасно је да постоји потреба креирања копије неког већ направљеног објекта.
  - Један начин да се то постигне су копирајући конструктори, раније описани.
  - Други начин за креирање копије већ направљеног објекта је механизам клонирања.
- Класа **Object** садржи метод **clone()**,
  који се може искористити за прављење копије тј. клона датог објекта.
- Метод **clone()** у класи **Object** је проглашен заштићеним,
  тј. означен је кључном речју **protected**.
  - Ово обично значи да он није потпуно спреман за коришћење, већ да се у класи која превазилази тај метод морају реализовати још неке додатне активности.
- Иако је метод за клонирање дефинисан у најопштијој класи, то не значи да свака од класа коју креира програмер треба да превазилази метод за клонирање
  - Програмер сигнализира да класа подржава клонирање имплементирањем **Cloneable**.

```
public interface Cloneable { }
```

ПРИМЕР 15

- Написати Јава програм где се клонирају примерци класе која представља запосленог и испитује понашање оригиналног и клонираног објекта.

ПРИМЕР 15 (2)

```
public class Zaposleni implements Cloneable {
  private String ime;
  private String prezime;
  private String opisPosla;
  private double plata;

  public Zaposleni(String ime, String prezime, String opisPosla, double plata) {
      this.ime = ime;
      this.prezime = prezime;
      this.opisPosla = opisPosla;
      this.plata = plata;
  }

  public Zaposleni(){ this("","", "", 0); }

  public void postaviPrezime(String prezime) { this.prezime = prezime; }
```

ПРИМЕР 15 (3)

```
    public void povecajPlatu(double zaProcenat) {
        double iznosPovisice = plata * zaProcenat / 100;
        this.plata += iznosPovisice;
    }

    @Override
    public String toString() {
        return "[име: " + ime + " " + prezime + ", посао: '" + opisPosla
                + "', плата: " + plata + "]";
    }

    @Override
    public Zaposleni clone() throws CloneNotSupportedException {
        Zaposleni klonirani = (Zaposleni) super.clone();
        return klonirani;
    }
}
```

ПРИМЕР 15 (4)

```
Zaposleni original = new Zaposleni("Јован", "Петровић", "припрправник",
30_000);
Zaposleni klon = original.clone();
System.out.println("После клонирања, пре промена:");
System.out.println("оригинал = " + original);
System.out.println("клон =     " + klon);

original.postaviPrezime("Станишић");
System.out.println("После постављања презимена оригинала на 'Станишић'");
System.out.println("оригинал = " + original);
System.out.println("клон =     " + klon);

klon.povecajPlatu(10);
System.out.println("После повећања плате клона за 10%");
System.out.println("оригинал = " + original);
System.out.println("клон =     " + klon);
```

ПРИМЕР 15 (5)

```
После клонирања, пре промена:
оригинал = [име: Јован Петровић, посао: 'припрправник', плата: 30000.0]
клон =     [име: Јован Петровић, посао: 'припрправник', плата: 30000.0]
После постављања презимена оригинала на 'Станишић'
оригинал = [име: Јован Станишић, посао: 'припрправник', плата: 30000.0]
клон =     [име: Јован Петровић, посао: 'припрправник', плата: 30000.0]
После повећања плате клона за 10%
оригинал = [име: Јован Станишић, посао: 'припрправник', плата: 30000.0]
клон =     [име: Јован Петровић, посао: 'припрправник', плата: 33000.0]
```

ДУБОКО КЛОНИРАЊЕ

- Претходни пример је демонстрирао клонирање тзв. „плитког“ објекта, тј. објекта који у себи нема под-објекте.
- Да би се спровело тзв. „дубоко“ копирање потребан је додатни труд.
- То демонстрирају примери 16 и 17, чије је разумевање потребно за највишу оцену.

ООП ПРИНЦИПИ

- Током протеклих деценија развоја програмирања, дефинисани су разнорсни прицпипи и препоруке.
- Предложени принципи доброг програмирања суштински представљају један начин формулисања шта треба радити а шта не.
- Наравно, те принципе не треба посматрати чврсто као матемтичке аксиоме, већ као мапу која олакшава налажење у путу.
- Ради лакшег памћења, обично се принцими програмирања означавају акронимима.
- Нпр. **DRY - Don't Repeat Yourself** (срп. Немој да се понављаш).
- **KISS - Keep It Simple, Stupid** (срп. Нека остане једноставно, блесави).
- Овде ће додатна пажња бити посвећена групи принципа доброг обектно оријентисаног програмирања која се означава акронимом **SOLID**.

SOLID ПРИНЦИПИ

- Група принципа **SOLID** се обично приписују Роберту Мартину (познатом и под надимком “Ујка Боб”),
- Иако он није осмислио свих пет SOLID принципа, његов ангажман је утицао на то да буду формулисани у баш овом облику.
- **S - Single responsibility principle** (срп. Принцип јединствене одговорности)
- **O - Open–closed principle** (срп. Принцип отворености и затворености)
- **L - Liskov substitution principle** (срп. Принцип замјенљивости Лискова)
- **I - Interface segregation principle** (срп. Принцип раздвајања интерфејса)
- **D - Dependency inversion principle** (срп. Принцип инверзије зависности).

S - ПРИНЦИП ЈЕДИНСТВЕНЕ ОДГОВОРНОСТИ

- Принцип једнозначне одговорности: свака класа треба да има тачно једну одговорност.
- Када се поштује овај принцип, тестирање је једноставније, што је у складу са агилним методологојама развоја софтвера, јако популарним у последње време.
- Мање функционалности у једној класи такође значи да има мање зависности од осталих класа, што доводи до боље организације кода.

ПРИМЕР 18

- Креирати класу за рад са подацима са подацима о запосленима,
  где ће функционалност која подржава упис информација о запосленом у базу
  података да буде у оквиру класе која представља запосленог.

ПРИМЕР 18 (2)

```
public class LosPrincipS {

    public static void main(String[] args) throws CloneNotSupportedException{
        Zaposleni z = new Zaposleni("Јован", "Петровић", "приправник",
                                    30_000);
        System.out.println("Запослени: " + z);
        z.postaviPlatu(z.uzmiPlatu() * 1.2);
        int uspeh = z.sacuvajBazaPodataka();
        if(uspeh == 0)
            System.out.println("Информације су успешно сачуване у бази
                                    података.");
        else
            System.out.println("Информације нису успешно сачуване у бази
                                    података.");
    }
}
```

ПРИМЕР 19

- Креирати класе за рад са са подацима о запосленима, где ће функционалност која подржава упис информација о запосленом бити издвојена у посебан део.

ПРИМЕР 19 (2)

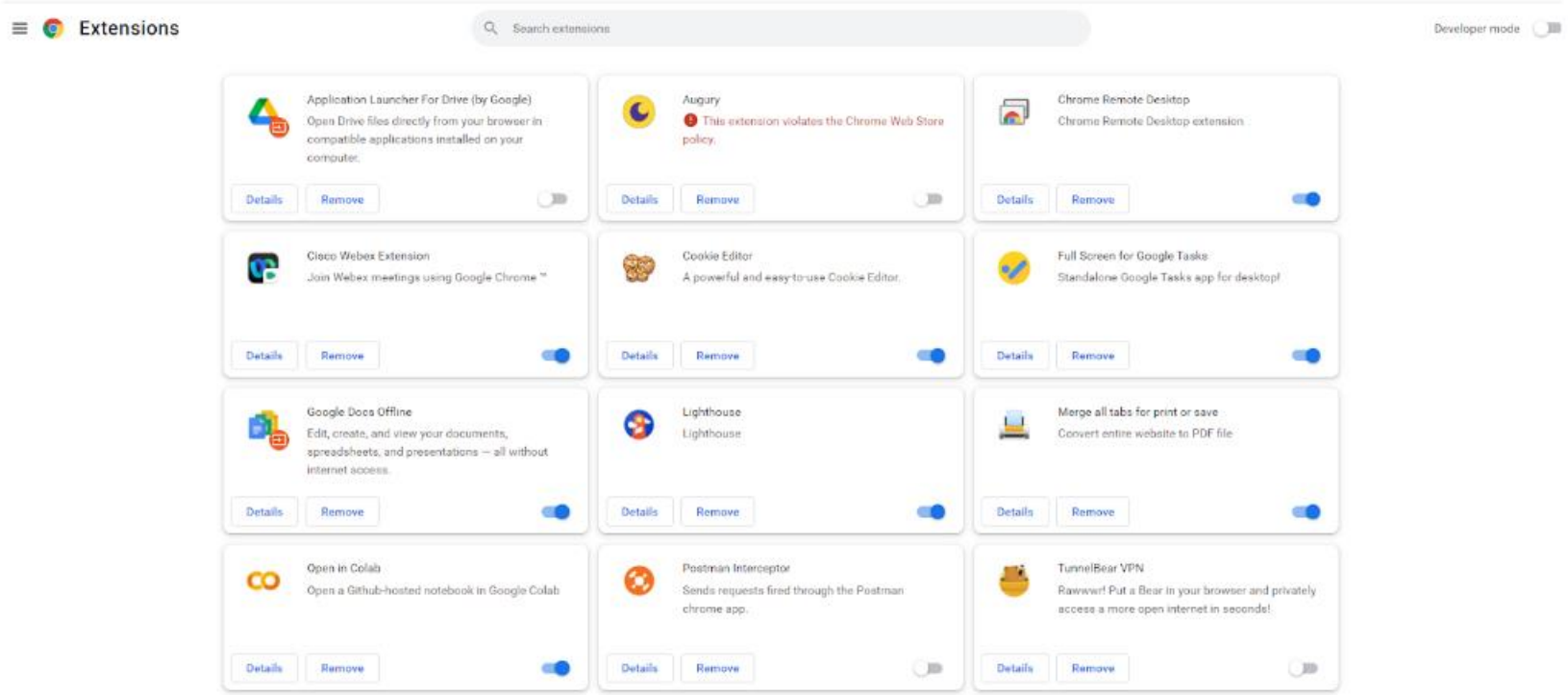
```
public class DobarPrincipS {

    public static void main(String[] args) throws CloneNotSupportedException{
        Zaposleni z = new Zaposleni("Јован", "Петровић", "припрправник",
                                    30_000);
        System.out.println("Запослени: " + z);
        z.postaviPlatu(z.uzmiPlatu() * 1.2);
        int uspeh = ZaposleniBazaPodataka.sacuvaj(z);
        if(uspeh == 0)
            System.out.println("Информације су успешно сачуване у бази
                                    података.");
        else
            System.out.println("Информације нису успешно сачуване у бази
                                    података.");
    }
}
```

O - ПРИНЦИП ОТВОРЕНОСТИ И ЗАТВОРЕНОСТИ

- Софтверске компонете треба да буду отворене за проширивање, али затворене за модификацију.
- По овом принципу, класа треба да буде структуирана тако да реализује своје задатке без претпостављања да ће у будућности бити мењано њено понашање.
- Истовремено, треба да буде омогућено проширивање функционалности класе, на неки од начина композиције, садржавања, наслеђивања и имплементације.
- На пример, модерни веб прегледачи (као што су Google Chrome, Mozilla Firefox, Microsoft Edge и сл.) су дизајнирани тако да омогућавају разноврсна проширења, која кад се укључе помажу у раду са одређеним типовима ресурса на вебу, али њихово укључивање и искучивње не омета основну функцију прегледача.

О - ПРИНЦИП ОТВОРЕНОСТИ И ЗАТВОРЕНОСТИ (2)

- Веб прегледачи су, дакле, дизајнирани тако да су затворени за модификацију али омогућавају проширења, па се може рећи да њихов дизајн на овом ниову поштује принцип отворености и затворености.



ПРИМЕР 20

- Креирати Јава програм за одређивање површне геометријских објеката (кругова и правоугаоника).

ПРИМЕР 20 (2)

```
public static double povrsinaPravougaonika(Pravougaonik p){
    return p.uzmiSirinu() * p.uzmiVisinu();
}

public static double povrsinaKruga(Krug k) {
    return PI * k.uzmiPoluprecnik() * k.uzmiPoluprecnik();
}

public static void main(String[] argumenti){
    Scanner sc = new Scanner(in);
    out.printf("Унесите полупречник круга: ");
    double r = sc.nextDouble();
    Krug k = new Krug(r);
    out.printf("Површина круга је: %f\n", povrsinaKruga(k));
    out.printf("Унесите ширину и висину правоугаоника: ");
    double a = sc.nextDouble();
    double b = sc.nextDouble();
    Pravougaonik p = new Pravougaonik(a,b);
    out.printf("Површина правоугаоника је: %f\n", povrsinaPravougaonika(p));
    sc.close();
}
```

ПРИМЕР 21

- Креирати Јава програм за одређивање површне геометријских објеката (кругова и правоугаоника), водећи рачуна о потреби његовог проширења.

ПРИМЕР 21 (2)

```
Mera[] oblici = new Mera[2];
Scanner sc = new Scanner(in);
out.printf("Унесите полупречник круга: ");
double r = sc.nextDouble();
oblici[0] = new Krug(r);
out.printf("Унесите ширину и висину правоугаоника: ");
double a = sc.nextDouble();
double b = sc.nextDouble();
oblici[1] = new Pravougaonik(a,b);
sc.close();
for(Mera o: oblici){
  out.printf("Површина oblika '%s': %f\n",o.getClass().getSimpleName(),
                o.povrsina());
}
```

L - ПРИНЦИП ЗАМЕНЉИВОСТИ

- Принцип заменљивости је добио име по научници Барбари Лисков.
- Он гласи: методе које користе референце за надкласе/интерфејсе морају да буду у могућности да успешно користе примерке подкласа/класа-имплеменација а да при том ни не знају о шта тачно користе.
- Укратко: класа мора реализовати све уговоре својих надкласа - и на нивоу синтаксе и на нивоу семантике.

ПРИМЕР 22

- Развити класе за правоу.гаоник и за квадрат које обезбеђују рачунање њихових површина

ПРИМЕР 22 (2)

```
public class Kvadrat extends Pravougaonik {

  public Kvadrat(double ivica) { super(ivica, ivica); }

  public Kvadrat() { this(2); }

  @Override
  public void postaviSirinu(double sirina) {
      super.postaviSirinu(sirina);
      super.postaviVisinu(sirina);
  }

  @Override
  public void postaviVisinu(double visina) {
      super.postaviVisinu(visina);
      super.postaviSirinu(visina);
  }
}
```

ПРИМЕР 22 (3)

```
public static void povrsinaProvera(Pravougaonik p) {
      p.postaviSirinu(3);
      p.postaviVisinu(2);
      double povrsina = p.povrsina();
      if(Double.compare(povrsina, 6.0) == 0)
        out.printf(p.getClass().getSimpleName()
            + ": Све ОК! Израчуната површина је %f%n", povrsina);
      else
        err.printf(p.getClass().getSimpleName()

            + ": Проблем! Израчуната површина је %f, треба да буде %f%n",
                        povrsina, 6.0);
  }

  public static void main(String[] argumenti){
      Pravougaonik p = new Pravougaonik();
      povrsinaProvera(p);
      Kvadrat k = new Kvadrat();
      povrsinaProvera(k);
  }
```

I - ПРИНЦИП РАЗДВАЈАЊА ИНТЕРФЕЈСА

- Фаворизује се дизајн са већим бројем малих интерфејса који зависе од клијента у односу на велике, опште, монолитне интерфејсе.
- Другим речима, клијенти не треба да буду приморани да имплементирају непотребне методе, тј. методе које неће користити.

ПРИМЕР 23

- Развити интерфејсе и класе за софтверски ситем који подржава рад ресторана и овди рачуна о прихватању поруџбина (лично, телефонско, он-лајн) и плаћању (лично, он-лајн).

ПРИМЕР 23 (2)

```
public interface Restoran{
	public void prihvatiOnLajnPorudzbinu();
	public void prihvatiTelefonskuPorudzbinu();
	public void platiOnLajn();
	public void staniURedZaLicnuPorudzbinu();
	public void platiLicno();
}
```

ПРИМЕР 23 (3)

```
public class OnLajnKlijentOpsluzivanje implements Restoran {
  @Override
  public void prihvatiOnLajnPorudzbinu() {
      out.println("Реализује се постављање он-лајн поруџбине!");
  }

  @Override
  public void prihvatiTelefonskuPorudzbinu() {
      err.println("Није могуће прихватити телефонску поруџбину за он-лајн клијента!");
  }

  @Override
  public void platiOnLajn() {
      out.println("Реализује се он-лајн плаћање за он-лајн клијента!");
  }

  @Override
  public void staniURedZaLicnuPorudzbinu() {
      err.println("Није могуће стати у ред личних поруџбина за он-лајн клијента!");
  }

  @Override
  public void platiLicno() {
      err.println("Није могуће лично платити за он-лајн клијента!");
  }
}
```

ПРИМЕР 24

- Развити интерфејсе и класе за софтверски ситем који подржава рад ресторана и овди рачуна о прихватању поруџбина (лично, телефонско, он-лајн) и плаћању (лично, он-лајн), тако да буде уважен принцип раздвајања интерфејса.

ПРИМЕР 24 (2)

```
public interface Porudzbina {
  public void prihvatiPorudzbinu();
}

public interface Placanje {
  public void platiPorudzbinu();
}

public class OnLajnPorudzbina implements Porudzbina {
  @Override
  public void prihvatiPorudzbinu() {
      out.println("Realizuje se postavljanje on-lajn poruzbine!");
  }
}

public class OnLajnPlacanje implements Placanje {

  @Override
  public void platiPorudzbinu() {
      out.println("Realizuje se on-lajn placanje za narudzbinu!");
  }
}
```

ПРИМЕР 24 (3)

```
public class OnLajnKlijent {
  private String ime;
  private String adresa;
  private String email;
  private Porudzbina porudzbina;
  private Placanje placanje;

  public OnLajnKlijent(String ime, String adresa, String email, Porudzbina porudzbina,
                       Placanje placanje) {
      this.ime = ime;
      this.adresa = adresa;
      this.email = email;
      this.porudzbina = porudzbina;
      this.placanje = placanje;
  }

  public Porudzbina getPorudzbina() { return porudzbina; }

  public void setPorudzbina(Porudzbina porudzbina) { this.porudzbina = porudzbina; }

  public Placanje getPlacanje() { return placanje; }

  public void setPlacanje(Placanje placanje) { this.placanje = placanje; }
}
```

D - ПРИНЦИП ИНВЕРЗИЈЕ ЗАВИСНОСТИ

- Треба зависити од апстрактција, а не од конкретне реализације.
- Модули вишег нивоа никако не треба да зависе од модула нижег нивоа, већ и модули нижег нивоа и модули вишег нивоа треба да зависе од апстракција.
- На пример, у процесу плаћања кореди̇ном картицом, само процесирање кредитних картица не зависи од типа кредитне картице.
- Мало конкретније: сваки пут када се у оквиру неког метода класе A креира примерак класе B, тада је оформљена зависност између класе A и класе B.
- Принцип инверзије зависности захтева да, уместо да класа A захтева креирање објекта дате класе, она би требала да захтева апстракцију објекта.

ПРИМЕР 25

- Развити класе за клијент-сервер систем,
  где клијент приликом реализације датог метода користи метод сервера.

ПРИМЕР 25 (2)

```
public class ServisB {
  public String getInfo() {
      return "Информације о сервису ServisB";
  }
}

public class KlijentA {
  ServisB servis = new ServisB();

  public void uradiNesto() {
      String info = servis.getInfo();
      out.println("KlijentA - " + info);
  }
}

public class LosPrincipD {

  public static void main(String[] args){
      KlijentA ka = new KlijentA();
      ka.uradiNesto();
  }
}
```

ПРИМЕР 26

- Развити интерфејсе и класе за клијент-сервер систем, где клијент приликом реализације датог метода користи метод сервера, водећи рачуна о принцицпу инверзије зависности.

ПРИМЕР 26 (2)

```
public interface Servis {
  String getInfo();
}

public class ServisB implements Servis {

  @Override
  public String getInfo() {
      return "Информације о сервису ServisB";
  }
}

public class ServisC implements Servis {

  @Override
  public String getInfo() {
      return "Информације о сервису ServisC";
  }
}
```

ПРИМЕР 26 (3)

```
public interface Klijent {
  void uradiNesto();
}

public class KlijentA implements Klijent {
  private Servis servis;

  public KlijentA(Servis servis) {
      this.servis = servis;
  }

  @Override
  public void uradiNesto() {
      String info = servis.getInfo();
      out.println("KlijentA - " + info);
  }
}
```

```
Servis sB = new ServisB();
Servis sC = new ServisC();
Servis sD = new ServisD();

Klijent kA = new KlijentA(sB);
kA.uradiNesto();

kA = new KlijentA(sC);
kA.uradiNesto();

kA = new KlijentA(sD);
kA.uradiNesto();
```

ПИТАЊА И ЗАДАЦИ

- Биће накнадно додато…