Класе
комбинација презентације (Хортона) и к'о бајаги скрипте (Ово је прича коју врло радо причам)

нови тип података: дефинишу могуће вредности података и операције над њима
Дефиниција класе садржи само **атрибуте** (променљиве) и **методе** (функције)
Атрибути описују како изгледа сваки објекат класе, методи – шта са њим може да се ради

атрибути:

1) произвољног
   примитивног (`byte`, `short`, `char`, `int`, `long`, `float`, `double`, `boolean`) или
   референтног типа (низ, класа, интерфејс; укључујући и тип класе која се управо дефинише)
2)

статички атрибут

- само један примерак променљиве постоји у меморији све време
- односи се на класу као целину, не на њене појединачне објекте
- добар пример: бројач креираних објеката
- дефинисање статичког атрибута:

```
private static <tip> <ime>; // automatska inicijalizacija podrazumevanom vrednoscu
                            // koja zavisi od tipa (kao za elemente nizova)
private static <tip> <ime> = <inicijalna_vrednost>;
```

`private` – енкапсулација, скривање детаља имплементације од других класа:
само кôду ове класе допуштено је да директно приступа вредности атрибута и мења је.

Конкретан пример:

```
private static int brojac; // automatski se inicijalizuje na 0 (jer je tipa int)
```

Уколико се, пак, ради о константи:

```
public static final <tip> <IME> = <konstantna_vrednost>;
```

`public` – јавно доступна, `final` – константна, не може јој се променити вредност

Конкретан пример:

```
public static final double PI = 3.14; // samo primer kako se definise konstanta,
                                      // a kada nam treba bas pi koristicemo Math.PI
```

нестатички атрибут (инстанцна променљива)

- сваки објекат има свој примерак сваког од нестатичких атрибута из дефиниције класе са вредношћу која је независна од вредности тог атрибута у другим објектима.
  Нпр. свака тачка у 2D има своју `x` и своју `y` координату, чије вредности су независне од вредности `x` и `y` координата других тачака.
- Нестатички атрибути дају објектима индивидуалност.
- Дефинисање нестатичког атрибута:

```
private <tip> <ime>; // automatska inicijalizacija podrazumevanom vrednoscu
                     // koja zavisi od tipa (kao za elemente nizova)
private <tip> <ime> = <inicijalna_vrednost>;
```

Конкретни примери:

```
private double x; // automatski se inicijalizuje na 0
private Tacka[] temena; // automatski se inicijalizuje vrednoscu null
private Osoba roditelj; // automatski se inicijalizuje vrednoscu null
private Osoba[] deca; // automatski se inicijalizuje vrednoscu null
```

методи:

- локалне променљиве метода се не иницијализују аутоматски, то се мора урадити експлицитно
- аргументи се преносе по вредности (преносе се локалне копије) →
  метод не може трајно променити вредност свог аргумента примитивног типа
  док садржај на који реферише аргумент референтног типа може,
  јер се преноси копија референце, а не самог објекта на који она реферише
- `final` параметри – спречава се да метод промени њихову вредност
  (има смисла само за аргументе референтних типова,
  али спречава се промена референце која се прослеђује методу,
  не и самог објекта на који она реферише)

статички методи

- могу се извршавати независно од постојања конкретних објеката класе
- не могу користити нестатичке атрибуте класе (без референсања на конкретан објекат)
- добри примери: функције опште намене, нпр. математичке функције из `java.lang.Math`

- дефинисање статичког метода:

```
public static <povratni_tip> <ime_metoda> (<lista_argumenata>){<naredbe>}
```

`<povratni_tip>` може бити `void`, произвољни примитивни или референтни тип

Пример: `main()`

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

- позивање изван класе у којој је дефинисан

```
<ime_klase>.<ime_metoda>(<argumenti>)
```

Конкретно:

```
double korenIzPi = Math.sqrt(Math.PI);
```

Уколико се изврши import статичких чланова класе, може им се приступати и само навођењем њиховог имена, нпр.

```
import static java.lang.Math.*; // import svih statickih clanova klase Math
     // import static java.lang.Math.PI;   // import samo statickog clana PI
     // import static java.lang.Math.sqrt; // import samo statickog clana sqrt
double korenIzPi = sqrt(PI);
```

Импорт статичких чланова класе није могућ за класе које се налазе у подразумеваном (default, безименом) пакету – пакет мора имати име!

- позивање унутар класе у којој је дефинисан (било где – из статичког или нестатичког кôда):

```
<ime_metoda>(<argumenti>)
```

нестатички методи

- могу се извршавати само за конкретне објекте класе
- могу приступати произвољном атрибуту класе (и статичком и нестатичком) навођењем само његовог имена

- дефинисање: као за статички, само без кључне речи `static`
- позивање изван класе у којој су дефинисани:

```
<referenca_na_objekat>.<ime_metoda>(<argumenti>)
```

Конкретан пример:

```
jedinicnaSfera.zapremina()
```

- позивање унутар класе у којој су дефинисани (унутар нестатичких метода):

```
<ime_metoda>(<argumenti>)
      // podrazumeva se referenca na tekuci objekat, this:
      // this.<ime_metoda>(<argumenti>)
```

Конкретан пример:

```
public double zapreminaPrizme(){
      return povrsinaOsnove()*visina;
}
```

Променљива `this`

унутар нестатичког метода представља референцу на **текући објекат**
(текући објекат је конкретан објекат за који је нестатички метод позван)
приликом обраћања нестатичким члановима (атрибутима и методима) класе
навођењем само њиховог имена, имплицитно се подразумева коришћење ове референце:

```
x ⇔ this.x
povrsinaOsnove() ⇔ this.povrsinaOsnove()
```

Нпр. у случају позива

```
double zapreminaJedinicneSfere = jedinicnaSfera.zapremina();
```

`this` унутар метода `zapremina()` референсаће на (текући) објекат `jedinicnaSfera`,
док ће нестатички атрибут `radius` имати вредност полупречника баш тог објекта (`jedinicnaSfera`) за који је метод позван пошто се `this` подразумева: `this.radius`.

Захваљујући променљивој `this`, метод "зна" који је текући објекат са којим ради.

```
// metod racuna i vraca zapreminu tekuce sfere
public double zapremina(){
   return radius * radius * radius * 4./3 * Math.PI;
}
```

`radius` ⇔ `this.radius` → „полупречник ове (текуће) сфере".
Када се рачуна запремина текуће сфере, треба нам полупречник баш те, текуће, сфере.

Постоје и ситуације када је **неопходно експлицитно писати `this`**:

- Имена променљивих које се декларишу унутар метода, локална су за тај метод.
- Дозвољено је користити имена локалних променљивих која су иста као и имена атрибута.
  У том случају, неопходно је користити `this` за референсање члана класе унутар метода.
  Сâмо име променљиве ће се увек односити на локалну променљиву метода, не на истоимени атрибут класе.
  Пример:

```
public void setRadius(double radius){
      this.radius = radius;
            // radius - argument metoda
            // this.radius - istoimeni atribut klase
}
```

Иницијализациони блокови (статички и нестатички) – о томе ћемо касније.

Потпис метода и overloading (преклапање) метода

- **Overloading:** у класи можемо имати већи број метода који се исто зову, али који имају различит „потпис".
- Два метода имају исти потпис ако се:
  - исто зову
  - имају исти број параметара
  - одговарајући парови параметара су им истих типова

  За потпис метода није битан његов повратни тип, као ни имена параметара.

  Пример:
  методи

```
String obrada(int a, String b){…}
double obrada(int c, String d){…}
```

  имају исти потпис јер се исто зову, имају по 2 параметра, први параметар је у оба метода типа `int`, а други је у оба метода типа `String`. То што први метод враћа `String`, а други `double` и што се први параметар у првом методу зове `a`, а у другом `c`, и други параметар првог метода се зове `b`, а другог `d`, за потпис метода није од значаја.

- Потписи свих метода класе морају да се разликују да би компајлер на основу позива метода могао тачно да одреди који метод ми желимо да позовемо.

Конструктори

Ово је потребно „добро утувити“ ☺, а нарочито га се сетити кад Вам нпр. програм пукне због избацивања изузетка типа `NullPointerException`:

Нема креирања објекта без позива конструктора!

Конструктори су посебна врста нестатичких метода.

- Конструктор се увек зове као и класа којој припада.
- Нема повратни тип, чак ни `void`!
- **Примарна сврха конструктора је да за објекат који се управо креира изврши иницијализацију нестатичких атрибута.**
- По потреби може да мења и вредности статичких.

Прича о преклапању метода важи и за конструкторе:
у класи је могуће имати већи број конструктора (сви се исто зову, као и класа), под условом да никоја два међу њима немају исти потпис.

Конструктори могу имати произвољан број аргумената.

Подразумевани конструктор

- Конструктор се назива подразумеваним ако нема аргументе.
- Подразумевани конструктор обично прави репрезентативни примерак класе
  (тачку која представља координатни почетак, јединичну сферу са центром у координатном почетку, банковни рачун са иницијалним стањем једнаким трошковима месечног одржавања итд.)
  Пример:

```
public Sfera(){
      // kreira jedinicnu sferu sa centrom u koordinatnom pocetku
      radius = 1;
      brojac++;
}
```

  Ако сами експлицитно не напишемо ниједан конструктор, компајлер ће аутоматски генерисати подразумевани конструктор са празним телом (важно касније код наслеђивања):

```
public Sfera(){}
```

  Када сами напишемо бар један конструктор, компајлер не генерише подразумевани са празним телом.

Позивање једног конструктора из другог конструктора исте класе

- Унутар једног конструктора можемо позвати неки други конструктор исте класе коришћењем кључне речи `this` уместо имена метода и то мора бити прва наредба у телу конструктора
  Пример:

```
public Sfera(double x, double y, double z, double radius){
      this(x, y, z); // poziv donjeg konstruktora
            // postavi radius na 1, uveca brojac,
            // postavi x, y i z na zadate vrednosti
      this.radius = radius; // postavljanje poluprecnika na zadatu vrednost
}

public Sfera(double x, double y, double z){
      this(); /* postavi nam radius na 1, i uveca brojac */
              /* this za poziv konstruktora */
      this.x=x; /* this za pristup istoimenom atributu */
      this.y=y;
      this.z=z;
}
```

- Дефиниција тог другог конструктора којег позивамо помоћу кључне речи `this` може се у дефиницији класе налазити и испод дефиниције конструктора у коме је тај позив.

Копи-конструктор

- креира објекат који је, у тренутку свог креирања, идентична копија постојећег објекта класе (у тренутку креирања копије, вредности нестатичких атрибута објекта и његове копије су идентичне)
- прима један аргумент типа класе у којој се налази
- објекат и његова копија су два независна објекта и „живе“ своје одвојене животе: промене вредности атрибута једног не утичу на промене вредности одговарајућих атрибута другог.

Пример:

```
public Sfera(final Sfera s){
      x=s.x;
      y=s.y;
      z=s.z;
      radius=s.radius;
      brojac++;
}
```

или

```
public Sfera(final Sfera s){
      this(s.x, s.y, s.z, s.radius); // poziv konstruktora sa 4 argumenta
            // prekopiraju se vrednosti (nestatickih) atributa objekta s
            // i uveca staticki atribut brojac
}
```

Пакети

- Свака класа налази се у неком пакету (не препоручује се употреба подразумеваног, безименог пакета. Један од разлога је већ наведен: није могуће импортовати статичке чланице класа које се налазе у безименом пакету).
- Пакет представља јединствено именовану колекцију класа.
- У једном пакету треба да се налазе класе које су на неки начин повезане.
- Пакетом је одређено која имена можемо користити за класе: унутар једног пакета можемо давати имена класама без бриге да ли та имена већ постоје у другим пакетима:
  имена класа из једног пакета неће се мешати са именима класа из других пакета јер Јава третира име пакета као део имена класе.
  Пуно квалификовано име класе `String` из пакета `java.lang` је `java.lang.String`
- Ако бисмо у неком нашем пакету дефинисали своју класу са именом `String`, унутар тог пакета коришћење имена `String` односило би се на ту нашу класу, док бисмо се стандардној класи `String` морали обраћати са `java.lang.String`
- Име пакета мора да одражава структуру директоријума на хард-диску, нпр. у случају пакета `java.lang`, `lang` је поддиректоријум од `java`.

Приступни атрибути (контрола приступа)

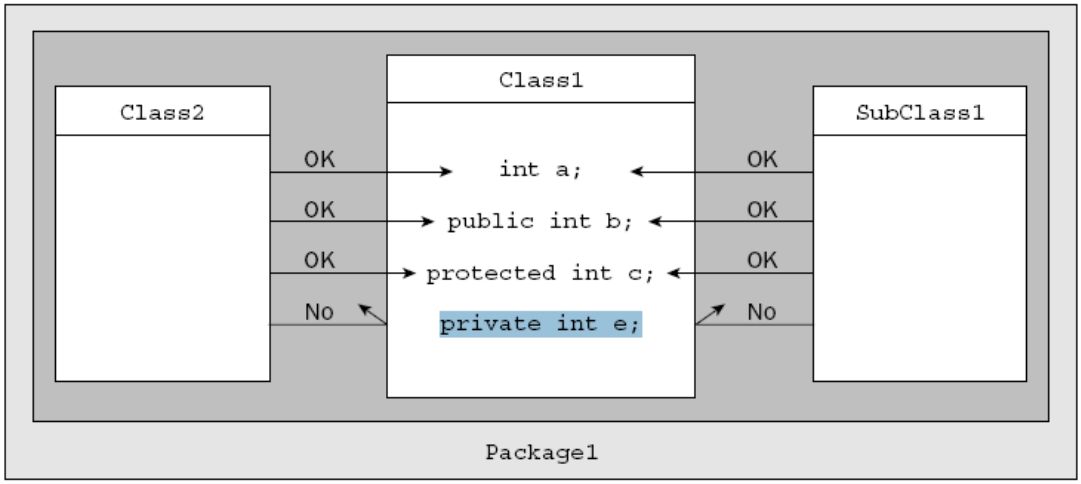
- за класе
  - `public` – јавно право приступа
  - – (ништа, без `public`) пакетно право приступа

  Одређују расположивост имена класе другим класама да га користе као тип променљивих и параметара метода.
  Име класе са пакетним правом приступа доступно је само другим класама из истог пакета.
  Име класе са јавним правом приступа, доступно је и класама из других пакета.

- за чланове (атрибуте и методе) класе
  - `public` – допуштен приступ из метода произвољне класе (не нужно из истог пакета, ако је и класа чији је ово члан декларисана као `public`)
  - `protected` – допуштен приступ из метода произвољне класе истог пакета и из произвољне потклассе (не нужно из истог пакета)
  - без приступног атрибута – допуштен приступ из метода произвољне класе из истог пакета
  - `private` – доступан само из метода унутар класе. Никакав приступ изван класе.

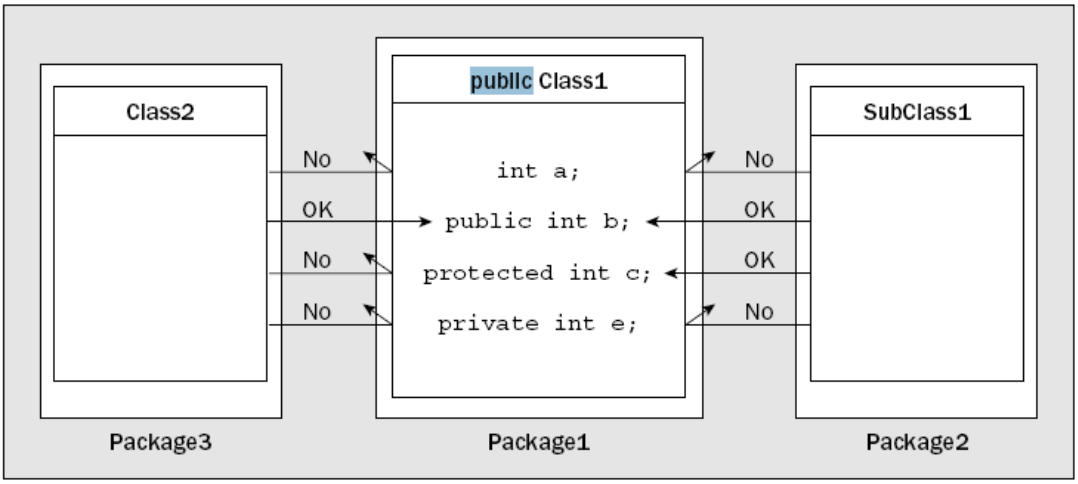
  Приступни атрибути сложени су растуће по рестриктивности. Најмање је рестриктиван `public`, а највише `private`.

Илустрација, све три класе су у истом пакету (Package1):



Закључак: из других класа истог пакета није могућ приступ само `private` члану.

Илустрација, све класе су у различитим пакетима:



Битно је што је класа Class1 дефинисана као `public`.

- Из поткласе Subclass1, која је у другом пакету, није могуће приступити члановима са пакетним правом приступа, као ни `private` члановима.
- Класа Class2, из другог пакета, и која није изведена из Class1, има приступ само јавним члановима класе Class1.

Избор приступних атрибута

- Најчешће, атрибути у `public` класи треба да буду `private`, а методи који ће се позивати изван класе `public`.
- Изузеци:
  - за класе које нису `public`, а тиме нису доступне изван пакета, конвенција је дозволити другим класама пакета директан приступ атрибутима (дефинисати атрибуте са пакетним правом приступа)
  - ако имамо `final` атрибуте, који су вероватно корисни изван класе, можемо их декларисати као `public`
  - методе класе, које ће интерно користити само други методи исте класе треба дефинисати као `private`
  - у класама попут `Math`, која је само контејнер за корисне функције и стандардне вредности података, све треба да буде `public`

get*() методи (приступни, accessor методи)

- како ће нам атрибути класа бити `private`, то кôд ниједне друге класе неће моћи директно да приступа њиховим вредностима и да их мења.
- ако желимо да допустимо да кôд неке друге класе може да прочита вредност нашег `private` атрибута, написаћемо одговарајући `public` метод `get*()`, који служи само да дохвати вредност тог атрибута
- ако се ради о нестатичком атрибуту, одговарајући метод `get*()` биће такође нестатички
- метод `get*()` за статички атрибут биће такође статички
- `get*()` нема аргументе, а повратни тип му је исти као тип атрибута чију вредност дохвата
- у телу мора имати наредбу `return` којом враћа вредност атрибута

Примери:

```
public double getX(){
	return x; // gde je x atribut tipa double
}

public static int getBrojac(){
	return brojac; // brojac je staticki atribut tipa int
}
```

set*() методи (мутатор методи)

- Ако желимо да допустимо да кôд неке друге класе може да промени вредност нашег `private` атрибута, написаћемо одговарајући `public` метод `set*()`, који служи само да промени вредност тог атрибута на задату вредност.
- Могуће је вршити проверу нове вредности и спречити придруживање неодговарајуће вредности атрибуту
- ако се ради о нестатичком атрибуту, одговарајући метод `set*()` биће такође нестатички
- метод `set*()` за статички атрибут биће такође статички
- повратни тип метода `set*()` је `void`, а метод има један аргумент чији је тип исти као тип атрибута чију вредност мења
- у телу има наредбу доделе којом се вредност атрибута поставља на задату
  (ако се врши провера и нова вредност није валидна, то не мора бити случај)

Примери:

```
public void setX(double x){
	this.x=x;  // x je atribut tipa double u klasi Sfera
}

/*
 * ovaj nemamo u klasi Sfera
 * jer nam nije bio potreban
 */
public static void setBrojac(int b){
	brojac=b; // brojac je staticki atribut tipa int
}
/*
 * kod ovog metoda bi bilo malo problema ako bi se
 * argument umesto b zvao brojac
 * jer u statickom metodu ne moze se koristi kljucna
 * rec this. Onda bismo statickom clanu brojac klase
 * Sfera morali da se obratimo sa Sfera.brojac
 * dok bi nam samo brojac bio istoimeni argument
 * metoda
 */
```

Eclipse може аутоматски генерисати потребне методе get*() и set*(): Source, Generate Getters and Setters..., штиклирају се жељени методи. OK. Методи бивају генерисани испод позиције на којој је курсор.

`public String toString()`

- Уколико у својој класи, нпр. `Sfera`, напишемо дефиницију метода са горњим прототипом, он ће бити аутоматски позиван у ситуацијама типа
  - `System.out.println(s)`, где је у `s` референца на објекат класе `Sfera`,
    ⇔ `System.out.println(s.toString())`
  - као и када је један од операнада оператора сабирања типа `String`, а други типа `Sfera`.
- Метод пишемо тако да врати `String`-репрезентацију текућег објекта у облику који ми желимо.

Рад са објектима (примери су из тест-класе за рад са сферама)
Креирање објекта класе (мора постојати позив конструктора!)

Променљива типа класе и објекат типа те класе нису исто!
Променљива МОЖЕ да чува РЕФЕРЕНЦУ на конкретан објекат.
Када само декларишемо променљиву типа класе, како нема позива конструктора, нема ни креирања објекта (речено је раније: нема креирања објекта без позива конструктора!).
Нпр.

```
Sfera lopta; // neinicijalizovana promenljiva
```

Покушај употребе неиницијализоване променљиве, нпр. `lopta.zapremina()`,
доводи до грешке при компајлирању.

```
lopta = null;
```

На овом месту променљива `lopta` добије специјалну вредност `null` – не реферише ни на један конкретан објекат класе `Sfera`.
Да бисмо креирали нови објекат класе, морамо да користимо кључну реч `new` коју прати позив одговарајућег конструктора, нпр.

```
new Sfera()
```

Горњим изразом се издвоји меморија неопходна за смештање објекта класе `Sfera`, тј. креира се по један примерак сваког од нестатичких атрибута класе: `x`, `y`, `z` и `radius`, извршри се тело конструктора да би се они иницијализовали (тело овог конструктора још и увећа онај статички атрибут `brojac`) и резултат је референца на тако добијени објекат.

Ту референцу можемо да сачувамо у променљивој типа `Sfera`, коју смо претходно декларисали:

```
s=new Sfera(); // moglo je i sve u jednom redu: Sfera s = new Sfera();
```

Овим смо успоставили везу између променљиве `s` и управо креираног објекта класе `Sfera`.
Сада се у променљивој `s` налази референца на тај објекат.
Ако желимо да раскинемо ту везу, можемо да кажемо:

```
s=null;
```

Сада променљива `s` није у вези ни са једним објектом (не показује ни на шта).
Или смо могли да је повежемо са неким другим објектом:

```
s=new Sfera(1,1,1,17);
```

(ово можемо ако у класи постоји конструктор са 4 аргумента)

Ако имамо и овако нешто:

```
Sfera t=new Sfera(0,0,0,5);
```

па кажемо:

```
s=t;
```

тиме смо оно што пише у `t` (а то је референца на сферу са центром у координатном почетку и полупречника 5) преписали у променљиву `s`, па сада имамо две променљиве (`s` и `t`) које обе реферишу на исти објекат у меморији. Нема позива конструктора, те нема креирања новог објекта - вредност статичког бројача остаје непромењена.

Ако након тога, користећи променљиву `s`, променимо полупречник објекта на који она реферише на 7, објекат на који реферишу и променљива `s` и променљива `t` биће сфера са центром у координатном почетку и полупречника 7.

Ако имамо сферу са центром у тачки (1, 2, 3) и полупречника 5:

```
s = new Sfera(1, 2, 3, 5);
```

па направимо њену копију, користећи копи-конструктор:

```
t = new Sfera(s);
```

добићемо нову сферу (позива се конструктор, статички бројач креираних објеката се увећава) која ће имати центар такође у тачки (1, 2, 3) и полупречник ће јој бити исто 5 као у оној полазној.

Али нова сфера и полазна сфера су два независна објекта.
Ако након креирања сфере-копије, `t`, променимо полупречник полазне сфере, `s`, на 7 (био је 5):

```
s.setRadius(7);
```

полупречник сфере-копије неће се променити (остаће и даље 5) – она је идентична копија објекта од ког је настала, али у тренутку свог креирања. Касније се та два објекта мењају независно један од другог.

Животни век објеката и Garbage Collector

Уништавање „мртвих“ објеката, тј. објеката на које више не реферише ниједна променљива у програму, аутоматски врши тзв. Garbage Collector. Иако аутоматски, то се не дешава увек и моментално, али у случају да не пишемо претерано захтевне програме (у погледу потребне меморије) не морамо о томе да бринемо - посао који за нас ради Garbage Collector је довољно добар.

До ситуације да променљива више не реферише на објекат долази се при изласку из опсега променљиве (опсег променљиве је од места на коме је декларисана до краја блока у коме је декларација променљиве) или када се променљивој, која наставља да постоји, додели специјална вредност `null` или пак референца на неки други конкретан објекат класе.

Могуће је да је атрибут неког другог објекта променљива која чува референцу. У том случају, референсани објекат је „жив“ бар колико и објекат чији атрибут реферише на њега.