Објектно оријентисано програмирање

Јован Самарџић, 13/2019

Професор: Александар Картељ

- дефиниције
- ознаке
- теореме
- докази
- примери

Година курса: 2020/21

Молим да ми све грешке пријавите преко мејла или друштвених мрежа.

1. Карактеристике Јаве и типови апликација

Карактеристике програмског језика Java:

1. Једноставан, објектно оријентисан и фамилијаран;

- нема управљања меморијом;
- садржи готове библиотеке за разне намене;
- сличан (синтаксно) C/C++.

2. Робустан и сигуран;

- добар компајлер (и интерпретер) ⇒ тешко се праве вируси ⇒ безбедан;
- као што смо рекли: нема показивача (ни њихове аритметике) ⇒ сигуран.

3. Архитектонски неутралан и преносив;

- пола компајлер, пола интерпретер: изворни код —(компајлер)→ бајт код —(интерпретер)→ машински код;
  - изворни код: оно што пишемо
  - бајт код: међукод, арх. неутралан, близак машинском
  - машински код: 0 и 1, није арх. неутралан
- величине простих типова увек исте;
- има исто извршавање на свакој платформи.

4. Перформантан;

- у смислу да може да парира данашњим (модерним) језицима; ипак, често је (благо) спорији од C;
- постоји аутоматски сакупљач отпадака (не морамо сами да деалоцирамо меморију);
- делови кода који успоравају се могу извући и директно превести у машински код.

5. Интерпретиран, вишенитан и динамичан.

- већ смо објаснили шта значи интерпретиран;
- вишенитан: паралелизација;
- динамички: учитава класе само по потреби.

Типови Јава апликација:

1. Апликације из командне линије;

   - не користе графичке компоненте;
   - ток извршавања је линеаран;
   - унос и испис се врше путем команде линије.

2. Апликације са графичким корисничким интерфејсом;

   - панели, прозори, дугмићи, листе...;
   - JavaFX - библиотека која олакшава програмирање GUI.

3. Апликације за мобилне уређаје;

   - за Android, iOS ...
   - нису толико популарне као пре, сада се то ради на друге начине (нпр. Swift)
   - JavaME - библиотека за развој оваквих апликација.

4. Аплети;

   - застарела технологија;
   - служиле су да корисници сами наместе веб странице по свом укусу, јер су тада биле „досадне".

5. Серверске апликације;

   - пример: Када купујемо телефон преко сајта Гигатрона, имамо филтере (бренд, боја, меморија...). Очигледно постоји нешто што повезује наше захтеве и њихову базу производа. Управо то ради серверска апликација.

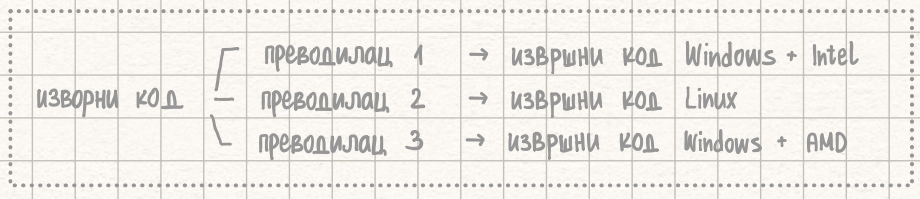
6. Библиотеке.

   - нису програми који служе за извршавање, већ скуп функција.

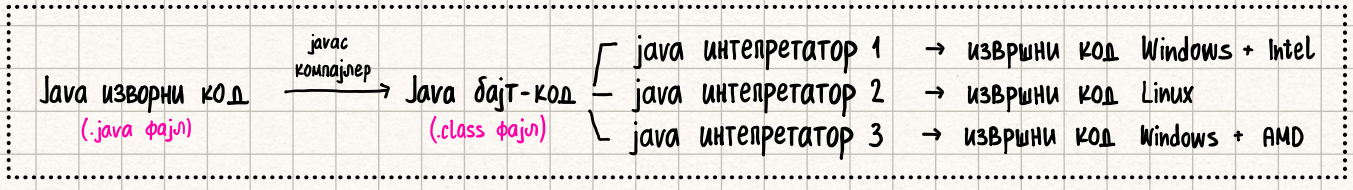
2. Начин извршавања Java програма

Идеја је да опишемо процес креирања извршног кода:

1° традиционални начин: За сваку платформу (процесор, OS...) постоји посебан преводилац.
То значи да за различите рачунаре, ми добијамо различите извршне кодове.



2° Java: Изворни код се, помоћу javac компајлера, доведе до бајт-кода. (исти без обзира на платформу).
Разлике у платформама се превазилазе тек у наредном кораку - интерпретирању (тј. у току извршавања).
За сваку платформу постоји посебан java интерпретатор.



JIT Java компајлер: Додатак интерпретатору који функционише по принципу кеш меморији.
↓
Just-In-Time

Интерпретирање је „лење" (део по део, не све одједном).
Када се нека метода први пут интерпретира, њену деф. запишемо у неку „табелу".
Следећи пут када се та метода позове, ако се већ налази у табели, не морамо је опет интерпретирати.
На овај начин (колико-толико) убрзавамо овај процес.

JVM - Java виртуелна машина: минијатурни виртуелни рачунар који постоји само у меморији.

- У њој се налази интерпретатор ⇒ свака платформа има посебну JVM.

- Садржи: - систем за учитавање класа;
  - подсистем за извршавање; (ово је интерпретатор)
  - област за податке приликом извршавања; (за методе, heap, stack, регистри...)
  - сакупљач отпадака; (Garbage Collector)
  - Java нити.

JDK - Java Development Kit: све ово заједно.

Java API - Application Programming Interface: скуп уграђених класа груписаних у пакете.

Java Core API: java.lang, java.io, java.util, java.net, java.awt, java.swing.

3. Објектно оријентисано програмирање

За разлику од процедуралне програмске парадигме, **објектно оријентисана парадигма** је заснована на појму *објеката*.
Ти објекти садрже податке - *атрибуте*, као и кодове који описују неке активности - *методе*.

Основни појмови:

* **Објекат:** интегрална целина података и процедура за рад са њима;
* **Атрибут:** подаци унутар објекта; зовемо их и поља;
* **Метод:** функција која је саставни део објекта;
* **Порука:** скуп информација који се шаље објекту;
  састоји се из адресе објекта примаоца и самог саопштења (преко „тачка нотације")

* **Класа:** рецепт за креирање објекта, тј. описује структуру објекта;
  није исто што и структура у C (структуре немају ништа налик методама);
* **Инстанца** = објекат. Ово је „живо", са њим радимо.

Пример:

```
public class Automobil{
    //atributi:
    int maxSpeed;
    int maxFuel;
    int currentFuel;
    String modelName

    //metoda:
    void refuel(int fuelLit){
        this.currentFuel += fuelLit;
    }

    public static void main(String[] args){

        Automobil auto1 = new Automobil();
        Automobil auto2 = auto1;

        //poruke:
        auto1.maxFuel = 100;
        auto1.refuel(10);
    }
}
```

(main није метод већ функција, јер извршавање почиње од ту, па не може никако бити придружено ни једном објекту)

(На heap-у се алоцира меморија за објекат auto1 класе Automobil. Оператор new враћа референцу на ту меморијску адресу.)

(објектна референца)

* Потклаcа: класа чије су све инстанце истовремено и инстанце неке друге класе;
* Наткласа: обрнуто.

Потклаcа настаје додавањем нових својстава (атрибута/метода) или модификовањем постојећих.
Тај механизам назива се наслеђивање и њиме формирамо релације између класа:

* Студент «јесте конкретизација» Човек;
* Човек «јесте генерализација» Студент;

Пример:
```
public class SportAutomobil extends Automobil
```

Осим наслеђивања, битан је и механизам садржавања који индукује нове две релације:

* Точак «јесте део од» Аутомобил;
* Аутомобил «садржи» Точак;

Пример:
```
public class Wheel {
    //...
}

public class Automobil{
    //...
    Wheel wheel1;
    //...
}
```
(Уколико не искористимо оператор new, објекат wheel1 ће имати спец. реф. вредност null)

* Java је пример једног објектно оријентисаног програмског језика.
Оно по чему се разликује од нпр. C++ је то што у Јави једна класа не може имати више наткласа.

Java има коренску хијерархију: све класе су потомци класе Object из пакета java.lang.
То значи да ако нема експлицитног наслеђивања, нова класа је потклаcа класе Object.

Већ смо га поменули у [2], али пакет је скуп класа намењених једној врсти посла.

* Апстрактна класа је класа у којој постоји бар један метод који нема дефиницију, већ само декларацију.

Пример:

```
public abstract class Shape{
    //...
    abstract void calculateArea();
}
```

```
public class Rectangle extends Shape{
    //...
    void calculateArea(){
        this.area = this.a * this.b;
    }
}
```

```
public class Circle extends Shape{
    //...
    void calculateArea(){
        this.area = this.r * this.r * Math.PI;
    }
}
```

Апстрактна класа се не може инстанцирати. (јер апстрактни метод има недефинисано понашање)
Поткласе морају да испуне једно од наредног:

1° Декларишемо све апстрактне методе - тада она постаје регуларна класа;
2° Не декларишемо апстрактне методе - тада и она остаје апстрактна.

* Интерфејс је тоталнo апстрактна класа - нема атрибуте, само апстрактне методе и деф. константи.

Користе се када баш ништа не знамо о томе како нешто имплементирати у том тренутку,
али знамо да ће нам сигурно требати.

Ово је начин да делимично компензујемо непостојање вишеструког наслеђивања у Јави.

Пример:

```
public interface Audible{
    void sound();
}
```

```
public class Cat implements Audible{
    public void sound(){
        System.out.println("Meow");
    }
}
```

```
public class Automobil implements Audible{
    public void sound(){
        System.out.println("Vroom");
    }
}
```

* UML = Unified Modeling Language: стандардизовани формат за моделовање софтвера, коришћењем графичких облика и текста.

Постоје разни, али нас занимају дијаграми класа.

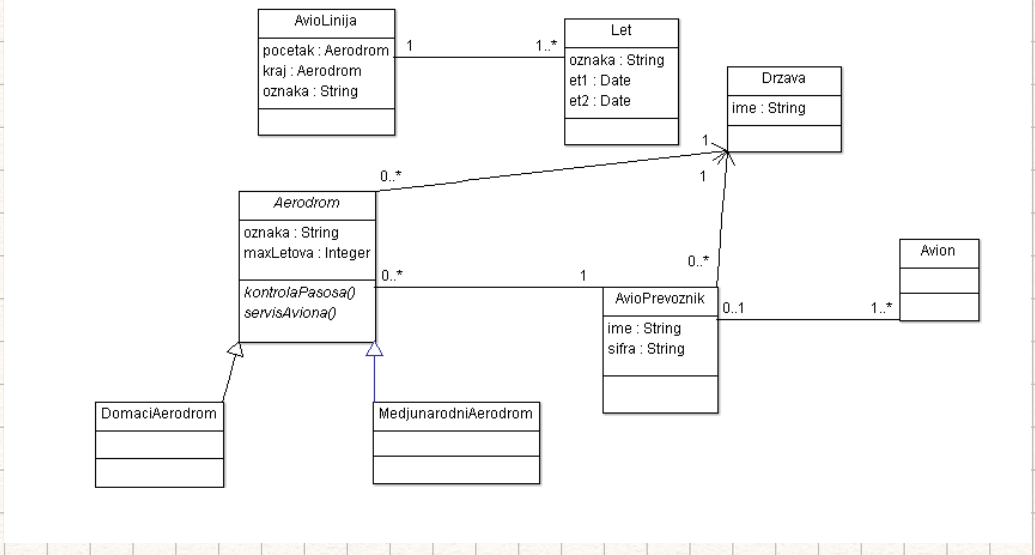
Нотација: *

ImeKlase
атрибути
методе

* потклаcа ⟶ наткласа

* класа ⇢ интерфејс

* садржавање: (*) (*) . где је (*) „арност" (нпр. 1 или 1...*)
↳ 1 или више



Користимо StarUML алат.

4. Елементарне конструкције језика Јава

Већ смо поменули да ми кад програмирамо, ми заправо пишемо изворни код.
То је неки низ знакова који се прослеђује преводиоцу, који то анализира.

Фазе анализе при превођењу:

1) **лексичка анализа:** проверава лексеме („реч по реч", не мора да има смисла као целина);

2) **синтаксна анализа:** проверава да ли су лексеми добро уклопљени (као граматика);

3) **семантичка анализа:** проверава смисленост, колико је могуће (нпр. int x="test"+3; нема смисла).

Елементарне конструкције или **токени** су елементи језика које компајлер издваја при превођењу.

1) **Идентификатори:**
   - служи за идентификовање неке конструкције (променљиве, класе, методе...) у Јави;
   - обавезно почиње словом или $ или _ ; у остатку могу и цифре;
   - case sensitive;
   - конвенција: класе велико слово, остало мало.

2) **Кључне речи:**
   - идентификатори који имају специјалну намену
   - abstract, break, char, class, double, else, float, for, goto, if, implements, public ...

3) **Литерали:**
   - речи које представљају неку вредност:

   3.1) **целобројни:** декадни, октални (0562), хексадекадни (0x562), бинарни (0B562) (може велико / може мало)
   3.2) **реални:** позициони запис (23.57) или експоненцијални запис (123e-5);
   float (мора да се нагласи: 12.45f) и double (подразумевано);
   3.3) **логички:** false, true;
   3.4) **знаковни:** сви знакови осим апострофа и обрнуте косе црте ('a', 'b', '2', ...);
   ескејп секвенце: '\'', '\"', '\\', '\r', '\n', '\f', '\t', '\b';
   3.5) **стринговни:** ниска између наводника ("Poz", "Poz\n").

4) **Сепаратори:**
   - служе само за раздвајање једне врсте токена од других;
   - то су: ( ) { } [ ] ; : , .

5) **Коментари:**
   - служе за додатна објашњења;
   - једнолинијски (//), вишелинијски (/* */), документациони (/** */).

6) **Белине:**
   - немају графички приказ, служе за обликовање;
   - размак, хориз. таб, знак за крај реда, знак за нову страну, знак за крај фајла.

7) Оператори:

- омогућавају операције над подацима;
- постоје: префиксни, инфиксни, постфиксни;

7.1) аритметички: + - * / % ++ --

7.2) релациони: == != < > >= <=

- овде мислимо на регистарска поређења!

7.3) битовни: & | ~ ^ « » »»

~ негација

»» шифтовање нулама

7.4) логички: && || !

7.5) условни: <log. izraz> ? <izraz1> : <izraz2>

7.6) инстанцни: <objekat> instanceof <klasa/interfejs>

7.7.) оператор доделе: = += -= *= /= %=

&= |= ^= «= »= »»=

k = 5: вредност овог израза је 5, па може k=l=m=5.

Приоритет	Оператор	Асоцијативност
1	`(), []`	неасоцијативан
2	`new`	неасоцијативан
3	`.`	лево-асоцијативан
4	`++, --`	неасоцијативан
5	`-(унарни), +(унарни), !, ~, ++, --, (tip)`	десно-асоцијативан
6	`*, /, %`	лево-асоцијативан
7	`+, -`	лево-асоцијативан
8	`<<, >>, >>>`	лево-асоцијативан
9	`<, >, <=, >=, instanceof`	неасоцијативан
10	`==, !=`	лево-асоцијативан
11	`&`	лево-асоцијативан
12	`^`	лево-асоцијативан
13	`|`	лево-асоцијативан
14	`&&`	лево-асоцијативан
15	`||`	лево-асоцијативан
16	`?:`	десно-асоцијативан
17	`=, *=, /=, %=, -=, <<=, >>=, >>>=, &=, ^=, |=`	десно-асоцијативан

5. Типови података у Јави

Тип података одређује скуп вредности које могу бити додељене промењивима или изразима.
Свака операција или функција реализује се над аргументима фиксираног типа.
Они доприносе прегледности и ефикасности програма.

Постоје два типа података:

1) **примитивни типови:** (подсетимо се: (из C) кад се покрене нека ф-ја прави се њен стек)

- представљају адресу у стеку која буквално садржи вредност;

нпр. `byte masa = 5;` ⇔ у стеку постоји локација именована masa: 00000101 (5)

- предефинисани;
- један овакав тип је одређен скупом вредности и скупом операција;

1.1) бројевни тип:

1° целобројни тип: byte, short, int, long, char;
8b 16b 32b 64b 8b(unsig.) - фиксно због JVM.

- оператори: аритметички, релациони, битовни;
- сви остали се преводе у int или long.

2° реални тип: float (32b), double(64b);
- IEEE 754

1.2) знаковни тип: char

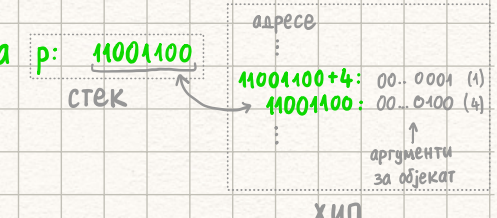
1.3) логички тип: boolean

- наредбе: if, while, for, do-while;
- оператор: ?:

2) **објектни типови:**

- представљају адресу у стеку која садржи адресу на хипу која садржи сам објекат;

нпр. `Point p = new Point(4,1)` ⇔ у стеку постоји адреса именована p: 11001100



2.1) кориснички: преко имена класе или интерфејса;
2.2) низовни: може и преко корисничког објектног типа и преко примитивног [7]
2.3) набројиви: преко кључне речи enum [19]

6. Променљиве, наредбе, изрази...

* **Променљива** је локација у меморији за запис неке вредности или за референцу на објекат.

Делимо их на: 1) **инстанцне:** припадају објекту; не постоје без постојања објекта;
2) **класне:** дефинисане на нивоу класе; постоје и без постојања објекта;
3) **локалне:** припадају методи: оне су на стеку.

Свака променљива има: име (идентификатор), тип (један од набројаних) и вредност (литерал).
+ var
или реф. на објекат.

Свака променљива мора бити декларисана. (тип + име)
Само локалне променљиве морају бити иницијализоване. (остале не морају)
↳ подразумевано: null - референца;
0 - нумеричка;
'\0' - знаковна;
false - логичка.

* **Наредба** је конструкција помоћу које се контролише ток програма и одвијање операција у Јави.
Обавезно се завршавају са ;

1) **наредба декларације:** тип идентификатор1, идентификатор2 = 5, идентификатор3;

2) **празна наредба:** Пример: while (uslov) { ; } (желимо чекање)

3) **обележена наредба:** користи се заједно са наредбама break; и continue;

Пример:
```
obelezje1:
for(...)
    for(...)
        if(...)
            break obelezje1;
        ...
```

4) **наредба доделе:** идентификатор = израз ;

5) **наредба израза**

6) **наредба блока**

7) switch, break, return...

* **Израз** је конструкција која се користи да донесе, израчуна и смести неку вредност.
У изразу могу се наћи операнди (константе, текућа вредност, промењиве...), оператори и сепаратори.

Ослањајући се на раније дефинисане операторе, постоје следећи типови израза:

1) израз доделе;
2) аритметички израз;
3) релациони израз;
4) логички израз;
5) израз са битовским операторима;
6) условни израз;
7) инстанцни израз;
8) **кастовање;**
9) комбинација претходних.

* **Блок** је секвенца од нула, једне или више наредби или декларација лок. промењ., и то између { и }.

Напомена: ова деф. је „рекурзивна", јер је блок деф. преко наредбе, а видели смо да је блок и сам наредба.

Уколико неку промењиву декларишемо у блоку, само ту је можемо и користити.

* **Компилационе јединице** су текстуални садржаји који улазе у процес компилације.

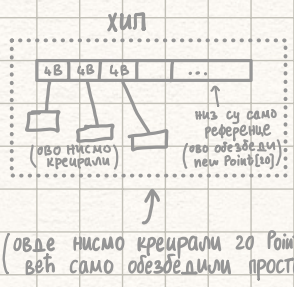
1) **package** директива;
2) **import** директива;
3) дефиниције класа, интерфејса, енумерација.

Пример: уместо сваки пут да пишемо java.util.Scanner,
напишемо import java.util.Scanner; и онда пишемо само Scanner.

7. Низови, наредбе гранања, наредбе циклуса...

* **Низ** је група променљивих истог типа које се појављују под заједничким именом.

Особине:
- садржи линеарно уређен и унапред познат број чланова;
- тип може бити примитивни или објектни;
- у меморији је континуирано (4B + 4B + 4B...).



Декларација:
I начин: `int[] a,b,c;` // svi su nizovi
II начин: `int a,b,c[];` // samo c je niz

Креирање: (не мора литерал може нпр. и)
I начин: `Point[] tacke = new Point[20];`
II начин: `String[] godDoba = {"prolece", "leto", "jesen", "zima"};`

Напомена: ако има наслеђивања, може у исти низ.

Дводимензиони низ: то је низ низова. (више низова „поређамо један испод другог")

Декларација:
I начин: `int[][] a,b,c;` // svi su nizovi
II начин: `int a,b,c[][];` // samo c je niz

Креирање: `int[][] a = new int[50][100];`

Вишедимензиони низ: све аналогно (3,4,5...)

Настављамо оно што је започето у претх. питању:

* **Наредбе гранања** омогућавају да се одабере извршавање једног дела у зависности од испуњења услова.

1) if / else;
2) switch.

* **Наредбе циклуса** омогућавају да се део програма понавља док год је неки услов испуњен.

1) while
2) do-while
3) for
   ↳ специјално колекцијско for: `for(int i : niz)` (22)

* **Обележена наредба:** у претх. питању.

* **break:** излаз из најближе петље.

* **continue:** прелаз на следећу итерацију најближе петље.

8. Дефиниције класе у Јави

Цео Java код налази се унутар **класа**.

Пример дефиниције класе: `public class Krug extends Figura implements Nacrtiv, Pomeriv { ... }`

* (Нова) подела променљивих:

1) **поља**: - синоним за атрибут, променљиве које представљају чланове-податке унутар класе;
   - декларација: (модификатор) + тип + идентификатор;
   - у зависности од модификатора, разликујемо:

   1.1) **инстанцна поља**: - сваки креирани објекат садржи сопствени примерак те променљиве;
   - приступ преко **тачка-нотације**: p1.x је различито од p2.x

   1.2) **класна поља**: - та променљива је дељена међу свим објектима дате класе;
   - пример: `static int pointCount = 0;`
   - постоји чак иако се не креира ни један објекат дате класе;
   - приступ: a.pointCount, а може и Point.pointCount (препоручљиво).

   1.3) **финална поља**: - не може се мењати;
   - пример: `final boolean NETACNO = false;`
   - може и код инстанцних и класних, као и код локалних променљивих;
   - приступ може, али само за узимање вредности.

2) **локалне променљиве**: декларисане у телу метода или блоку;

3) **формални аргументи**: декларисане у заглављу метода. (такође су чланови метода, тј. функција)

* **this**: у оквиру метода примерка или конструктора, this ⇔ објекат над којим је позвана метода;

Пример:
```
public void pomeri(int x, int y){
        this.x = x;
        this.y = y;      // pravimo razliku
    }
```

* **Опсег важења променљиве**: део програма у ком се име променљиве може користити:

1° инстанцне: док год постоји објекат ком припада.
2° класне: увек;
3° локалне: у оквиру блока у ком се налази;

9. Методе

* Већ смо рекли да је **метод** функција која је саставни део објекта;

Пример: <povratni-tip> imeMetoda (arg1, ..., argn) { ... }

Подела метода:

1) **инстанцни метод**: - мора постојати бар један објекат дате класе да би се метод користио;
   - прави се копија за сваку инстанцу.

2) **класни метод**: - не мора постојати ни један објекат дате класе да би се метод користио;
   ⇕ (функција статички метод) - обавезно има static;
   - пример: public static void main (String args[]) { ... }

* **Преоптерећивање метода** је дефинисање више метода са истим именом, а различитим параметрима. За разлику од C, у Јави је ово дозвољено (јер је овде метод одређен са повр. вр. + име + бр. арг.)

Пример: 1) може:
```
void skaliraj (double coef) { ... }
void skaliraj (double coefX, double coefY) { ... }
```

2) не може:
```
void skaliraj (double coef) { ... }
void skaliraj (double k) { ... }
```

* **Конструктор** је метод класе који се позива када хоћемо да креирамо нови примерак дате класе. (статички) (то радимо преко оператора new)

- име му је исто као име класе и никад не враћа вредност;

- ако корисник сам не направи конструктор, користи се подразумевани конструктор (нема аргументе и све на подразум. вр.)

- и овде је дозвољено преоптерећивање:

Пример:
```
public Point (int x, int y) { this.x = x; this.y = y; }
public Point ( ) { x = 0; y = 0; }
public Point (int x, int y, Color color) { this(x,y); this.color = color; }
```

* **Копирајући конструктор**: сређује „референцијалну зависност". Препоручљиво да увек постоји.

Пример: проблем:

```
class Line extends GeometryObject {
        Point a;
        Point b;

        Line( Point a, Point b ) {
                this.a = a;
                this.b = b;
        }

        public static void main( String[] argsKL ) {
                Point tackaA = new Point( 19, 20 );
                Point tackaB = new Point( 7, 52 );
                Line linijaAB = new Line( tackaA, tackaB );
                tackaA.setX(200);
        // ovde je doslo do promene vrednosti linijaAB
        // sa [(19,20)-(7, 52)] na [(200,20)-(7, 52)]
        // iako nije radjeno sa linijaAB, vec sa tackaA
        }
}
```

; решење:

```
class Point extends GeometryObject {
        int x; int y;

        Point( int x, int y ) {
                this.x = x;
                this.y = y;
        }

        Point( final Point tacka ) {
                this.x = tacka.x;
                this.y = tacka.y;
        }
}
```

```
class Line extends GeometryObject {
        Point a;
        Point b;

        Line( Point a, Point b ) {
                this.a = new Point( a );
                this.b = new Point( b );
        }

        public static void main( String[] argsKL ) {
                Point tackaA = new Point( 19, 20 );
                Point tackaB = new Point( 7, 52 );
                Line linijaAB = new Line( tackaA, tackaB );
                tackaA.setX(200);
                // promena kod tackaA ne menja vrednost linijaAB
        }
}
```

10. Превазилажење метода

* **Превазилажење метода** омогућава да метод потклассе има другачије понашање него у класи. Нови метод има исто име, број и типове аргумената, као и исти повратни тип (тј. може и подтип) То постижемо преко **@Override**

Пример:

```
public class Line{
    void draw() {...}
}
```

```
public class DottedLine extends Line{
    @Override
    void draw() { ... }
}
```

Могуће је и позивање метода из натклассе који је превазиђен у потклаcи: преко **super.**

Пример:

```
public class DottedLine extends Line{
    @Override
    void draw() { ... }

    super.draw();    // poziva draw iz Line
}
```

Могуће је и позвати конструктор из натклассе: опет преко **super.** (исто као што смо могли да позовемо други конструктор преко this)

Пример:

```
public class DottedLine extends Line{
    ...
    DottedLine(int x, int y, double radius){
        super(x,y);   //poziva konstruktor iz Line
        this.radius = radius;
```

* У Јави постоје и **рекурзивни методи.** (JVM омогућава то преко стека)

* **Метод финализатор** је метод дефинисан у класи Object. (зато га у свакој класи можемо превазићи) Активира се када и аутоматски сакупљач отпадака, па то користимо на начин као у примеру. Не користи се више.

Пример:

```
protected void finalize() {
    System.out.println("Obrisan" + this);
}
```

* **Метод main:** `public static void main(String args[]) { ... }`

args[] садржи аргументе командне линије: java Prvi 3 5.34 "Poz poz" Poz poz

11. Објекти

Као што смо рекли, **објекат** је интегрална целина података и процедура за рад са њима.
Синоними су му инстанца и примерак.

- **new** је оператор који се користи за креирање објеката.

- Јасно, компоненте објекта су инстанцне променљиве и инстанцни методи.
Њима се приступа преко **тачка-нотације**.

- Када се креира објекат, променљива којој се додељује је референца на тај објекат.
Дакле, ако је при позиву метода неки аргумент објекат, прослеђује се вредност референце, а не сам објекат.
(за арг. примитивног типа важи обрнуто - шаље се вредност директно)

- Објекте **поредимо** на два начина: 1) регистарски садржај: == !=

2) семантички садржај: **equals** - метода у Object;
- свака класа га може превазићи.

Пример:
```
@Override
public boolean equals(Object obj){
    Point p = (Point) obj;
    return this.x = p.x && this.y = p.y;
}
```

- Могуће је одредити класу датог објекта: преко **getClass** - метод у Object, који се НЕ МОЖЕ превазићи
- враћа класу, а **getName** враћа String.

Пример: `String s = objekat.getClass().getName();`

Може и да се провери: преко оператора **instanceof**; Пример: `"bilo sta" instanceof String // true`

* **Конверзија типова:**

1) примитивни ↔ примитивни. пример: `(int) x/y;`

2) примитивни ↔ објекат: само код **Wrappera**: класа-омотача
пример:
```
Integer ceoObj = new Integer(45);
int ceo = ceoObj.intValue();
Integer ceoObj2 = new Integer(2*ceo);
```

3) објекат ↔ објекат: само повезани (extends, implements):
пример:
```
Vozilo x = new Kamion();
Kamion y;
y = (Kamion) x;
```
ако би било нпр. Vozilo ↦ Kamion, може само x=y.

* **Иницијализациони блокови:**

1) инстанцни и.б. `int b; { b=7; }`
2) статички и.б. `static int a; static { a=5; }`

(користимо ако пре иницијализ. мора нпр. петља, па не може у један ред)

12. Стрингови

Стринг је објектни тип који се користи за текст.

Примерци класе String су немутирајући:
```
String test = "nesto";
String kopija = test;
test += " drugo";   // pravi se novi string, kopija ≠ test
```

* length(): враћа дужину низа;

* Надовезивање се врши оператором + или +=;

* toString(): у класи Object, може се превазићи; (препоручује се да увек постоји)

* Поређење се врши методама equals() или equalsIgnoreCase(); (враћа false/true)
* Лексикографско поређење се врши методом compareTo(String) (враћа разлику, као strcmp)

* valueOf(): креира String од вредности примитивног типа; (статички метод)

* startsWith(String): проверава да ли наш стринг почиње String-ом аргументом;
  endsWith(String): завршава

* indexOf(int ch): враћа индекс првог појављивања ch у нашем стрингу, -1 ако га нема;
  indexOf(String s): почетка првог појављивања s

* lastIndexOf(int ch): исто, само са краја;
  lastIndexOf(String s):

* substring(int start): враћа String са почетком у start, па до краја нашег стринга;
  substring(int start, int end): враћа String са почетком у start, а крајем у end-1;

* replace(char, char): враћа String добијен када сваку појаву првог, заменимо другим char-ом;

* trim(): враћа String добијен када обришемо белине са почетка и краја нашег стринга;

* char[] toCharArray(): креира низ карактера од нашег стринга;

* String.copyValueOf(char[]): креира стринг од низа карактера.

13. Модификатори

Модификатори су специјалне кључне речи које мењају понашање класа, метода, променљивих или наредби увоза.
Има их много:
- модификатори синхронизације (за паралелизацију),
- нативни модификатори (подешавања за OS)
- модификатори-анотације (за билине објашњава код)
- ...
- **модификатори контроле приступа** (њима се угл. бавимо)

1) **public** - из било које класе (чак и из других пакета);
2) **package** - из било које класе у том пакету; (подразумевано)
3) **protected** - из класе и њених поткласа (чак и из других пакета);
4) **private** - само из те класе

Пример: ако су променљиве приватне, за рад са њима се користе **getter / setter**-и

```
public int getR(){
    return r;
}
```

```
public void setR(int r){
    this.r=r;
}
```

- **модификатор static:** променљиве - променљива везана за постојање класе;
  методе - метода везана за постојање класе;
  наредбе увоза - омогућава остало без употребе пуне квалификације;
  иницијализациони блокови - иницијализација статичких поља.

  сврха: није везано за објекат, него саму класу.

- **модификатор final:** класе - не могу бити наслеђиване;
  поља - не могу бити мењана након иницијализације;
  методе - не могу бити превазиђене;
  параметри метода - не могу бити мењани унутар метода.

- **модификатор abstract:** за дефинисање апстрактних класа и метода.

14. Апстрактне класе и интерфејси

* Као што смо рекли, **апстрактна класа** је класа у којој постоји бар један метод који нема дефиницију, већ само декларацију.

Користимо их када знамо да нешто треба да постоји, али нисмо сигурни како тачно.

Пример:

```
public abstract class Shape{
    //...
    abstract void calculateArea();
}
```

```
public class Rectangle extends Shape{
    //...
@Override void calculateArea(){
        this.area = this.a * this.b;
    }
}
```

```
public class Circle extends Shape{
    //...
@Override void calculateArea(){
        this.area = this.r * this.r * Math.PI;
    }
}
```

* Као што смо рекли, **интерфејс** обезбеђује апстрактно понашање које се додаје било којој класи, а које није обезбеђено преко њених наткласа.

Преко њих у некој мери компензујемо недостатак вишеструког наслеђивања.

Пример:

```
public interface Audible{
    void sound();
}
```

```
public class Cat implements Audible{
    public void sound(){
        System.out.println("Meow");
    }
}
```

```
public class Automobil implements Audible{
    public void sound(){
        System.out.println("Vroom");
    }
}
```

Један интерфејс може наследити више других: `interface Drugi extends Prvi, Primarni { ... }`

Једна класа може имплементирати више интерфејса: `class Klasa implements InA, InB { ... }`

Пример: интерфејс који се често имплементира је **Comparable**, који има једну методу:

```
int compareTo(Object other);
```

овим можемо сортирати: методом `static void sort(Object[] a)` из класе Arrays.
↳ елементи морају имплем. Comparable.

15. Препоруке за наслеђивање

1) Заједничке операције и поља сместити у наткласе;

   - све заједничко од Employee и Student треба ставити у њихову наткласу Person.

2) Избегавати употребу protected поља;

   - није толико безбедна (направимо поткласу и имамо приступ protected пољу, а може и из било које друге класе у том пакету)

3) Користити наслеђивање за моделирање релације <јесте конкретизација>;

   - не треба користити наслеђивање на силу, већ само када то заиста природно.

4) Не користити наслеђивање осим ако оно има смисла за све методе наткласе;

   - слично.

5) Приликом превазилажења метода, не мењати очекивано понашање (тј. поштовати принцип замене);

   - у смислу да не треба мењати сврху методе, већ само по потреби мало прилагодити.

6) Користити полиморфизам, а не информације о типу;

16.

Пакети

* **Пакети** служе за груписање сличних типова (класа, интерфејса, енум. типова и типова нотације)

Помоћу њих: * лакше одређујемо да ли су типови повезани;
* лакше тражимо потребне типове;
* избегавамо именске конфликте (paketA.Klasa ≠ paketB.Klasa);
* допуштамо да типови унутар пакета имају неограничен међусобни приступ.

Како креирамо пакете: 1) избор имена: пракса је обрнути Интернет домен (rs.ac.bg.matf.mv19013);
2) креирање структуре директоријума: први фолдер rs, други ac ...
у сваки можемо убацивати типове;
3) додавање package наредбе: одмах на почетку: `package rs.ac.bg.matf;`

За коришћење уграђених класа, морамо знати где се налазе у оквиру система.
То радимо преко команде оперативног система **CLASSPATH**.

* **Угњеждена класа** је класа дефинисана унутар неке друге класе.
Ово користимо исто због логичког груписања, с тим што овде користимо једну особину:
угњеждене класе могу да виде приватна поља из спољашње класе.

Пример:
```
class OuterClass{
    static class StaticNestedClass{ ... }
    class InnerClass { ... }
}
```

Пример:
```
class Math{
    class Complex { ... }
    class Trigonometry { ... }
}
```

Може бити: 1) **статичка угњеждена класа**: - има static уз декларацију;
- нису „чланице" спољ. класе;
- могу позивати само статичке ствари из спољ. класе;
- декл. у main: `var snc = new StaticNestedClass();`

2) **унутрашња класа**: - јесу „чланице" спољ. класе;
- могу позивати све из спољ. класе;
- не можемо правити ништа статичко у њима (јер су везане за постојање објекта)
- декл. у main:
```
var oc = new OuterClass();
var ic = oc.new InnerClass();
```

* **Локална унутрашња класа** је класа дефинисана унутар неког метода.
На слајдовима је дат добар пример са телефонима.

* **Анонимна класа** је лок. ун. класа која се користи само једном. Оне немају име.
Допуштају да се истовремено декларише класа и креира њен примерак.
На слајдовима је дат добар пример са компаратором студената.

17.

Изузеци

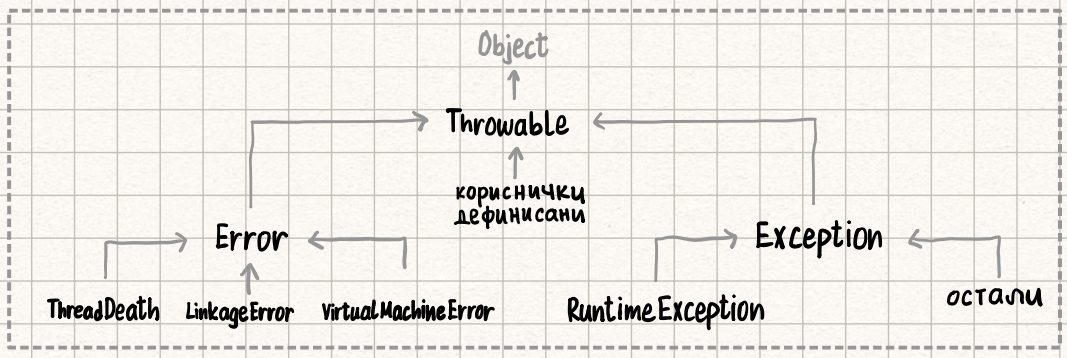
Изузеци се у Јави користе као начин сигнализирања озбиљних проблема при извршавању програма. Прецизније, то су објекти са информацијама о проблему.

Каже се да је изузетак **избачен**, а код који прима објекат изузетак као параметар га **хвата**.

Пример:
```
public class Primer{
    public static void main(String args[]){
        int a[] = new int[2];
        System.out.println("Pristupam elementu: " + a[3]);
    }
}
```
ово можемо схватити као методу која у себи садржи и изузетак

Ово не може да се преведе, већ добијамо **ArrayIndexOutOfBoundsException**.

Ситуације:
1) грешке кода или података: лоше кастовање, оно из примера, дељење нулом...;
2) изузеци стандардних метода: нпр. избацивање StringIndexOutOfBoundsException;
3) избацивање кориснички деф. изузетака;
4) Јава грешке: обично последица грешке у нашем програму.



Throwable је наткласа свих изузетака: и кориснички деф. и стандардних.

Сви стандардни су распоређени у две класе:

Error: они који не треба да се хватају. Углавном се одмах прекине програм.

1) **ThreadDeath:** када се нит која се извршава намерно стопира;
2) **LinkageError:** озбиљни проблеми са класама (нпр. покушај креирања непостојећег класног типа);
3) **VirtualMachineError:** када се деси катастрофални пад JVM. (има 4 потклассе)

Exception: они који треба да се хватају. Скоро увек мора да се укључи код који рукује њима;

1) **RuntimeException:** -једино овде треба избегавати избацивање изузетка; (нпр. if-else)
   -најчешће логичка грешка.

2) **остали:** обавезно морају да се разреше. (IO, FileNotFound, Parse ...)

18. Руковање изузецима

Када сматрамо да постоји „опасност" од неког конкретног изузетка, уз име методе пишемо следеће:

Пример: `double metod() throws IOException, FileNotFoundException { ... }`

Уз то, ако неки други метод користи овај наш, онда и тај нови мора или да их обради или да има throws.

* Да бисмо руковали изузецима, укључујемо три врсте блокова:

1) try: обухвата код где се може јавити један или више изузетака;
2) catch: обухвата код који рукује изузецима одређеног типа (дешава се ако је у try избачен изузетак);
3) finally: обухвата код који се увек извршава, без обзира на то да ли је избачен неки изузетак.

Пример:

```
public class Primer{
    public static void main(String args[]){
        try{
            int a[] = new int[2];
            System.out.println("Pristupam elementu: " + a[3]);
        } catch (ArrayIndexOutOfBoundsException e){
            System.out.println("Izuzetak izbacen: " + e);
        } catch (IOException e){
            System.out.println("Izuzetak izbacen: " + e);
        } finally{
            System.out.println("Pozdrav");
        }
    }
}
```

Напомене:

1) Ако класа која је аргумент за catch има поткласе, онда тај catch блок рукује и тим поткласама;
2) Ако постоји хијерархија, редослед catch блокова треба да иде од изведенијих ка основнијим;
3) try блок не може да стоји сам, већ уз њега следи бар један од catch и finally блокова.

Напомена: На quora.com дат је одличан пример са жирком и шампањцем.

При крају предавања, дат је одличан пример како правимо сами своје изузетке.

19. Енумерисани типови

Када за неку променљиву имамо коначан скуп допустивих вредности, користимо **енумерисане типове**.

Пример: `enum Velicina {SMALL, MEDIUM, LARGE, EXTRA_LARGE};`

Понашају се као класе.
Класа из примера садржи тачно 4 објекта типа Velicina и не можемо направити нове.
Подразумевано, степен видљивости је public.

Могуће је декларисати променљиву енум. типа.

Пример:
```
Velicina v = Velicina.MEDIUM;     // нема new, v је obj. ref. na obj. MEDIUM
Velicina v1 = NULL;
```

Напомена: пореде се са `==`, а не equals.

Могуће је додати поља, методе и конструкторе типу енумерације
↳ само за конструисање наших константи, не за нове

Пример:
```
enum Planeta {
        MERKUR, VENERA, ZEMLJA, MARS, JUPITER, SATURN, URAN, NEPTUN;

        private double mass;  private double r;
        public double getMass() { return mass;}
        public Planeta(double mass, double r){ this.mass = mass; this.r = r}
}
```

Пошто су енум. типови потклаце класе **Enum**, они наслеђују неке методе те класе:

* **toString()**: враћа име објекта (нпр. Velicina.MEDIUM.toString(); враћа „MEDIUM") (може се превазићи)
* **valueOf()**: враћа објекат са тим именом (нпр. Velicina v = (Velicina) Enum.valueOf(Velicina.class, "SMALL"); поставља v на Velicina.SMALL)
* **values()**: враћа низ свих објеката тог енум. типа
* **ordinal()**: враћа редни број (почевши од 0) (дакле, редослед је битан)

Може се користити и са switch наредбом.

20. Генерички типови

Генерички типови омогућавају да типови буду параметри (при деф. класа, интерфејса и метода).
Предности: - строга контрола типа (при превођењу);
- нема кастовања
- омогућава имплементацију тзв. генеричких алгоритама (као qsort у C)

Такође, преко генеричких типова можемо имплементирати стек и разне друге ствари.

Дефиниција генеричког типа:

```
public class Box<T> {                          // T - predstavlja tip
    private T value;
    public T getValue(){ return value;}

    public static void main(String[] args){
        Box<Integer> box1 = new Box<Integer>();   // generički poziv tipa: konstruktoru šaljemo i tip
        Box<Integer> box2 = new Box<>();          // može i ovo, kao podrazumevano
        var v1 = box1.getValue;
}
```

Генерички метод: опсег параметра ограничен само на тај метод

```
public class Util {
    public static <K,V> boolean compare(Pair<K,V> p1, Pair<K,V> p2){
        return p1.getKey().equals(p2.getKey()) && p1.getValue().equals(p2.getValue());
}   // ovaj metod poredi dva para iz generičke klase Pair.
```

Позив:
```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

Дакле, <K,V> иде пре повратног типа.

21. Ограничења за генеричке типове

* Рецимо да хоћемо да одредимо најмањи елемент низа генеричког типа T.
Како можемо бити сигурни да класа T садржи метод compareTo?
Морамо да уведемо тзв. ограничење за генерички тип T.

Решење: `public static <T extends Comparable> T min(T[] niz)`

- Сада метод min можемо да позовемо само над низовима примерака класа које имплементирају интерфејс Comparable (у овом случ. Comparable је и сам ген. интерфејс)

- Зашто extends, а не implements?
Зато што горњи запис изражава да је T подтип Comparable. (нису хтели нову кљ. реч да уводе)
T и тип који ограничава (овде је то Comparable) могу бити и класе, али и интерфејси.

Више ограничења: `<T extends Comparable & Serializable>`

- Могуће више интерфејса, али само једна класа за ограничење.
Такође класа иде пре свих интерфејса

* Како генерички типови функционишу у JVM, тј. при превођењу?

Кад год се дефинише генерички тип, аутоматски се обезбеђује тзв. сирови тип.
Променљиве се замене типом који их ограничава или са Object (ако нема ограничења)

Пример: ген. тип: през. 9, слајд 30;
сирови: през. 9, слајд 31.

Дакле, у JVM не постоје генерици

* Каква је веза генерика и наслеђивања?

У општем случају, Pair<S> и Pair<T> нису ни у каквој вези (без обзира на однос T и S)

22. Колекције

* **Колекција** представља набројиву групу ентитета.
Тачније, оне су **имплементације** – класе које имплм. интерфејс.

Интерфејс: Показујемо на поједностављеном примеру реда (FIFO) из стд. библ. (27)

```
interface MyQueue<E> {          // ne govori kako je realizovano/implementirano
    void add(E element);
    E remove();
    int size();
}
```

Имплементација: свака класа која имплементира интерфејс MyQueue: (наводимо две)

```
* class MyLinkedListQueue<E> implements MyQueue<E> {
    private Node<E> head;
    private Node<E> tail;

    @Override public void add(E element){...}
    @Override public E remove(){...}
    @Override public int size(){...}

    private class Node<E>{                          //čvor
        private E element;
        private Node<E> next;                       //„pokazivač" na sledeći
        public Node(E element){ this.element = element;}   // konstruktor
    }
}

* class MyCircularArrayQueue<E> implements MyQueue<E>{
    //analogno, samo u konstr. kao arg. imamo capacity i imamo niz
}
```

Main: Када програм користи колекцију, он не мора да зна како је имплементирана. (тако „обједињујемо")
Зато се имплементација користи само за креирање, а за реф. на обј. користимо интерфејс.

```
MyQueue<Integer> red = new MyLinkedListQueue<>();   //nema arg. jer prazna lista = NULL
red.add(34);
```

Напомена: Имплементација кружним низом је бржа, али је меморијски ограничена.

Напомена: У API постоје класе AbstractList, AbstractSet, AbstractQueue...
Када правимо сопствену класу за ред, лакше је да наследимо AbstractQueue
него да имплементирамо све методе интерфејса Queue.

↓ 23 на крају

* Основни интерфејс (предак свима) за колекцијске класе у Јави је интерфејс Collection:

```
public interface Collection<E> extends Iterable<E> {
    boolean add(E element);
    // ima još...
}

public interface Iterable<E> {
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
    void remove();
}
```

iter u primeru
//
// vraća objekat koji implementira interfejs Iterator
// a taj obj. ima mogućnost prolaska kroz kolekciju

(slika)
// briše onaj na koji je poslednji put "pređen"

Пример:
```
Collection<String> c = ... ;
Iterator<String> iter = c.iterator();
while (iter.hasNext()) { System.out.println(iter.next()); }
```

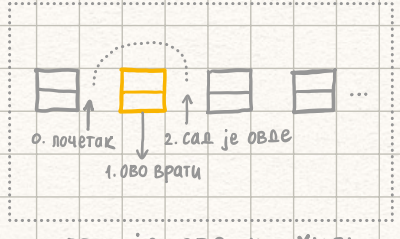
Напомена: Ако позовемо iter.next() без провере hasNext, избацује се NoSuchElementException

Напомена: Скраћени запис (уместо 3. реда): преко колекцијског for. (7)

```
for (String element : c) { System.out.println(iter.next()); }
```

Ово се заправо и дешава при превођењу.

Напомена: Итератор приликом позива iter.next истовремено напредује за једну позицију:



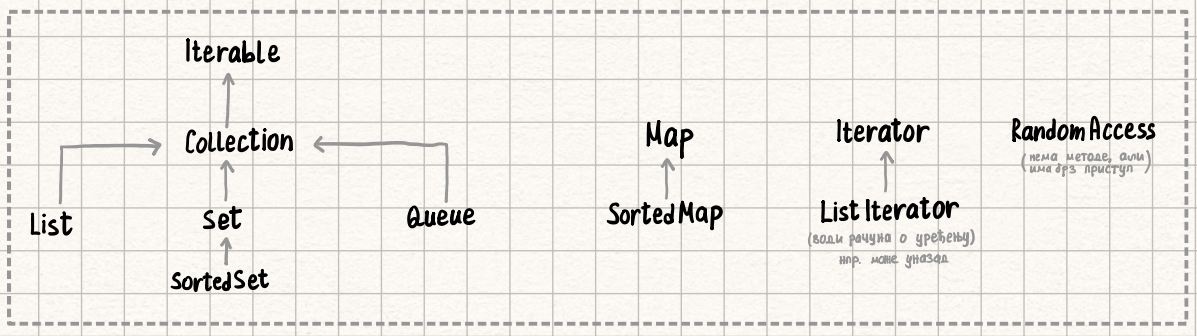
ово је све на хипу

Напомена: Постоје и „специјализовани" итератори: нпр. ListIterator.

(он наслеђује интерфејс Iterator)

23. Преглед колекцијских интерфејса

Колекцијски интерфејси из стд. библиотеке:
Пишемо и методе (оне нису имплм, значи описи говоре шта треба да раде класе које их имплементирају).



* Најчешће коришћене класе (које имплементирају наведене интерфејсе) у JDK су:

24 LinkedList, ArrayList, - наслеђују AbstractList;
25 HashSet, TreeSet, - наслеђују AbstractSet;
27 PriorityQueue, - наслеђују AbstractQueue;
26 HashMap, TreeMap. - наслеђују AbstractMap.

(ово постоји јер неке методе можемо одмах реализ. без обзира коју од ових имплементација изаберемо)

* Како сортирамо елементе у колекцији?

→ Претпостављамо да имплементирају интерфејс Comparable, па онда помоћу compareTo() (као низови);

→ Иначе, има и други начин:

```
public interface Comparator<T> {
    int compare(T a, T b);
```

(њега прослеђимо методу који сортира)

24.

Листе

Листе јесу специјалан случај колекција, с тим што је овде постоји уређење.
Због тога, оне користе посебан итератор: interface ListIterator<E> extends Iterator<E>
↳ може уназад

Следеће две класе имплементирају интерфејс List:

1) class LinkedList<E> - поткласа од AbstractCollection

- двострука листа!
- метод listIterator() ове класе (а „пореклом" из интерфејса) враћа итератор:

```
ListIterator<String> iter = osoblje.listIterator();
```

- не подржава брз приступ (не импл. RandomAccess, него има линеаран приступ);
  - без обзира на то, постоји метод за приступ n-том елеи: String s = lista.get();

2) class ArrayList<E>:

- ништа друго него низ енкапсулиран у класу; при томе се динамички прилагођава захтевима;
  (то значи не треба да бринемо о мањку меморије)
- све методе (get, set, isEmpty, iterator, listIterator) се извршавају за константно време;
  (како? Користи се својство са слике из 7, тј. низ = реф. по 4B, па знамо колико треба места да одемо да стигнемо до сваког елем. у низу)

Из овога, јасно је кад је боље користити коју имплементацију. (шифтовање vs. претрага)

25.

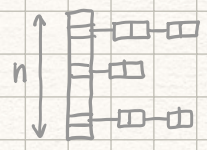
Скупови

Скупови јесу специјалан случај колекција, с тим што овде нема дупликата.

Следеће две класе имплементирају интерфејс **Set**:

1) class **HashSet<E>**: користи се кад редослед није битан.

Морамо прво објаснити концепт **Hash табеле**:



- то је низ (димензије n) повезаних листи

- Сваком објекту се додељује вредност - то се зове **Hash код**;
  Израчунамо (Hash код) mod n - добијемо индекс који одговара листи у коју смештамо обј;

- Некад долази до колизије (нпр. два разл. обј. исти mod ⇒ исти код);
  Тада се тај обј. пореди са свим обј. у одг. листе и ако је већ присутан, не додајемо га;
  Ако је табела препуна, мора да се направи нова, већа (да се рехешира).

У класи Object, постоји метод int **hashCode()** који враћа Hash код за објекат this.
Обавезно морамо да ускладимо ту, као и методу **equals**. (да не буде дупликата)

2) class **TreeSet<E>**: користи се кад је редослед битан, тј. желимо неки вид уређења.

Уместо Hash табеле, користи се дрво (приступ логаритамски);

- Подразумева се да обј. из скупа имплементирају Comparable; (или имамо Comparator)

- Може имплементирати интерфејсе SortedSet, као и NavigableSet;

Из овога, јасно је кад је боље користити коју имплементацију. (претрага vs. уређење)

26. Каталози

Интерфејс:

Каталози су структуре података које чувају парове **кључ/вредност**.
Код њих се лако може наћи вредност ако се наведе кључ.

Пример: кључеви: ИД бројеви; вредности: објекти типа Employee.

Напомена: кључ може бити null, али вредност не може.

Класе:

Следеће две класе имплементирају интерфејс **Map**:

1) class **HashMap<K,V>**: не сортира каталоге (као и код скупова у 25);
2) class **TreeMap<E>**: сортира каталоге (као и код скупова у 25).

Каталог није самостална колекција, већ се састоји из три подструктуре:

- Set<K> **keySet**(): скуп кључева; (нема понављања)
- Collection<K> **values**(): колекција вредности; (није скуп, јер могу понављања)
- Set<Map.Entry<K,V>> **entrySet**(): скуп парова кључ/вредност.

Пример:
```
Map<String, Employee> staff = new HashMap<String, Employee>();
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
Employee e = staff.get("987-98-9996");
```

27.

Редови

Интерфејси:

* Редови (интерфејс Queue) су спец. случај колекција који омогућују ефикасно додавање на крај и уклањање са почетка.

  Имплементира га класа LinkedList. (погледати 22)

* Редови са два краја (интерфејс Deque) омогућују ефикасно додавање и уклањање и са почетка и са краја.

  Имплементирају га класе LinkedList и ArrayDeque.

Класа:

* class PriorityQueue, иако при уносу прима елементе у произв. поретку, враћа их сортиране. У букв. смислу, то је самоорганизовано дрво (гомила).

  - Кад се позове remove, добија се најмањи елемент (по дефинисаном критеријуму).
  - Користи се за распоређивање послова (сви имају свој приоритет)

* (JDK) Колекције и генерици:

  - Генерици згодни нпр. генеричку методу максимум колекције можемо импл. преко итератора.
  - У класи Collections постоје методе: shuffle, као и binarySearch и sort (Merge).

28. Токови, читачи и писачи, уланчавање токова

* Постоје две реализације улаза/излаза у Јави: IO (орић. ка токовима) и NIO (орић. ка баферу).
  (овде / везбе)

Улаз и излаз у Јави су реализовани преко **токова података**, прилагођених читању бинар. фајлова.

Поступак је да се креира ток, који ће приликом позива конструктора бити придружен датотеци/конзоли, а улазно/излазне операције се реализују позивима метода над тим током
↳ могу да бацају изузетке (IOException)

Основу чине две апстрактне класе:

1) class **InputStream**: има потклассе FileInputStream, DataInputStream... (преко њих се заправо реализује, јер је InputStream апстрактна)

   - метод public abstract int **read**() чита 1 бајт; (број 0-255) (враћа int да би могао да врати -1 за неуспех)
   - има и методе read(byte b[]), read(byte b[], int start, int length) throws IOException (преоптерећен)

2) class **OutputStream**: има потклассе FileOutputStream, DataOutputStream...

   - методе: public abstract void **write**(int b) - исписује b;
     public void **flush**() - празни излазни бафер; (интерфејс Flushable)
     public void **close**() - затвара излазни ток података. (Closable)

Инстанце ових класа су стандардни улаз и излаз: System.in и System.out

* Осим токова, за улаз и излаз се користе **читачи** и **писачи**: иста ствар, само за читање карактера.

Поступак аналоган.

Основу чине две апстрактне класе:

1) class **Reader** (једна од њених потклаcа је BufferedReader)

   - метод public abstract int **read**() чита 1 цео број; (Unicode карактер)
   - има и методе skip, ready, mark, reset, close;

2) class **Writer**

   - методе: public abstract void **write**(byte b) преоптерећен; (има још write метода)
     public void **append**(char c) додаје знак у писач (Appendable)
     public void **flush**() празни излазни бафер; (интерфејс Flushable)
     public void **close**() затвара излазни ток података. (Closable)

* Токови се могу **уланчавати**: тиме добијамо додатну функционалност:

Пример: DataInputStream din = new DataInputStream(new FileInputStream("employee.dat"));

29.

Рад са датотекама

У Јави за рад са фајловима и директоријумима користимо класу File.

- објекти те класе нису букв. ти фајлови, већ они енкапсулирају путању. (не гарантује да постоји)

- постоје 4 конструктора:

```
1) File myDir  = new File("C:/ ... ");                  - String је путања на коју чувамо;
2) File myFile = new File(myDir, "File.java");          - дајемо који дир. и име фајла;
3) File myFile = new File("C:/ ... ", "File2.java");    - дајемо путању до дир. и име фајла;
4) File remoteFile = new File(new URI("http://..."));   - URI је тип који енкапс. униф. ид. ресурса на вебу.
```

(путање горе су апсолутне, а могли смо и релативне)

- методе ове класе:

- getName()
- getPath()
- getAbsolutePath()
- isAbsolute()
- getParent()
- getParentFile()
- toString()
- equals()

- exists()
- isDirectory()
- isFile()
- isHidden()
- canRead()
- canWrite()

- list()
- listFiles()
- length()
- lastModified()
- listRoots()

- renameTo(File path)
- setReadOnly()
- mkdir()
- mkdirs()
- createNewFile()
- createTempFile()
- delete()
- deleteOnExit()

- методе за филтрирање листе (за list() и listFiles)

30. Класа Scanner, серијализација

* Класа **Scanner** користи се за парсирање примитивних типова и String-ова
Објекат ове класе може читати и из сваког обј. који имплементира интерфејс **Readable**.
Улаз раставља на токене.

Пример: `Scanner sc = new Scanner(System.in)`
↳ public static поље из класе InputStream (28)
ту може и неки наш фајл

Методе ове класе:

- `next()`
- `nextXXX()`
- `hasNext()`
- `hasNextXXX()`

(XXX = Boolean, Double, Int...)

- `public Scanner(InputStream source)`
- `public Scanner(File source)`
- `public Scanner(String source)`

* **Серијализација** је процес писања објекта у ток података. (не текст, него Hex)
- неопходно је да објекат имплм. интерфејс **Serializable**. (он нема методе, служи као индикатор)
- метод **writeObject** за обј. типа ObjectOutputStream; (баца IOException)

```
Пример: ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(
                                    new File("C:/temp.imenik.bin")));
        out.writeObject(imenik); out.close();
```

Десеријализација је процес читања објекта из фајла.
- неопходно је да објекат имплм. интерфејс Serializable.
- метод **readObject** за обј. типа ObjectInputStream; (баца IOException, ClassNotFoundException)

```
Пример: ObjectInputStream in = new ObjectInputStream(new FileInputStream(
                                  new File("C:/temp.imenik.bin")));
        imenik = (HashMap<Osoba, Unos>) in.readObject(); in.close();
```

Напомена: Могуће је да се серијализација објекта класе одвија аутоматски.
Треба да буде испуњено одређених 6 услова.

Проблем: Ако се класа измени у току читања (проблем са serialVersionUID)