FIȘA LABORATOR 6

Obiective

Comparabilitate semantică și operații asupra colecțiilor de obiecte.

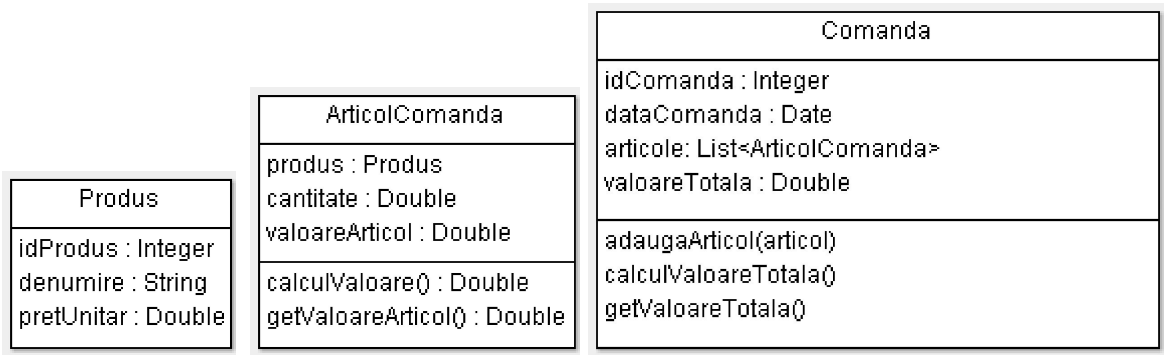
Concepte [Sub-teme țintă]

Egalitate și comparabilitate semantică
Operații cu colecții: ordonarea obiectelor în colecții (comparabilitate în contextul colecțiilor, alternative: Comparable și Comparator)
Operații cu colecții: intersecții, diferențe, reuniuni
Operații cu colecții: interogări rudimentare (căutare pe bază de egalitate și comparabilitate în contextul colecțiilor)
Operații cu colecții: indexări sau lucru cu mape-asociații
equals(), compareTo(), Random, Math.random(), try/catch/Exception

Desfășurare-Repere

Exemplu de predare

(i) Creați (dacă nu există deja) clasele **Produs** cu structura *idProdus*, *denumire*, *pretUnitar*, **ArticolComanda** cu structura *produs* și *cantitate*, **Comanda** cu structura ***idComanda***, *dataComanda* și *articole*. (vezi L2).



- (nu uitați să adăugați câmpul *idComanda* în clasa **Comanda**);
- generați getteri, setteri și constructorii necesari (cu și fără parametri);
- generați metode *equals* (*Source – Generate hashcode and equals*) pentru ambele clase astfel încât principiul de "egalitate" între **Produs**-e să se bazeze pe valorile câmpului *idProdus*, iar pentru **Comenzi**, pe valorile câmpului *idComanda*;

- generați metode *toString* (*Source – Override/Implement method – Object - toString*) pentru ambele clase **Produs** și **Comanda**, metodele sunt necesare pentru reprezentarea (conversia) instanțelor acestor în șiruri de caractere (String). Completați aceste metode astfel:

```
public class Produs {

	/*... ... ... ... ... ... ... ... ... ... ... ... ... ... ...*/
	@Override
	public String toString() {
		return "Produs id: " + this.idProdus + ", denumire: " + this.denumire +
				", pret unitar: " + this.pretUnitar;
	}
	/*... ... ... ... ... ... ... ... ... ... ... ... ... ... ...*/

}

public class Comanda {

	/*... ... ... ... ... ... ... ... ... ... ... ... ... ... ...*/
	@Override
	public String toString() {
		return "Comanda " + this.idComanda + ", valoare totala " + this.getValoareTotala();
	}
	/*... ... ... ... ... ... ... ... ... ... ... ... ... ... ...*/

}

public class ArticolComanda {

	/*... ... ... ... ... ... ... ... ... ... ... ... ... ... ...*/
	@Override
	public String toString() {
		return this.produs + ", cantitate:" + this.cantitate +
			", valoare articol " + this.getValoareArticol(); }
	/*... ... ... ... ... ... ... ... ... ... ... ... ... ... ...*/

}
```

(ii) Pentru generarea și gestionarea (interogarea, sortarea etc.) colecțiilor de produse și comenzi creați clasa *Registru* cu următoarea structură:

- inițializați corespunzător cele două *List*-e de *produse* și *comenzi* folosind clasa *ArrayList*;
- adăugați metodele *generateRandomProduse* și *generateRandomComenzi* pentru a obține colecțiile de test cu care se va lucra în continuare.

```
private void generateRandomProduse(Integer nrProduse){
	Random randomPret = new Random();
	Integer pret;
	for (int i=1; i <= nrProduse; i++){
		pret = 50 + randomPret.nextInt(1450);
```

```
                produse.add(new Produs(i, "Produs_" + i, pret.doubleValue()));
        }
}

private void generateRandomComenzi(Integer nrComenzi){
        Random randomNrArticole = new Random();
        Random randomCantitate = new Random();
        Random randomProdus  = new Random();
        //
        Comanda comandaRandom;
        ArticolComanda articolRandom;
        Double cantitateRandom;
        Integer produsPozRandom;

        for (int i=1; i <= nrComenzi; i++){
                comandaRandom = new Comanda(i, new Date());
                for (int j=1; j <= 1+randomNrArticole.nextInt(4) ; j++){
                        cantitateRandom = 1.0 + randomCantitate.nextInt(100);
                        produsPozRandom = randomProdus.nextInt(produse.size() -1);
                        articolRandom = new ArticolComanda(produse.get(produsPozRandom),
                                cantitateRandom);
                        comandaRandom.adaugaArticol(articolRandom);
                }
                comenzi.add(comandaRandom);
        }
}
```

- adăugați getteri, setteri și constructorii necesari(constructorul cu parametri pentru *nrProduse* și nrComenzi și constructorul fără pametri) și completații cu apelurile către metodele de generare dinamică randomizată a instanțelor de produse și comenzi;

```
public Registru() {
        /* Implicit sunt create 20 de produse și 100 de comenzi */
        generateRandomProduse(20);
        generateRandomComenzi(100);

}

public Registru(Integer nrProduse, Integer nrComenzi) {

        generateRandomProduse(nrProduse);
        generateRandomComenzi(nrComenzi);

}
```

Prin urmare, vor fi generate, în mod implicit, următoarele obiecte:
- 10 produse;
- 100 comenzi conținând între 1 și 5 articole.

Creați clasa de test *TestColectii* generându-i metoda *main* în care va fi instanțiată în mod direct clasa *Registru* și vor fi afișate comenzile generate:

```
public class TestColectii {
        public static void main(String[] args) {
                // Afiseaza comenzi generate
                System.out.println("Afizeaza comenzi : ****************************************");
                for(Comanda c: registru.getComenzi()){
                        System.out.println(c);
                        for(ArticolComanda a: c.getArticole()){
```

```
                                System.out.println(a);
                        }
                        System.out.println("----------------------------------------");
                }
        }
}
```

(iii) *Probleme de ordonare*: afișați produsele ordonate după *idProdus* și după *denumire*.

Soluția 1: Clasa *Produs* implementează interfața *Comparable*.
- în definiția clasei *Produs* se introduce sintagma implements *Comparable<Produs>*, se generează metoda *compareTo* (Source - *Override/Implement method – Comparable<Produs> - compareTo*) care va delega "principiul" de comparabilitate către câmpul *idProdus*:

```
public class Produs implements Comparable<Produs>{
        /*... ... ... ... ... ... ... ... ... ... ... ... ... ...*/
        @Override
        public int compareTo(Produs o) {
                // Produse ordonate dupa id
                return this.idProdus.compareTo(o.getIdProdus());
                // Produse ordonate dupa id reverse
                //return -this.idProdus.compareTo(o.getIdProdus());
        }
        /*... ... ... ... ... ... ... ... ... ... ... ... ... ...*/

}
```

- în clasa Registru se va introduce metoda *getProduseOrdonateDupaId* care va returna o colecție (Collection) generică de elemente de tip Produs, dar va folosi în implementare o colecție ordonată de tip *TreeSet*.

```
public Collection<Produs> getProduseOrdonateDupaId(){
        TreeSet<Produs> produseOrdonate = new TreeSet<Produs>();
        // Produs implementeaza Comparable dupa idProdus
        produseOrdonate.addAll(this.produse);
        return produseOrdonate;
}
```

Soluția 2: O clasă dedicată comparării instanțelor clasei **Produs**, clasă care implementează interfața *Comparator*.
- creați o clasă simplă *ComparatorProduseDenumire* care va implementa (rubrica *Interfaces* din dialogul *New – Class*) interfața *Comparator*. Completați metoda *compare* a acestei clase luând în calcul proprietarea *denumire*:

```
public class ComparatorProduseDenumire implements Comparator<Produs> {
        @Override
        public int compare(Produs p1, Produs p2) {
                return p1.getDenumire().compareTo(p2.getDenumire());
        }
}
```

- în clasa **Registru** se va introduce metoda *getProduseOrdonateDupaDenumire* care va returna o colecție (Collection) generică de elemente de tip **Produs**, dar va folosi în implementare clasa utilitară *Collections* ce va invoca o instanță a clasei *ComparatorProduseDenumire.*

```
public Collection<Produs> getProduseOrdonateDupaDenumire(){
        List<Produs> produseOrdonate = new ArrayList<Produs>();
        produseOrdonate.addAll(this.produse);
        Collections.sort(produseOrdonate, new ComparatorProduseDenumire());
        return produseOrdonate;
}
```

- adăugați secvența necesară în clasa de test:

```
System.out.println("Afizeaza produse ordonate dupa idProdus :
****************************************");
Collection<Produs> produseOrdonate = registru.getProduseOrdonateDupaId();
for(Produs p: produseOrdonate){
        System.out.println(p);
}
System.out.println("Afizeaza produse ordonate dupa denumire :
****************************************");
produseOrdonate = registru.getProduseOrdonateDupaDenumire();
for(Produs p: produseOrdonate){
        System.out.println(p);
}
```

- procedați similar pentru a afișa comenzile ordonate după valoare...

(iv) *Probleme de interogare 1 (operații de căutare simplă în colecții)*: căutați și afișați produsul cu id 5 și comanda cu id-ul 10. Semnalizați prin excepții situația în care nu există obiectele căutate.
- adăugați metoda *getProdus* în *Registru* parametrizată cu *idProdus* și care va semnaliza posibilitatea căutării eșuate prin excepții generice (poate arunca excepții instanțiate din clasa *Exception*). Căutarea propriu-zisă se va face prin operația *indexOf* a colecției, operație ce va primi ca argument un obiect-șablon de căutare ce va fi comparat cu obiectele din colecție folosind mecanismul *equals* (observați existența clauzei *throws* în semnătura operației);

```
public Produs getProdus(Integer idProdus) throws Exception{
        Produs p = new Produs();
        p.setIdProdus(idProdus);
        Integer pIndex = produse.indexOf(p);
        if (pIndex >= 0)
                return this.produse.get(pIndex);
        else
                throw new Exception("No data found: Produs inexistent!");
}
```

- adăugați similar metoda *getComanda* în *Registru*, semnalizarea căutării eșuate va fi realizată prin intermediul instanțelor clasei *RuntimeException* (observați inexistența clauzei *throws* în semnătura operației):

```
public Comanda getComanda(Integer idComanda){
        Comanda c = new Comanda();
        c.setIdComanda(idComanda);
        Integer cIndex = comenzi.indexOf(c);
        if (cIndex >= 0)
                return this.comenzi.get(cIndex);
        else
```

```
                throw new RuntimeException("No data found: Comanda inexistenta!");
}
```

Existența/inexistența clauzei *throws* are consecințe asupra modului de testare:

```
System.out.println("Afizeaza produsul cu id 5 si comanda 10:
****************************************");
try {
        System.out.println(registru.getProdus(10000));
} catch (Exception e) {
        e.printStackTrace();
}
System.out.println(registru.getComanda(10));
```

(v) *Probleme de interogare 2 (operații cu mulțimi)*: găsiți produsele vândute prin comenzile 4 și 5.
- mai întâi adăugați în clasa Registru metoda *getProduseDinComanda* parametrizată cu id-ul comenzii și care va returna o listă cu produsele din articolele comenzii indicate (observați implicarea metodei *getComanda*):

```
public List<Produs> getProduseDinComanda(Integer idComanda) {
        Comanda c = getComanda(idComanda);
        List<Produs> produse = new ArrayList<Produs>();
        for (ArticolComanda a : c.getArticole())
                produse.add(a.getProdus());
        return produse;
}
```

- apoi adăugați în clasa Registru metoda *getProduseDinReuniuneComenzi* parametrizată cu id-urile comenzilor și care va returna o listă cu produsele din articolele comenzilor indicate (observați folosirea operației *addAll* pentru reuniunea colecțiilor rezultate în urma apelului metodei *getProduseDinComanda*):

```
public Collection<Produs> getProduseDinReuniuneComenzi(Integer idComanda1,
                Integer idComanda2) {
        List<Produs> produse_c1 = this.getProduseDinComanda(idComanda1);
        List<Produs> produse_c2 = this.getProduseDinComanda(idComanda2);

        Set<Produs> produse_c1_union_c2 = new TreeSet<Produs>();
        produse_c1_union_c2.addAll(produse_c1);
        produse_c1_union_c2.addAll(produse_c2);

        return produse_c1_union_c2;
}
```

- procedați similar pentru introducerea în Registru a metodelor *getProduseComuneComenzi* (pentru intersecția colecțiilor folosiți operația *retainAll*) și *getProduseDiferentaComenzi* (pentru diferența colecțiilor folosiți operația *removeAll*):

```
public Collection<Produs> getProduseComuneComenzi(Integer idComanda1,
                Integer idComanda2) {
        List<Produs> produse_c1 = this.getProduseDinComanda(idComanda1);
        List<Produs> produse_c2 = this.getProduseDinComanda(idComanda2);

        Set<Produs> produse_c1_union_c2 = new TreeSet<Produs>();
```

```
        produse_c1_union_c2.addAll(produse_c1);
        produse_c1_union_c2.retainAll(produse_c2);

        return produse_c1_intersect_c2;
}

public Collection<Produs> getProduseDiferentaComenzi(Integer idComanda1,
                Integer idComanda2) {
        List<Produs> produse_c1 = this.getProduseDinComanda(idComanda1);
        List<Produs> produse_c2 = this.getProduseDinComanda(idComanda2);

        Set<Produs> produse_c1_union_c2 = new TreeSet<Produs>();
        produse_c1_union_c2.addAll(produse_c1);
        produse_c1_union_c2.removeAll(produse_c2);

        return produse_c1_minus_c2;
}
```

Testarea acestor metode ar trebuie făcută astfel:

```
System.out.println("Afizeaza produse din comenzile 4 si 5: ************************************");
                Collection<Produs> produse_c4_union_c5 = registru.getProduseDinReuniuneComenzi(4,5);
                for (Produs p: produse_c4_union_c5)
                        System.out.println(p);
System.out.println("Afizeaza produse comune comenzilor 4 si 5: ***********************************”);
                Collection<Produs> produse_c4_intersect_c5 = registru.getProduseComuneComenzi(4, 5);
                for (Produs p: produse_c4_intersect_c5)
                        System.out.println(p);
System.out.println("Afizeaza produse din comenda 4, dar nu si din 5: ****************************");
                Collection<Produs> produse_c4_minus_c5 = registru.getProduseDiferentaComenzi(5, 4);
                for (Produs p: produse_c4_minus_c5)
                        System.out.println(p);
```

(vi) *Probleme de indexare 4 (operații de căutare folosind mape)*: căutați și afișați produsul cu idul 8 și/sau denumirea “Produs_8”.
O alternativă la colecțiile indexate simplu (liste în care căutarea se poate face folosind operația *indexOf*, așa cum s-a exemplificat mai sus) o reprezintă *mapele* în care obiectele-valoare pot fi asociate cu orice fel de obiecte-chei de căutare.
- În acest sens, în Registru vor fi introduse pentru gestiunea produselor și comenzilor următoarele mape:

```
Map<Integer, Produs> produseMapId = new HashMap<Integer, Produs>();
Map<String, Produs> produseMapdenumire = new TreeMap<String, Produs>();
```

Mapele TreeMap (la fel ca și TreeSet-urile) stochează elementele ordonat după cheia de căutare.
- Metodele de căutare (bazate pe oerațiile *get* specifice mapelor) ar putea arăta astfel (în scop de testare, a fost introdus și codul de populare a mapelor care invocă operația *put*):

```
public Produs getProdusMapId(Integer idProdus) {
        if (produseMapId.isEmpty()) {
                for (Produs p : this.produse)
                        produseMapId.put(p.getIdProdus(), p);
        }
        return produseMapId.get(idProdus);
}

public Produs getProdusMapDenumire(String denumire) {
        if (produseMapdenumire.isEmpty()) {
                for (Produs p : this.produse)
```

```
                        produseMapdenumire.put(p.getDenumire(), p);
        }
        return produseMapdenumire.get(denumire);
}
```

Testarea se face printr-o secvență extrem de simplă:

```
System.out.println("Afizeaza produse cu id 8 sau numele Produs_8:
**********************************");
System.out.println(registru.getProdusMapId(8));
System.out.println(registru.getProdusMapDenumire("Produs_8"));
```