Clase și obiecte

1. Scopul lucrării

Obiectivele de învățare ale acestei sesiuni de laborator sunt cunoașterea și stăpânirea:

- Modului corect de declarare a claselor, definirea și instanțierea variabilelor de tip referință, apelul corect al metodelor
- Variabilelor Java și a modului de folosire a acestora în expresii

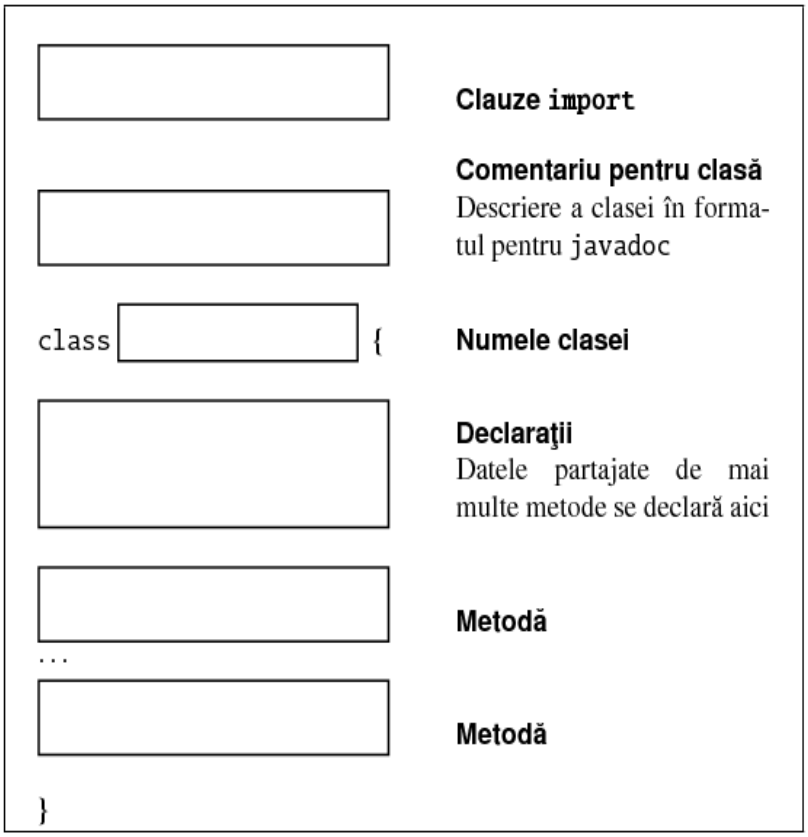
2. Clase simple

2.1. Declararea claselor

Clasele sunt cărămizile de bază în construcţia programelor Java. Clasele pot fi comparate cu planurile pentru clădiri. În loc să specifice structura clădirilor, clasele descriu structura "lucrurilor" dintr-un program. Aceste lucruri sunt apoi create ca obiecte software ale programului. Lucrurile care merită să fie reprezentate sub forma claselor sunt de obicei substantivele importante din domeniu problemei. De exemplu, o aplicaţie „cărucior pentru cumpărături on-line" e probabil să conţină clase care reprezintă clienţi, produse, comenzi, linii de comenzi, cărţi de credit, adrese pentru livrare şi furnizori de produse.

Pentru declararea unei clase în Java folosiţi următoarea sintaxă:

```
[public] [abstract|final] class NumeClasa [extends NumeClasaParinte] [implements NumeInterfete]
  {
    // variabilele şi metodele sunt declarate în interiorul acoladelor clasei
  }
```



- O clasă poate avea vizibilitate **public** sau *implicită* (fără modificator de acces).
- Poate fi sau **abstract**, **final** sau *concretă* (fără modificator).
- Trebuie să folosiţi cuvântul cheie **class**, urmat de un identificator legal.
- Opţional clasa poate extinde una dintre clasele din ascendenţă. Implicit va extinde **java.lang.Object**.
- Opţional poate implementa oricâte interfeţe, separate prin virgulă.
- Variabilele şi metodele clasei se declară între acoladele exterioare '{}' care urmează după identificatorul clasei.
- Fiecare fişier sursă .java poate conţine doar o singură clasă publică. Un fişier sursă poate conţine orice număr de clase cu nivelul de acces implicit.
- Numele fişierului sursă trebuie să fie identic cu cel al clasei publice.

2.2. Constructori

La crearea unei noi instanţe a unei clase (un obiect nou), folosind cuvântul cheie **new**, este invocat un constructor pentru clasa respectivă. Constructorii sunt folosiţi pentru a iniţializa variabilele instanţă (câmpurile) unui obiect. Constructorii sunt asemănători metodelor, dar există câteva diferenţe importante.
Numele constructorului este numele clasei. Un constructor trebuie să aibă acelaşi nume cu clasa în care se află.
Constructorul implicit. Dacă nu definiţi un constructor pentru o clasă, compilatorul creează automat un implicit, fără parametri. Constructorul implicit invocă constructorul implicit pentru părinte şi iniţializează toate variabilele instanţă la valorile implicite (zero pentru tipurile numerice, null pentru referinţe la obiecte şi false pentru booleene).
Constructorul implicit este creat numai atunci când nu sunt definiţi constructori. Dacă definiţi constructori pentru o clasă, atunci nu se mai creează automat un constructor implicit.
Diferenţe între metode şi constructori :

- Constructorii nu au tip returnat. Valoarea este obiectul însuşi, aşa că nu este nevoie să se indice o valoare returnată.
- Nu există instrucţiune return în corpul constructorului.
- Prima linie din corpul constructorului trebuie să fie ori un apel la un alt constructor al aceleiaşi clase (folosind **this**), ori un apel al constructorului superclasei (folosind **super**). Dacă prima linie nu este nici unul dintre apeluri, compilatorul inserează automat un apel la constructorul fără parametri al superclasei.

Aceste diferenţe de sintaxă dintre un constructor şi o metodă sunt uneori greu de văzut în sursă. Poate ar fi fost mai bine să existe un cuvânt care să marcheze clar constructorii, aşa cum sunt în unele limbaje.
this(...) – Apelează un alt constructor din aceeaşi clasă. Adesea un constructor cu mai putini parametri apelează un constructor cu mai mulţi parametri dând valori implicite parametrilor care nu sunt prezenţi. Folosiţi acest apel pentru constructori din aceeaşi clasă.
super(...) – Folosiţi *super* pentru a apela un constructor dintr-o clasă părinte. Apelul constructorului pentru superclasă trebuie să fie prima instrucţiune din corpul unui constructor. Dacă constructorul implicit al superclasei satisface nevoile, atunci nu este nevoie să faceţi apelul, deoarece acesta se va face automat (*super* va fi folosit şi exemplificat la capitolul despre moştenire).

Exemplu de apel explicit al constructorului this:

```
public class Point {
    int x;
    int y;

    //============ Constructor cu parametri
    public Point(int px, int py)  {
        x = px;
        y = py;
    }

    //============ Constructor fara parametri
    public Point() {
        this(0, 0);  // Apeleaza constructorul cu parametri; creează punctul (0, 0)
    }
    . . .
}
```

2.3. Declararea metodelor

O sintaxă generală pentru declararea metodelor este:

```
[modificatori] tip_returnat nume_metoda (lista_parametri) [clauza_throws]
 {
    [lista_instrucţiuni]
 }
```

Tot ce este între paranteze pătrate [] este opţional. Bineînţeles că nu scrieţi parantezele pătrate în codul sursă; aici ele sunt folosite pentru a indica elementele opţionale. O declaraţie minimală de metodă cuprinde:

- **Modificatori**: set de cuvinte cheie ce definesc accesul la metode (**modificatori de access**) sau anumite proprietăți speciale ale metodelor (**modificatori non-access**).
- **Tipul returnat**: tipul returnat este fie un tip Java valid (primitiv sau clasă) sau **void** dacă nu se returnează nici o valoare. Dacă metoda declară un tip returnat, atunci fiecare cale de ieşire din metodă trebuie să aibă o instrucţiune **return**.
- **Numele metodei**: numele metodei trebuie să fie un identificator Java valid.
- **Lista de parametri**: parantezele care urmează după numele metodei conţin zero sau mai multe perechi tip/identificator care constituie lista de parametri. Fiecare parametru este separat cu o virgulă. Lista de parametri poate fi vidă.
- **Throws**: o listă de exceptii aruncate de metodă
- **Acoladele**: corpul metodei este cuprins între acolade. În mod normal corpul metodei conţine o listă de instrucţiuni Java separate prin punct-şi-virgulă care se execută secvenţial. Tehnic, totuşi, corpul metodei poate fi vid.

Numele metodei combinat cu lista de parametri pentru fiecare metodă dintr-o clasă trebuie să fie unic.
Unicitatea unei liste de parametri ia în considerare ordinea parametrilor.
Astfel că `int myMethod(int x, String y)` este diferită de `int myMethod(String y, int x)`.

2.3.1. Modificatorii de access

Vizibilitatea unei metode (cunoscută şi ca zona în care este accesibilă) defineşte ce obiecte o pot invoca şi dacă subclasele o pot suprascrie. Cei patru modificatori de vizibilitate sunt: `public`, `protected`, `private`, şi fără modificator. Păstrarea cât mai ascuns cu putinţă a metodelor unui obiect ajută la simplificarea API (Application Programming Interface: specificaţia care defineşte cum poate accesa programatorul metodele şi variabilele unui set de clase). Nu faceţi metoda mai vizibilă decât este necesar. Spre exemplu, dacă metoda urmează să fie suprascrisă într-o subclasă, dar nu va fi apelată niciodată de codul client, faceţi vizibilitatea `protected`, nu `public`. Dacă o metodă nu trebuie niciodată invocată de o altă clasă şi nu urmează să fie suprascrisă, faceţi-o `private`.

Lista modificatorilor de acces este definită în următorul tabel:

Modificator	Poate fi accesată de
`public`	Orice clasă
`protected`	Clasa care o deţine, orice subclasă, orice clasă din acelaşi pachet (package)
Fără modificator	Clasa care o deţine, orice clasă din acelaşi pachet (package)
`private`	Clasa care o deţine

2.3.2. Modificatorii non-access

Pe lângă modificatorii de acces, în plus, o metodă poate fi descrisă de următorul set de cuvinte cheie (denumit şi set al modificatorilor non-access).

Cuvinte Cheie	Descriere
Vizibilitate	Poate fi una dintre valorile: `public`, `protected`, sau `private`. Determină care clase pot invoca metoda.
`static`	Metoda poate fi invocată la nivel de clasă, în loc de nivelul instanţei clasei. Spre exemplu, `String.valueOf(35)` apelează `valueOf` pe clasa `String` în loc de un anumit obiect String. Desigur, metodele statice pot fi apelate şi pe instanţe de clasă (obiecte): `myString.valueOf(35)`.
`abstract`	Metoda nu este implementată. Clasa trebuie extinsă şi metoda trebuie implementată în subclasă.
`final`	Metoda nu poate fi suprascrisă într-o subclasă.
`native`	Metoda este implementată în alt limbaj.
`synchronized`	Metoda necesită să fie obţinut un monitor (*lock*) de către codul care o invocă înainte de execuţia metodei. Utilizat în cazul execuţiei *multi-thread*.

2.3.3. Parametrii sunt transmişi prin valoare

În Java, transmiterea unui argument al unui **apel de metodă** se face **prin valoare**.
La transmiterea unei valori primitive spre o metodă, se face o copie a valorii primitive. **Copia este de fapt manipulată de metodă.** Aşa că, deşi valoarea copiei poate fi schimbată în metodă, valoarea originală rămâne neschimbată.
La transmiterea unei **referinţe spre un obiect** sau a unei **referinţe spre un tablou** spre o metodă, **metoda manipulează de fapt o copie a acelei referinţe**. Aşa că, **metoda poate schimba atributele obiectului**. Dar, dacă re-asignează referinţa la un alt obiect sau un alt tablou, re-asignarea afectează doar copia, nu referinţa originală.

Un exemplu de transmitere a obiectelor este dat mai jos. Deşi, aparent cele două metode *modify1* şi *modify2* par să facă acelaşi lucru, rezultatele sunt diferite (discuţii la problema 4.1).

```
public class Person {
	private String name;
	public Person(String name) { this.name = name; }
	public String getName() { return this.name; }
	public void setName(String newName) { this.name = newName; }
	public static void modify1(Person p) {
		p = new Person("Modified Name");
		System.out.print(p.getName());
	}
	public static void modify2(Person p) {
		p.setName("Modified Name");
		System.out.print(p.getName());
	}
	public static void main(String[] args) {
		Person p = new Person("Initial Person");
		modify1(p);
		System.out.println("Method modify1(Person p):" + p.getName());
		modify2(p);
		System.out.println("Method modify2(Person p):" + p.getName());
	}
}
```

2.3.4. Supraîncărcarea metodelor

Supraîncărcarea metodelor implică folosirea unui termen pentru a indica semnificații diverse. Supraîncărcarea unei metode Java înseamnă că scrieţi mai multe metode cu acelaşi nume, dar cu argumente diferite.
Un exemplu de supraîncărcare a unei metode:

```
public int test(int i, int j) {
	return i + j;
}

public int test(int i, byte j) {
	return i + j;
}
```

3. Variabile

Variabilele sunt locaţii din memorie în care se păstrează valori. Variabilele sunt de mai multe feluri şi fiecare limbaj de programare abordează diferit caracteristicile acestora:

- **Numele variabilei**.
- **Tipul de dată** specifică natura informaţiei pe care o variabilă o poate memora. Java are două tipuri generale de date.
  - 8 tipuri de bază sau *primitive* (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`).
  - Un număr nelimitat de tipuri *obiect* (`String`, `Color`, `JButton`, ...). O variabilă obiect Java păstrează o *referinţă* (pointer) către obiect, şi nu obiectul propriu-zis, care este întotdeauna memorat în zona heap.
- **Domeniul** unei variabile stabileşte cine poate să o vadă. Domeniul unei variabile este determinat în mare măsură de structura programului: d.e., bloc, metodă, clasă, pachet, subclasă.
- **Durata de viaţă** este intervalul de timp dintre momentul creării şi respectiv al distrugerii unei variabile. Următoarele convenţii sunt esenţiale pentru înţelegerea funcţionării sistemului Java. Variabilele locale şi parametrii se creează la momentul iniţierii unei metode şi sunt distruse la momentul în care metoda execută return. Variabilele instanţă sunt create de constructorul `new` şi sunt distruse când nu mai există nici o referinţă către ele. Variabilele de clasă (static) sunt create la momentul încărcării clasei şi sunt distruse la terminarea programului.
- **Valori iniţiale**. Ce valoare primeşte o variabilă atunci când este creată? Există mai multe posibilităţi.
  - Nici o valoare iniţială. Variabilele **locale** Java **nu au valori iniţiale**. Totuşi compilatoarele Java fac o analiză simplă a fluxului de instrucţiuni pentru a se asigura de faptul că fiecărei variabile locale i se va atribui o valoare înainte de a fi utilizată. Mesajele de eroare privind variabilele neiniţializate sunt de obicei justificate; uneori veţi fi nevoiţi să atribuiţi o valoare iniţială variabilei deşi ştiţi că acest lucru nu este neapărat necesar.
  - O valoare iniţială specificată de utilizator. Java permite atribuirea de valori iniţiale în enunţul de declarare a unei variabile.
  - Variabilele instanţă şi statice au valori iniţiale implicite: `zero` pentru numere, `null` pentru obiecte, `false` pentru variabile de tip boolean.
- **Obligativitatea declaraţiilor**. Java, ca şi alte limbaje de programare, vă pretinde să *declaraţi* variabilele – să informaţi compilatorul despre tipul variabilei etc. Declaraţiile sunt importante deoarece ajută programatorul să construiască programe fiabile şi eficiente.
  - Declaraţiile permit compilatorului să găsească locurile în care o variabilă este utilizată eronat, d.e. parametri de tip eronat. Faptul că astfel de erori sunt găsite în timpul compilării ajută mult la reducerea timpului de testare a programelor. Anomaliile (*bugs*) nedepistate de compilator sunt mult mai greu de localizat şi există riscul de a nu fi găsite decât după ce programul a fost livrat clientului.

- O declaraţie este de asemenea locul ideal pentru a scrie un comentariu ce descrie variabila respectivă şi modul în care aceasta este utilizată.
- Deoarece declaraţiile furnizează compilatorului mai multe informaţii, acesta poate genera cod mai bun.

Variabile locale/instanţă/clasă

Limbajul Java prevede trei categorii de variabile:

- **Variabile locale** sunt declarate în interiorul unei metode, constructor sau bloc. La momentul introducerii unei metode, o zonă asociată metodei se plasează în *stiva de apeluri.* Această zonă conţine câte o înregistrare pentru fiecare variabilă locală şi fiecare parametru. Când este apelată metoda, fiecare înregistrare parametru este iniţializată cu valorile parametrului respectiv. La momentul terminării metodei zona este eliminată din stivă şi memoria devine disponibilă pentru următoarea metodă apelată. Parametrii sunt variabile locale esenţiale care sunt iniţializate cu valorile parametrilor actuali. Variabilele locale nu sunt vizibile în afara metodei.
- **Variabile instanţă** sunt declarate într-o clasă, dar în afara unei metode. Ele mai sunt numite şi *variabile membru* sau *variabile câmp* . Când un obiect este alocat în zona *heap*, se creează câte o înregistrare pentru fiecare valoare a variabilelor instanţă. Astfel, o variabilă instanţă este creată/distrusă odată cu obiectul căruia îi aparţine. Variabila instanţă este vizibilă în toate metodele şi în toţi constructorii care aparţin clasei care o definesc. În general trebuie declarată privată, dar i se poate conferi şi o vizibilitate sporită.
- **Variabile de clasă/statice** sunt declarate precizând cuvântul cheie `static` în interiorul unei clase, dar în afara unei metode. Există o singură copie per clasă, indiferent câte obiecte au fost create din ea. Ele sunt memorate în zona de memorie statică. În majoritatea cazurilor, variabilele statice se introduc în declaraţii `final` şi se utilizează pe post de constante publice sau private.

Caracteristica	**Variabile locale**	**Variabile instanţă**	**Variabile de clasă**
Unde se declară	Metodă, constructor, sau bloc.	Într-o clasă, dar în afara unei metode. De obicei `private`.	Într-o clasă, dar în afara unei metode. Trebuie declarate `static`. De obicei au şi atributul `final` când definesc valori constante.
Utilizare	Variabilele locale păstrează valori utilizate în calcule într-o metodă.	Variabilele instanţă păstrează valori ce trebuie referite în mai mult de o metodă. (d.e. componente ce păstrează valori de genul şiruri de caractere, variabile pentru realizarea desenelor, etc.), sau care sunt părţi esenţiale ale stării unui obiect ce trebuie să existe între invocarea a două metode diferite.	Variabilele clasă sunt utilizate în general pentru constante, variabile care nu îşi modifică niciodată valorile lor iniţiale.
Durata de viaţă	Sunt create la introducerea metodei sau a constructorului. Sunt distruse la ieşire.	Sunt create la momentul creării unei instanţe prin `new`. Sunt distruse când nu mai există nici o referire la obiectul care le conţine (ele sunt preluate de colectorul de reziduuri « garbage collector »).	Sunt create la pornirea programului. Sunt distruse la terminarea programului.

Domeniu/ Vizibilitate	Variabilele locale (inclusiv parametrii formali) sunt vizibili numai în metoda, constructorul sau blocul unde au fost declarate. Modificatorii de acces (`private`, `public`, ...) nu pot fi utilizaţi pentru variabile locale. Toate variabilele locale sunt efectiv private pentru blocul în care au fost declarate. Ele nu sunt vizibile din nici o altă parte a programului, cu excepţia metodei/blocului unde au fost declarate. Un caz special îl constituie o variabilă locală declarată în partea de iniţializare a unei instrucţiuni `for` ; aceasta are drept domeniu domeniul instrucţiunii respective.	Variabilele instanţă (*fields*) sunt vizibile de către toate metodele unei clase. Numărul claselor pentru care mai sunt vizibile este determinat de atributul lor de acces. Alegerea implicită în declararea lor ar trebui să fie `private` . Nici o altă clasă nu poate vedea variabilele instanţă private. Aceasta ar fi cea mai bună alegere. Pentru a păstra flexibilitatea reprezentării interne, şi pentru a întări consistenţa datelor se recomandă a defini metode pentru citirea respectiv iniţializarea variabilei dacă valorile trebuie aduse din afara clasei. Implicit (convenţie numită şi vizibilitatea pachetului) o variabilă poate fi văzută din orice clasă a aceluiaşi pachet. Cu toate acestea `private` este de preferat. `public` – poate fi văzută din orice clasă. In general este o idee greşită. `protected` – variabilele sunt vizibile din orice clasă de descendenţi. Este o alegere ieşită din comun şi probabil o alegere greşită.	La fel ca şi variabilele instanţă, dar deseori sunt declarate `public` pentru a oferi utilizatorilor clasei valorile constantelor.
Declarare	Declaraţia trebuie plasată într-o metodă sau bloc oriunde înainte de utilizare.	Oriunde la nivelul clasă (înainte sau după utilizare).	Oriunde la nivelul clasă împreună cu atributul `static`.
Valori iniţiale	Niciuna. Trebuie să i se atribuie o valoare înainte de prima utilizare.	*Zero* pentru numere, *false* pentru variabile booleene, *null* pentru referinţe la un obiect. Valorile pot fi atribuite într-o declaraţie sau într-un constructor.	La fel ca şi variabilele instanţă, dar în plus li se pot atribui valori într-un bloc special de iniţializare statică.
Acces din afară	Imposibil. Numele variabilelor locale este cunoscut numai în interiorul metodei.	Variabilele instanţă trebuie declarate `private` pentru a asigura ascunderea informaţiei, astfel ele nu mai pot fi accesate din afara clasei. Totuşi există situaţii mai rar întâlnite în care ele trebuie accesate din afara clasei; în aceste cazuri variabilele vor fi calificate de un obiect (d.e. myPoint.x).	Variabilele clasă sunt calificate de numele clasei (d.e., Color.BLUE). Ele pot fi calificate şi de către un obiect dar este un stil amăgitor (poate crea confuzie) .

Sintaxa numelor	Reguli standard	Reguli standard, dar deseori sunt prefixate pentru a face vizibilă diferenţa faţă de variabilele locale, d.e. `my`, sau `m` (pentru variabile membru) `myLength`, sau `this` ca şi în numele `this.length`.	Variabilele (constantele) `static public final` se scriu toate cu majuscule, altfel ele se conformează convenţiilor obişnuite de numire.

Accesarea variabilelor statice și apelul metodelor statice
Variabilele și metodele ne-statice nu pot fi accesate/apelate din interiorul metodelor statice. Pentru a accesa/apela variabilele și metodele ne-statice din interiorul unei metode statice, acest lucru se poate face doar dacă se creează un obiect prin intermediul căruia acestea pot fi accesate.

Câteva exemple:

- Metodele statice nu pot accesa variabile/metode ne-statice

```
class Ex{
	int size = 42;
	void go() {  }
	static void doMore() {
		int x = size;
		go();
	}
}
```

- Variabilele și metodele ne-statice pot fi accesate doar de un obiect

```
class Ex{
	int size = 42;
	void go() {  }
	static void doMore() {
		Ex f = new Ex();
		int x = f.size;
		f.go();
	}
}
```

- Metodele statice pot accesa doar variabile/metode care la rândul lor sunt statice

```
class Ex{
	static int count;
	static void woo() { }
	static void doMore() {
		woo();
		int x = count;
	}
}
```

Exemplu de variabile de clasă (statice)

```
class Dog {
	static int dogCount = 0;
	public Dog() {
		dogCount += 1;
	}
	public static void main(String[] args) {
		new Dog();
		new Dog();
		new Dog();
		System.out.println("Dog count is now " + dogCount);
	}
}
```

4. Mersul lucrării

4.1. Studiați și înțelegeți exemplele din laborator.

4.1.1. Parcurgeți codul din secțiunea 2.3.3.

a) Fără a compila și executa acest cod, încercați să stabiliți rezultatul obținut în cazul apelării celor două metode *modify1* și *modify2*.

b) Compilați și rulați codul dat ca exemplu. Comparați rezultatele cu ce ați stabilit anterior.

4.1.2. Ce mesaj ar fi afișat în ultimul exemplu cu variabile de clasă din secțiunea 3, dacă variabila `dogCount` ar fi una instanță și nu una statică?

4.2. Scrieți un program Java în care:

- Proiectați și implementați o clasă Autovehicul care să fie caracterizat prin: *marcă, culoare* (vezi clasa predefinită Color), *viteza curentă* (în km/oră), *treapta de viteză curentă, viteza maximă pe care o poate atinge, numărul de trepte de viteză disponibile*. Un autovehicul ar trebui să poată efectua următoarele acțiuni: *accelerare* – care are ca efect creșterea vitezei cu un număr de km/oră*, decelerare, schimbarea treptelor de viteză, oprire*.
- Simulați deplasarea unui autovehicul într-o metodă statică a unei clase TestDrive. Aceasta ar trebui să presupună acțiuni de accelerare/decelerare, schimbare de viteză, determinare număr de km parcurși, oprire etc. ale unui anumit autovehicul. Afișați informații despre starea curentă a autovehiculului după fiecare acțiune întreprinsă.

Indicații de implementare:

- Modificatorul de acces al variabilelor instanță să fie de tip private, iar cel al metodelor de tip public sau protected.
- Pentru accesul extern la variabilele instanță, implementați metode accesoare (*get*) și mutatoare (*set*) acolo unde este cazul.
- Definiți cel puțin doi constructori: cu și fără parametri.
- Scrieți metoda *toString* în clasa Autovehicul pentru a returna starea curentă a autovehiculului.

4.3. Pornind de la programul dezvoltat la punctul anterior, adăugați în clasa Autovehicul următoarele:

- O constantă caracteristică pentru clasa Autovehicul.
- O variabilă instanță referitoare la șoferul autovehiculului. Definiți clasa Sofer care să fie caracterizat prin *nume*, *prenume*, *vârstă*, *număr permis de conducere*.
- O variabilă instanță referitoare la rezervorul autovehiculului. Definiți clasa Rezervor care să fie caracterizat prin *capacitate maximă*, *nivel curent de umplere* și care să permită umplerea și golirea acestuia cu o anumită cantitate de combustibil.
- O metodă care să fie supraîncărcată.

4.4. Proiectați și implementați un program nou - la alegere - care să conțină minim două clase și să simuleze activități/fenomene din lumea reală (ex.: bilete și extragere loto; împrumut de cărți de la bibliotecă; managementul angajaților la o firmă etc.).