Capitolul2

Capitolul 2: clase, obiecte, metode 2
Alte instructiuni de control 2
Break 2
Continue 3
Bucle repetitive imbricate 4
Operatorul conditional 5
Programare orientata pe obiecte 5
Obiecte 5
Clasa 6
Forma generala a unei clase 7
Functii 11
Constructori 16
Constructori cu unul sau mai multi parametrii 17
Garbage Collection si finalizatori 19
Metoda finalize 20
Cuvantul cheie this 21
Pachete 22
Definirea pachetelor 23
Pachete si CLASSPATH 23
Importarea pachetelor 25

Capitolul 2: clase, obiecte, metode

Alte instructiuni de control

Break

Se poate uneori forta parasirea unei bucle repetitive(*for*, *while*), utilizand instructiunea **break**. In cele ce urmeaza, conform exemplelor anterioare am realizat o bucla for, dar de data aceasta nu mai exista conditie de continuare, deci oprirea nu poate avea loc decat din cadrul acestui for:

```
class BreakSample {
public static void main(String args[]) {
	int num;
	num = 10;
	// continua la "nesfarsit", conditia de continuare nu este
	for(int i=0; ; i++) {
		if(i >= num) break; // i a ajuns la 10, se iese din bucla
			System.out.print(i + " ");
		}
	System.out.println("S-a terminat bucla.");
	}
}
```

Acest program va genera urmatoarea secventa:

```
0 1 2 3 4 5 6 7 8 9 S-a terminat bucla.
```

Dupa cum se poate vedea acest tip de instructiune are loc in conjunctura cu instructiuni conditionale: altfel s-ar executa doar odata bucla, eliminand caracterul de repetitivitate al acesteia. Alte scopuri ale utilizarii acestei instructiuni pot fi, spre exemplu, citirea repetitiva unor caractere: in momentul in care ceea ce am citit este egal ca valoare cu caracterul ce marcheaza oprirea citirii (de exemplu ’q’), parasim imediat bucla repetitva in care are loc citirea.

```
char ch;
for( ; ; ) {
	ch = (char) System.in.read(); // citesc un caracter
	if(ch == 'q') break;
	}
```

Alt mod de a folosi break, pe langa cele mentionate, este ca si instructiune de salt la o anumita eticheta. Java nu are o instructiune clasica *goto* deoarece aceasta permite un flux logic nestructurat.

Programele care folosesc tipul acesta de instructiuni sunt greu de urmarit si de mentinut, de aceea expunerea este pur informativa, si nu recomand folosirea *break* in acest scop. Forma generala a unui *break* cu sens de goto este:

break eticheta;

In acest caz, *eticheta* este numele etichetei care identifica un bloc de instructiuni. Atunci cand se ajunge la instructiunea `break eticheta;` se transfera controlul acelui bloc indicat de numele *eticheta.*

Iata un exemplu pentru a ilustra pe scurt functionarea acestei instructiuni:

```
class BreakLabel
{
	public static void main(String args[])
	{
		int i;
		int count=0;
			et1: for(i=++count; ; i++)
			{
				System.out.println("\n i este " + i);
				if(i>2) break et1;
			}
			System.out.println("Dupa blocul for.");
	}
}
```

Se poate observa ca dupa trei rulari a instructiunii for, se va executa instructiunea **break** cu directionare catre eticheta *et1.* Se va afisa mesajele „i este 1”, „i este 2”, „i este 3” si se executa acel *break* cu salt la *et1*. Cand are loc acest fapt, controlul se va preda blocului imediat urmator si anume afisarea mesajului „Dupa blocul for”.

Continue

Aceasta instructiune este folosita intr-o bucla repetitiva permitand ca, in cazul in care nu vrem sa continuam cu instructiunile din acea bucla repetitiva, instructiuni care urmeaza dupa *continue*, sa trecem la pasul urmator. Practic se „sare” la urmatorul pas din instructiunea repetitva ignorand ce se intampla dupa *continue*.

De exemplu daca vrem sa afisam numerele pare dintre 0 si 50:

```
class ContinueSample
{
	public static void main(String args[])
	{
		int i;
		// afisez numere pare intre 0 si 50
		for(i = 0; i<=50; i++)
		{
			if((i%2) != 0) continue; // daca nu e par trec la
						//urmatorul pas din for
				System.out.println(i);
		}
	}
}
```

In acest exemplu for fi afisate numerele pare deoarece: pe instructiunea conditionala se intra doar daca restul impartirii numarului curent la 2 este zero. Atunci cand se intra (deci cand numarul este impar) se efectueaza *continue,* adica salt la urmatorul *i* din *for* si nu se mai executa afisarea *i*-ului curent.

Bucle repetitive imbricate

Inainte de a incheia acest subcapitol si anume al instructiunilor de baza vom exemplifica folosirea buclelor repetitve „una in interiorul celeilalte". Astfel in cele ce urmeaza putem calcula factorii fiecarui numar de la 2 la 50 in felul urmator: se parcurge fiecare numar de la 2 la 50; in clipa in care am facut un pas suntem in pozitia in care putem calcula factorii numarului pe care ne aflam – sa zicem X. In acest moment urmeaza o alta bucla pentru a parcurge numerele pana la X (poate fi evident optimizat). In ultima bucla se verifica restul impartirii lui X la numerele din intervalul [2..X), daca este zero inseamna ca am gasit un factor al lui X:

```
class Factors {
    public static void main(String args[])  {
        for(int i=2; i <= 50; i++)
        {
            System.out.print("Factorii lui " + i + ": ");
            for(int j = 2; j < i; j++)
            if((i%j) == 0) System.out.print(j + " ");
            System.out.println();
        }
    }
}
```

Operatorul conditional

Acest operator *?:* este un operator ternar mostenit din C. Permite inglobarea unei conditii intr-o expresie. Primul operand este separat de al doilea operand prin *?* in timp ce al doilea operand este separat de al treilea prin *:*. Primul operand trebuie sa fie de tip boolean. Al doilea si al treilea operand vor fi de orice tip de data insa de acelasi tip de data, sau convertibil catre acelasi tip de data.

Cum se evalueaza acest operator?

Se evalueaza primul operand, daca e *true* operatorul evalueaza al doilea operand si ii va folosi valoarea. Daca este *false* operatorul evalueaza al treilea operand si ii returneaza valoarea. De exemplu:

```
int max = (x>y) ? x : y;
String nume = (nume!=null) ? nume : ” necunoscut”;
```

max va lua valoarea lui x daca x este mai mare ca y, altfel va lua valoarea lui y.

Programare orientata pe obiecte

Obiecte

Obiectele sunt principalul concept din cadrul acestei tehnologii. Obiectele reale ce ne inconjoara, ca masa, televizorul, unelte, animale etc, au doua caracteristici in comun: *stare* si *comportare*. De exemplu animalele au stari (nume, colorit, specie) si comportare (mod deplasare, mod de a respira, de a se hrani). Bicicletele au stari (viteza curenta, greutate, cadenta pedalarii) si comportare (accelerare, franare, schimbarea treptelor de viteza).

Obiectele sunt modelate in programare prin una sau mai multe *variabile*, iar comportarea obiectelor este modelata prin *metode*.

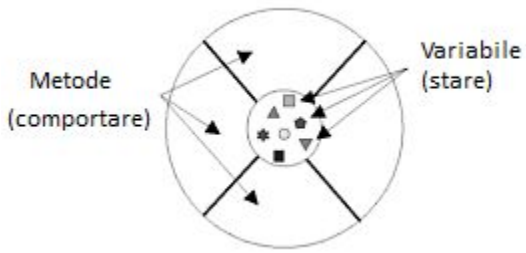

Figura 2.1. Obiectul modelat software

Tot ce se stie despre un obiect reprezinta starea lui exprimata prin variabile si ce poate obiectul face, reprezinta comportamentul sau reprezentat prin metode. De exemplu un obiect software care modeleaza o bicicleta va avea variabile ce indica starea curenta a bicicletei: viteza de 10 km/h, cadenta pedalelor de 90 rpm, si treapta de viteza curenta (a treia). Pe langa *variabile*, o bicicleta reprezentata software poate avea *metode* pentru a frana, schimba treapta de viteza. Totusi nu poate avea metoda de schimbare a vitezei pentru ca viteza este consecinta cadentei de pedalare, a franarii/sau nu, a pantei. Eventual poate avea metoda de redare a vitezei calculata pe baza acestor factori. Aceste metoda se numesc metode de **instanta** pentru ca evalueaza sau modifica o stare a unei anume biciclete si nu tuturor bicicletelor.

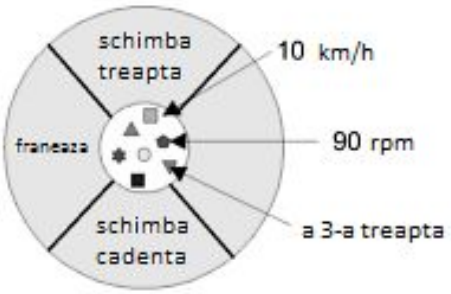

Figura 2.2 Un obiect bicicleta modelat ca obiect software

Diagramele de mai sus semnaleaza faptul ca variabilele formeaza centrul, nucleul obiectelor, iar metodele inconjoara acest nucleu. Impachetarea variabilelor obiectului pentru a proteja informatia continuta de ele, se numeste *incapsulare*.

Clasa

In lumea reala, multe obiecte pot fi categorizate ca fiind de acelasi *tip*. De exemplu automobilul detinut de cititor este la aproximativ la fel cu orice automobil din aceasta lume. Au roti pe care se deplaseaza, motorul cauzeaza deplasarea, si este nevoie de combustibil (sub o forma sau alta) pentru a alimenta motorul. Asemenea si o bicicleta, va avea roti, un sistem de pedalare, una sau mai multe viteze etc. Putem spune ca o anume bicicleta – de exemplu Atomik Mountain Bike – este o instanta a clasei *bicicleta*. Se cheama ca am construit un obiect cu anume caracteristici (culoare, rezistenta) dar care arata ca o bicicleta si se comporta asemenea.

Alta alegorie pentru a intelege mai bine diferenta intre obiect si clasa este o cel compus dintr-o matrita de imprimat bancnote si bancnota. Hartia, bancnota propriu zisa este *obiectul* ca instanta a *clasei* matrita care va imprima sute de hartii toate cu aceleasi caracteristici dar comportamente diferite.

Intr-o clasa, atat variabilele cat si metodele se vor numi membrii acelei clase.

Forma generala a unei clase

Clasa este creata folosind cuvantul cheie *class* ca mai jos:

```
class numeclasa
{
	//declar variabilelor clasei, ce vor fi disponibile in fiecare instanta
	type var1;
	type var2;
	// ...
	type varN;
	// declare metodele
	type metoda1(parametri)
	{
		//corpul metodei 1
	}
	type metoda2(parametri)
	{
		//corpul metodei 2
	}
	// ...
	type metodaN(parametri)
	{
		//corpul metodei N
	}
}
```

Acesta este sintaxa generala de definire a unei clase, se poate ca o clasa sa nu contina decat o variabila, sau o metoda, sau niciuna.

Pentru a ilustra conceptul de clasa, vom scrie o clasa ce cuprinde informatiile despre automobile. Aceasta clasa se numeste **Automobil**, si va contine trei tipuri de informatii ca: model, consum, viteza maxima. Mai jos avem definitia clasei cu cele trei variabile:

```
class Automobil
{
	String model;	//combi, SUV, MCV, caminon etc
	int consum; / 5 .. 15
	int viteza; //viteza maxima ce poate fi atinsa: 180 .. 340
}
```

Noul tip de data se numeste *Automobil*, si orice obiect de acest tip va contine cele trei variabile membru. Atentie, codul de mai sus este doar o descriere a tipului si nu prezinta crearea unui obiect.

Pentru a **instantia** un nou obiect de tipul Automobil se va folosi operatorul ***new***:

```
Automobil minivan = new Automobil();  //Creez un nou obiect
```

In urma acestei instructiuni, minivan devine instanta clasei Automobil. De fiecare data cand cream o nou instanta a unei clase, se aloca acelui obiect nou creat un spatiu de memorie necesar membrilor clasei din care obiectul face parte. Fiecare obiect de tip **Automobil** va contine propriile copii ale instantelor variabilelor *model*, *consum*, *viteza*. Pentru a accesa aceste variabile se va folosi operatorul „.”. Acesta expune membrii (atat variabile cat si metode) unui obiect:

```
object.member
```

Spre exemplu putem seta minivan-ului un anume consum:

```
minivan.consum = 10;
```

In sectiunea de cod ce urmeaza vom vedea cum in alta clasa, utilizam obiecte de tip **Automobil**:

```
class Automobil
{
	String model;	//combi, SUV, MCV, caminon etc
	int consum; // 5 .. 15
	int viteza; //viteza maxima ce poate fi atinsa: 180 .. 340
}
public class AutomobilDemo
{
	public static void main(String[] args)
	{
		Automobil minivan = new Automobil();
		minivan.model = "Logan MCV";
		minivan.consum =10;
		minivan.viteza = 180;
		System.out.println(minivan.model + " are un consum de "
		+ minivan.consum + " si viteza maxima " + minivan.viteza);
	}
}
```

Inainte de a trece mai departe vom prezenta un alt exemplu in care lucram cu doua obiecte, instante ale aceleasi clase. Scopul este pentru a clarifica faptul ca variabilele continute de un obiect pot diferi ca si valoare de variabilele continute de alt obiect, chiar daca cele doua obiecte au aceeasi clasa, deci sunt de acelasi tip:

```
class Automobil
{
	String model;	//combi, SUV, MCV, caminon etc
	int consum; // 5 .. 15
	int viteza; //viteza maxima ce poate fi atinsa: 180 .. 340
}
public class AutomobilDemo
{
	public static void main(String[] args)
	{
		Automobil minivan = new Automobil();
		Automobil masinasport = new Automobil();
		minivan.model = "Logan MCV";
		minivan.viteza = 180;

		masinasport.model = "BMW";
		masinasport.viteza = 320;

		System.out.println(minivan.model +" are viteza maxima " +
minivan.viteza);
		System.out.println(masinasport.model +" are viteza maxima "
+ masinasport.viteza);
	}
}
```

In urma rularii acestui cod se va obtine:

```
Logan MCV are viteza maxima 180
BMW are viteza maxima 320
```

Ce se intampla la crearea obiectelor?

Atunci cand apelam operatorul new ca in instructiunea:

```
Automobil minivan = new Automobil();
```

prima data are loc declararea variabilei minivan: `Automobil minivan`

iar apoi instantierea unui nou obiect `new Automobil()` pentru ca mai apoi copia noului obiect sa fie atribuita variabilei minivan. Operatorul **new** aloca dinamic memorie la momentul rularii programului, si returneaza referinta obiectului pentru care a alocat memorie. Aceasta referinta este adresa din memorie a noului obiect alocat. Astfel in Java, toate obiectele trebuie alocate dinamic.

Pentru a disocia cele doua notiuni de declarare si instantiere separam instructiunea de mai sus:

```
Automobil minivan;  //declaram unu obiect de tip referinta Automobil
Minivan = new Automobil(); //alocam memorie pentru un obiect de tip
                           //Automobil
```

Asignarea obiectelor

In cazul obiectelor, operatorul de asignare si anume „=" actioneaza altfel decat in vazul variabilelor primitive de tip **int**. Atunci cand asignam o variabila de tip primitiv cu valoarea altei variabile de tip primitiv, variabila din stanga operatorului primeste o ***copie*** a valorii variabilei din dreapta.

Atunci cand asignam *un obiect* catre alt obiect, lucrurile se schimba, deoarece obiectele inglobeaza mai multe variabile. De exemplu:

```
Automobil masina1 = new Automobil();

Automobil masina2 = masina1;
```

La prima vedere, este usor de spus ca *masina1* si *masina2* se refera la obiecte diferite, insa nu este deloc asa: atat *masina1* cat si *masina2* se vor referi la acelasi obiect. De aceea atunci cand vom afisa cosumul ambelor vom avea „surpriza" sa constatam ca este acelasi.

```
System.out.println(masina1.consum);

System.out.println(masina2.consum);
```

Alt exemplu pentru a intelege si mai bine asignarea:

```
Automobil masina1 = new Automobil();

Automobil masina2 = masina1;

Automobil masina3 = new Automobil;

masina2 = masina3; //acum atat obiectul masina2 si masina3 se
                   //refera la acelasi obiect, iar masina1 este
                   //neschimbat si independent de celelalte doua
```

Functii

In exemplele de mai sus, clasa Automobil contine date sub forma unor variabile, insa nici o metoda. Desi clasele ce contin doar variabile sunt perfect valabile, majoritatea vor contine metode, pentru asigurarea unui flux logic.

Metodele sunt functii, subrutine care trateaza, manipuleaza date definite in clase, pentru a asigura logica dorita. O metoda consta din semnatura acesteaia si corpul ei. Semnatura metodei se refera la *tipul de data* returnat, *nume* si *lista parametrilor* iar *corpul* inseamna blocul de instructiuni executat ori de cate ori functia este apelata.

O metoda contine una sau mai multe instructiuni, gandite ca impreuna sa efectueze o anumita sarcina. Metodele au nume, prin care sunt identificate in cadrul clasei, iar dupa nume intotdeauna urmeaza paranteze. Intre paranteze sunt declarati parametrii metodei (variabile de diverse tipuri). Aceste variabile vor lua valori in momentul apelului metodei. O metoda generala are forma:;

```
tip_de_returnat nume(lista de parametrii)
{
      //corpul metodei
}
```

Tipul_de_returnat reprezinta orice tip de data valid, inclusiv clase pe care le creati; ca exemple de tip de data avem int, Integer, String, double, Automobil. In cazul in care metoda nu returneaza nici un tip de data acesta va fi ***void***. Numele metodei poate fi orice identificator valabil, si respecta ca si formare, denumirea variabilelor (a se revedea conditiile de numire a variabilelor). Lista de parametrii este o secventa de declaratii de variabile/parametrii ce vor fi vizibili in cadrul metodei, separati prin virgula:

```
int myFunction(double param1, Integer param2, Automobil minivan)
{
      //... instructiuni din corpul metodei
      return param2;
}
```

Adaugarea unei metode intr-o clasa

Sa luam spre exemplu clasa Automobil de mai sus, si clasa AutomobilDemo, la care mai adaugam o metoda pe langa metoda deja existenta si anume PrintProperties. Aceasta este declarata cu *static*, deoarece va fi apelata dintr-o metoda statica si anume *main*. Alt mod de a „rezolva” aceasta problema era sa cream un nou obiect de tip AutomobilDemo in main si sa apelam metoda prin acel obiect dupa cum vom vedea imediat dupa exemplul de mai jos:

```
class Automobil
{
	String model;	//combi, SUV, MCV, caminon etc
	int consum; // 5 .. 15
	int viteza; //viteza maxima ce poate fi atinsa: 180 .. 340
}
public class MethodDemo
{
	public static void main(String[] args)
	{
		Automobil minivan = new Automobil();
		Automobil masinasport = new Automobil();
		minivan.model = "Logan MCV";
		minivan.viteza = 180;
		masinasport.model = "BMW";
		masinasport.viteza = 320;
		PrintProperties(minivan);
		PrintProperties(masinasport);
	}
	static void PrintProperties(Automobil vehicul)
	{
		//verificam sa nu avem obiecte null
		//altfel avem eroare la accesarea membrilor lui

		if (vehicul == null)
		{
			System.out.println("Introduceti un obiect instantiat!");
			return;
		}
		System.out.println("Masina "+ vehicul.model + " are viteza maxima
" + vehicul.viteza);

	}
}
```

In acest exemplu se poate observa declararea metodei PrintProperties ce are ca parametru un obiect de tip Automobil:

```
static void PrintProperties(Automobil vehicul)
```

Aceasta inseamna ca atunci cand vom apela aceasta metoda, va trebui sa specificam intre paranteze doar obiecte de tip Automobil, sau care deriva din clasa Automobil (vom detalia in capitolul urmator). Mai jos este apelul functiei cu un parametru de tipul Automobil.

```
PrintProperties(minivan);
```

In exemplul de mai sus avem o functie care nu returneaza nici o valoare. Sa vedem si un exemplu de functie ce returneaza o valoare care va fi folosita ulterior:

```
class Rectangle
{
	public int width; //latimea dretunghiului
	public int height; //inaltimea dreptunghiului

	static int GetArea(Rectangle aRectangle)
	{
		if (aRectangle == null)
		{
			System.out.println("Introduceti un obiect instantiat!");
			return -1;
		}
		return aRectangle.width*aRectangle.height;

	}
	static int GetPerimeter(Rectangle aRectangle)
	{
		if (aRectangle == null)
		{
			System.out.println("Introduceti un obiect instantiat!");
			return -1;
		}
		return 2*(aRectangle.width+aRectangle.height);
	}
}
```

```
public class MethodDemo
{
	public static void main(String[] args)
	{
		//exemplu pentru doua dreptunghiuri diferite
		Rectangle myRectangle = new Rectangle();
		Rectangle myOtherRectangle = new Rectangle();
		myRectangle.width = 10;
		myRectangle.height = 7;
		myOtherRectangle.width = 12;
		myOtherRectangle.height = 20;


		System.out.println("Aria primului dreptunghi " +
Rectangle.GetArea(myRectangle));
		System.out.println("Aria celuilalt dreptunghi " +
Rectangle.GetArea(myOtherRectangle));


		System.out.println("Perimetrul primului dreptunghi " +
Rectangle.GetPerimeter(myRectangle));
		System.out.println("Perimetrul celuilalt dreptunghi " +
Rectangle.GetPerimeter(myOtherRectangle));
	}
}
```

In acest exemplu, metodele se vor declara in clasa Rectangle si vor fi apelate din interiorul metodei main din clasa MethodDemo. Se observa faptul ca apelul functiilor comporta in fata numelui lor si numele clasei: *Rectangle*. Aceasta pentru ca metodele sunt statice, iar in acest caz o metoda statica poate fi apelata fara sa fie nevoie de instantierea unui obiect de acel tip. Metodele statice si variabilele statice sunt aceleasi per clasa, altfel spus variabilele membre clasei Rectangle vor avea aceleasi valori pentru orice obiect de aceasta clasa. Daca un obiect de tip Rectangle A, modifica valoarea lui width la 5, si width ar fi declarat *static* in Rectangle, atunci si un alt obiect B va avea ca width tot valoarea 5.

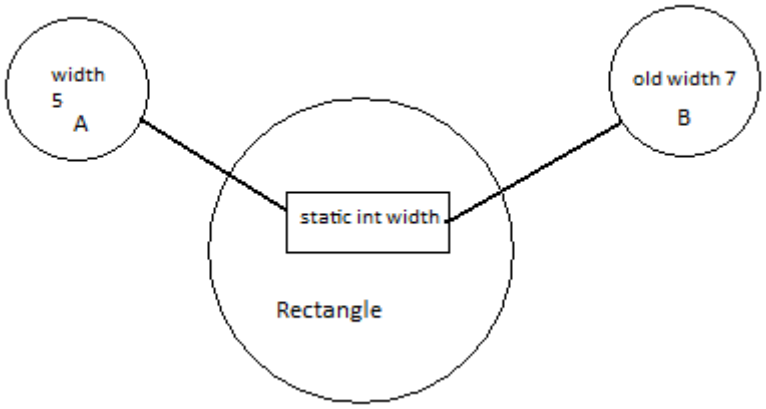

Figura 2.3 sensul cuvantului cheie static

Alt fapt de remarcat este ca valoarea returnata de functie poate fi folosita imediat in expresii cum ar fi alcatuirea unui String sau in asignari, urmand ca variabila care ia valoarea returnata de functie sa fie utilizata mai departe:

```
	System.out.println("Perimetrul primului dreptunghi " +
Rectangle.GetPerimeter(myRectangle));
```

Aceasta instructiune poate fi scrisa si asa:

```
	int perimeter = Rectangle.GetPerimeter(myRectangle);

	System.out.println("Perimetrul primului dreptunghi " + perimeter);
```

Utilizarea mai multor parametrii

In exemplele de mai sus am folosit doar un singur parametru (de tip rectangle). In cele ce urmeaza vom avea doi si chiar trei parametrii de tipuri diferite pentru a evidentia utilizarea parametrilor.

```
class Numbers
{
	int GetSumofTwoInt(int a, int b)
	{
		return a+b;
	}
	int GetSumofThreeInt(int a, int b, int c)
	{
		return a+b+c;
	}
```

```
	double GetSumofTwoDouble(double a, double b)
	{
		return a+b;
	}
}
public class MoreParameters
{
	public static void main(String[] args)
	{
		Numbers number = new Numbers();
		System.out.println("Suma 6 + 7 = " + number.GetSumofTwoInt(6,7));
		System.out.println("Suma 6 + 7 +8 = " +
number.GetSumofThreeInt(6,7,8));
		System.out.println("Suma 6.6 + 7.2 = " +
number.GetSumofTwoDouble(6.6,7.2));
	}
}
```

Dupa cum se poate observa in acest exemplu toate cele trei metode sunt apelate dupa instantierea obiectului *number*. Aceasta deoarece ele nu sunt statice, astfel ca acum nu mai putem utiliza numele clasei pentru a apela functia. Ca exercitiu incercati sa refaceti metodele astfel ca ele sa fie statice, eliminand cu totul utilizarea obiectului *number*.

Constructori

In exemplele anterioare, in cele in care am folosit clasa *Automobil*, am initializat membrii clasei ca *model*, *viteza* in functia in care am si creat noul obiect *minivan*:

```
minivan.model = "Logan MCV";
minivan.viteza = 180;
```

Modul acesta de lucru nu este unul profesional, deoarece necesita o atentie in plus din partea programatorului, acesta putand omite initializarea unor membrii (ce s-ar intampla daca avem zece membrii in aceeasi clasa?).

Un *constructor* este o metoda care initializeaza un obiect atunci cand acesta este creat (cu operatorul ***new***). Are acelasi nume ca si clasa si nu returneaza nimic. In mod normal constructorii sunt folositi pentru a crea proceduri de initializare pentru a „da o forma” noului obiect. Mai jos avem un exemplu de un constructor pentru clasa Automobil:

```
class Automobil
{
	String model;	//combi, SUV, MCV, caminon etc
	int consum; // 5 .. 15
	int viteza; //viteza maxima ce poate fi atinsa: 180 .. 340
	//Acesta este constructorul clasei Automobil apelat la instantierea
	//unui nou obiect de tip Automobil
	public Automobil()
	{
		model = "Necunoscut";
		consum = 0;
		viteza = 0;
	}
}
...
public class AutomobilDemo
{
	public static void main(String[] args)
	{
		Automobil minivan = new Automobil(); //Acesta este locul in care
							//apelam constructorul
	....
```

Constructori cu unul sau mai multi parametrii

In exemplul anterior, a fost folosit un constructor fara parametrii. Desi este ok, in multe situatii, exista anumite momente cand trebuie sa avem constructori prin care sa manipulam valorile initiale ale unui obiect. Pentru aceasta, vom transmite aceste valori ca parametrii unui constructor al aceleasi clase, la fel cum se intampla la apelul unor functii cu diversi parametrii:

```
class Automobil
{
	String model;	//combi, SUV, MCV, caminon etc
	int consum; // 5 .. 15
	int viteza; //viteza maxima ce poate fi atinsa: 180 .. 340
	//constructor fara parametrii
	public Automobil()
	{
		model = "Necunoscut";
		consum = 0;
		viteza = 0;
	}
	//constructor cu un parametru
	public Automobil(String modelulinitial)
	{
		model = modelulinitial;
		consum = 0;
		viteza = 0;
	}
	//constructor cu doi parametrii
	public Automobil(String modelulinitial, int vitezainitala)
	{
		model = modelulinitial;
		consum = 0;
		viteza = vitezainitala;
	}
}
public class AutomobilDemo
{
	public static void main(String[] args)
	{
		//Constructor cu un parametru
		//ulterior mai trebuie sa initializez separat viteza
		Automobil minivan = new Automobil("Logan MCV");
		minivan.viteza = 180;
```

```
            //Constructor cu doi parametrii, initializez tot
            //direct prin apelul acestui
            Automobil masinasport = new Automobil("BMW",320);


            System.out.println(minivan.model +" are viteza maxima " +
minivan.viteza);
            System.out.println(masinasport.model +" are viteza maxima " +
masinasport.viteza);
      }
}
```

In exemplul de mai sus avem un constructor fara parametrii (implicit sau *default*) si doi constructori cu unul respectiv doi parametrii. Evident tipul de data al parametrilor poate varia, exact ca in cazul metodelor.

Am precizat ca un costructor seamana cu o functie. Atunci cand este apelat un costructor? Atunci cand folosim operatorul ***new*** pentru a crea un nou obiect, mai intai se apeleaza constructorul a caror parametrii sunt de tipul celor cu care s-a facut apelul, si apoi se creează referinta pentru acel obiect: `Automobil minivan = new Automobil("Logan MCV");`

Cum eliminam atunci referinta unui obiect atunci cand nu mai este nevoie de el?

Garbage Collection si finalizatori

Atunci cand instantiem un obiect, se creează o zona de memorie care va fi alocata acelui obiect. Memoria insa, nu este infinita, si memoria libera trebuie bine manipulata. Se poate uneori ca *new* sa esueze pentru ca nu exista memorie suficienta pentru a crea obiectul. De aceea, o componenta cheie a limbajului este un mecanism de alocare dinamic, si de recuperarea a memoriei de la obiectele nefolosite, facand ca acea memorie sa fie disponibila altor noi obiecte. In C++, acest lucru este facut manual, de catre programator, cu anumite instructiuni gen *delete*, sau *free*.

Java foloseste altceva si anume *garbage collector* (GC). GC este un mecanism ce recupereaza memoria de la obiectele ce nu mai sunt folosite (si aici sunt mai multe scenarii), si o face disponibila pentru alte noi obiecte. Pentru eficienta, GC va rula cand doua conditii sunt indeplinite: exista obiecte care trebuie reciclate si este nevoie ca ele sa fie reciclate. De aceea nu se poate preciza, sau controla cand GC va rula.

In acest caz cum controlam ce se intampla la stergerea obiectului?

Metoda finalize

Se poate defini o metoda care va fi apelata automat inainte ca obiectul sa fie distrus de GC. Aceasta se numeste **finalize()**, si se utilizeaza cand dorim ca obiectul sa fie distrus intr-un mod ordonat. De exemplu se poate folosit metoda *finalize()* pentru a inchide un fisier eventual deschis de acel obiect.

Forma generala a metodei este:

```
protected void finalize()
{
      //codul la distrugerea obiectului
}
```

Aici cuvantul cheie *protected* este un specificator de acces ce nu permite accesul la aceasta metoda din afara clasei (decat in anumite conditii asupra carora vom reveni). In cele ce urmeaza am prezentat un exemplu pentru a intelege mai bine aceasta metoda:

```
class Finalize
{
      public static void main(String args[])
      {
            int count;
            FinalizeDemo ob = new FinalizeDemo(0);
            /* Generam un numar mare de obiecte, care apoi vor fi
            distruse de GC la un moment dat*/
            for(count=1; count < 100000; count++)
                  ob.generator(count);
      }
}
class FinalizeDemo
{
      int x;
      FinalizeDemo (int i)
      {
            x = i;
      }
```

```
        //apelat de GC
        protected void finalize()
        {
              System.out.println("Finalizarea lui " + x);
        }
        // genereaza un obiect care este imediat distrus
        void generator(int i)
        {
              FinalizeDemo o = new FinalizeDemo(i);
        }
  }
```

In acest exemplu se creează 10000 de obiecte de tip *FinalizeDemo*, prin intermediul metodei *generator()*. Atunci cand se paraseste domeniul de definitie al acestei metode (in cadrul for-ului din metoda *main()*) practic obiectul *o* nu mai este folosit. Acest lucru va fi remarcat de GC atunci cand acesta va face colectarea obiectelor care nu mai sunt folosite. Cand *o* este distrus se va apela automat metoda *finalize()*. Aceasta metoda, pentru fiecare obiect de tip *FinalizeDemo* afiseaza mesajul „Finalizarea lui ” si numarul care a fost transmis (in cadrul for-ului) ca parametru constructorului, si anume valoarea lui *x*. Acest *x*, poate fi vazut ca un identificator de obiect in cadrul celor 10000 care vor fi create si la un moment dat distruse.

Cuvantul cheie this

Atunci cand apelam o metoda ce apartine unui obiect din interiorul obiectului, implicit se considera ca referinta cu ajutorul caruia aceasta metoda s-a apelat este obiectul ce o invoca. Aceasta referinta este *this*. ***this*** reprezinta instanta obiectului curent. Pentru a intelege mai bine this avem exemplul de mai jos, in care calculam puterea unui numar:

```
class Power
{
      double b;
      int e;
      double val;
      Power(double base, int exp)
      {
            this.b = base;
            this.e = exp;
            this.val = 1;
            if(exp==0) return;
```

```
            for( ; exp>0; exp--)
                  this.val = this.val * base;
      }
      double get_power()
      {
            return this.val;
      }
}
class DemoPower
{
      public static void main(String args[])
      {
            Power x = new Power(3, 2);
            System.out.println(x.b + " la " + x.e +
            " este " + x.get_power());
      }
}
```

In cazul acesta, in constructor instructiunea `this.b = base;` se refera valoarea *b* declarata ca membru al clasei Power. Atunci cand cream un nou obiect se va apela constructorul Power, iar in cadrul constructorului initializam membrii noului obiect adica *this*.

Pachete

In programare este folositor sa grupam organizat, modulele scrise. In Java, acest lucru este posibil datorita existentei pachetelor. Pachetul ofera un mecanism de organizare a bucatilor dintr-un program ca un tot unitar si totodata ofera un mod de grupa colectiile de clase. Mai mult clasele definite intr-un pachet pot fi ascunse in acel pachet si astfel nu vor fi accesibile unui alt pachet, sau in afara pachetului in care sunt definite.

In general cand denumim o clasa, ii alocam un spatiu de nume: *namespace*. Acesta este ca o regiune in cadrul careia doua clase nu pot avea acelasi nume. De ce este necesar acest *namespace*?

In cadrul programelor simple, ca cele prezentate mai sus numele claselor difera. In programe mari, insa putem avea acelasi nume de clasa pentru doua scopuri diferite: spre exemplu un program care ar efectua diverse calcule pentru un motor ar putea avea doua clase *Power* , una pentru calculul puterii unui motor, iar alta pentru calculul matematic de ridicare la putere. In acest caz separam codul in doua module/namespace-uri unul denumit eventual *engine.characteristics* iar altul denumit *math*. In cele ce urmeaza vom analiza cum se definesc aceste nume.

Definirea pachetelor

Toate clasele apartin, in Java, unui pachet. Atunci cand nu apare instructiunea *package* pentru a specifica in mod clar apartenenta, se foloseste de fapt pachetul *default* sau *global*.

Pentru a crea un nou pachet, comanda **package** va apare la inceputul fisierului sursa:

```
Package pachet;
```

Aici pachet este numele unui pachet. De exemplu, pentru crearea unui pachet **Project1**:

```
Package Project1;
```

Sistemul de tratare a pachetelor in Java este in felul urmator: fiecare pachet este stocat in folderul cu numele sau. De exemplu fisierele **.class** declarate ca fiind din proiectul Projectul1 trebuie salvate in folderul **Project1**.

Se poate crea o ierarhie de pachete. Pentru a face acest lucru fiecare pachet poate fi inclus ca un arbore:

```
package pack1.pack2.pack3;
```

Desigur folderul *pack3* va sta in folderul *pack2* iar *pack2* va fi subfolder al lui *pack1*.

Pachete si CLASSPATH

Problema care intervine este, daca vom avea o ierarhie de clase si multe foldere, de unde stie Java sa ia aceste clase?

Se poate specifica prin intermediul CLASSPATH, care este o variabila de mediu, o serie de cai, unde o cale va fi de forma (in Windows) C:\Pachete\Project1. Fiind o variabila de mediu, aceste cai vor fi separate prin „;”.

Mai jos avem un exemplu care sa lamureasca folosirea pachetelor:

```
package BookPack;
class Book
{
      private String title;
      private String author;
```

```
	Book(String t, String a)
	{
		title = t;
		author = a;
	}
	void show()
	{
		System.out.println(title);
		System.out.println(author);
	}
}
class BookDemo
{
	public static void main(String args[])
	{

		Book book0 = new Book("Cel mai iubit dintre pamanteni","Marin
Preda");
		Book book1 = new Book("Batranul si marea","Ernest Hemingway");
		Book book2 = new Book("Dune","Frank Hebert");

		book0.show();
		book1.show();
		book2.show();
	}
}
```

In acest exemplu, ambele clase se afla in pachetul *BookPack*. In urma compilarii folosind comanda

```
javac BookDemo.java
```

vom obtine doua fisiere: Book.class, BookDemo.class . Acestea trebuie sa fie in folderul BookPack neaparat. Din folderul parinte lui BookPack vom lansa interpretatorul:

```
java BookPack.BookDemo
```

Cum folosim alte pachete?

Importarea pachetelor

Folosind cuvantul cheie **import** putem „aduce” in codul curent declaratia tuturor claselor din pachetele pe care le „importam”. Intr-un fel acest *import* functioneaza ca si directiva *include* din C++.

Acesta este declaratia generala:

```
import pkg.classname;
```

unde, pkg este numele pachetului importat, iar classname este evident numele clasei ce va fi importata. Daca vrem sa importam toate clasele, sau tot continutul unui pachet se va folosi „*” .

```
import pachet.MyClass;
```

```
import pachet.*;
```

Java contine o serie de pachete ce servesc la definirea claselor pe care le vom studia in continuare. Iata cateva:

java.lang	Contine un numar consistent de clase cu scop general
java.util	Contine clasele care au diverse functionalitati utile
java.io	Contine clasele care se ocupa cu operatiunile input/output
java.net	Contine clasele care se ocupa de lucru in retea
java.applet	Contine clasele care ajuta la lucru cu applet-uri
java.awt	Contine clasele pentru lucru cu Abstract Window Toolkit