Capitolul 3

Supraincarcarea metodelor

In Java, se pot gasi doua sau mai multe metode in cadrul aceleiasi clase, care sa aiba acelasi nume, atata timp cat parametrii lor sunt diferiti. In acest caz se spune ca metoda este supraincarcata, iar procedeul se numeste de supraincarcarea metodelor. Este unul dintre modurile prin care Java suporta polimorfismul.

Ce inseamna parametrii diferiti? In acest procedeu exista o restrictie: parametrii trebuie sa fie diferiti: atat tipul de data al parametrilor cat si numarul lor poate sa difere. Daca tipul de data returnat de functie si numai acesta este cel care difera, compilatorul nu poate alege care metoda trebuie apelata. Tipurile de data returnate nu ofera informatie suficienta pentru a face diferenta. In cele ce urmeaza vom exemplifica supraincarcarea metodelor.

```
class Supraincarcare
{
	void functieSupraincarcata()
	{
		System.out.println("Fara parametrii");
	}
	void functieSupraincarcata(int unparam)
	{
		System.out.println("Un parametru: " + unparam);
	}
	int functieSupraincarcata(int a, double b)
	{
		System.out.println("Doi parametrii: " + a + " " + b);
		return a + (int)b;
	}
}
public class DemoSupraincarcare
{
	public static void main(String args[])
		{
			Supraincarcare obj = new Supraincarcare();
			//Apelul functiei fara nici un parametru
			obj.functieSupraincarcata();
			//Apelul functiei cu un parametru de tip int
			obj.functieSupraincarcata(5);
			//Apelul functiei cu doi parametrii
			int i  = obj.functieSupraincarcata(5,5.7);
			System.out.println(i);

		}
}
```

In exemplul de mai sus in clasa *Supraincarcare* avem functia *functieSupraincarcata* redefinita de doua ori dupa prima definitie fara nici un parametru:

```
int functieSupraincarcata(int a, double b)
void functieSupraincarcata(int unparam)
```

Dupa cum se vede una din functii nu intoarce nimic (void) pe cand cealalta returneaza o valoare de tip int care va fi influentata de parametrii transmisi la apelul functiei:

```
i  = obj.functieSupraincarcata(5,5.7);
```

Atunci cand se efectueaza acest apel, Java va chema acea functie care are ca prim parametru de tip int, iar al doilea de tip double sau transformabil catre double. Astfel si un apel de tipul acesta este valabil:

```
i  = obj.functieSupraincarcata(5,6);
```

Dupa cum am precizat, supraincarcarea nu consta doar in schimbarea tipului de data returnat de functie. Mai jos este un exemplu in care am definit doua functii cu acelasi parametru, dar tip de data returnat diferit:

```
class Supraincarcare
{

      void functieSupraincarcata(int b)
      {
            System.out.println("Un parametru: " + b);
      }
      int functieSupraincarcata(int a)
      {
            System.out.println("parametrul int: " + a);
            return a*a;
      }
}
public class DemoNoSupraincarcare
{
      public static void main(String args[])
            {
                  Supraincarcare obj = new Supraincarcare();
                  //Apelul functiei cu un parametru de tip int
                  obj.functieSupraincarcata(5);
                  //Apelul functiei tot cu un parametru
                  int i  = obj.functieSupraincarcata(7);
                  System.out.println(i);

            }
}
```

In exemplul acesta vom avea urmatoarele erori de compilare:

```
DemoNoSupraincarcare.java:8: functieSupraincarcata(int) is
already defined in Supraincarcare
         int functieSupraincarcata(int a)
             ^
DemoNoSupraincarcare.java:22: incompatible types
found   : void
required: int
                        int i  = obj.functieSupraincarcata(7);
```

Functia *functieSupraincarcata* cu parametru de tip *int* a mai fost definita, desi cu un tip de data returnat altul decat *int*, si anume *void* dupa cum ne dam seama din eroarea a doua.

De ce avem nevoie de supraincarcarea unor functii?

Acest mecanism este denumit si paradigma „o interfata, metode multiple", iar in limbajele care nu suporta acest conceput, fiecare metoda trebuie sa aiba nume diferit. Sa consideram functia *abs()* care returneaza valoarea absoluta a unui numar. Numerele pot fi reprezentate in multe feluri, astfel ca in C spre exemplu, aceasta functie ar purta denumirea de *labs()* in cazul in care returneaza un long, sau *fabs()* in cazul in care returneaza un float. Problema este ca pentru un om, devine destul de complicat sa retina cateva nume diferite pentru aceeasi functie, daca vorbim deja de cateva sute de functii pe care le foloseste in mod uzual. Evident functia *abs()* este un singur exemplu pentru a raspunde la intrebare.

Supraincarcarea constructorilor

Constructorul este o metoda, astfel ca suporta acelasi mecanism de supraincarcare descris mai sus. In cele ce urmeaza am ales un exemplu clasic de supraincarcare a constructorilor:

```
class Numbers
{
	int m_int;
	short m_short;
	double m_double;
	float m_float;

	Numbers()
	{
		m_int = 0;
		m_double =0;
		m_short = 0;
		m_float =0;
	}
	Numbers(double val)
	{
		m_double = val;
		m_int = 0;
		m_short = 0;
		m_float =0;

	}
	Numbers(short val)
	{
		m_short = val;
		m_int = 0;
		m_double =0;
		m_float =0;

	}
```

```
	Numbers(float val)
	{
		m_float = val;
		m_int = 0;
		m_double =0;
		m_short = 0;

	}
	Numbers(int ival, double dval)
	{
		m_int = ival;
		m_double = dval;
		m_short = 0;
		m_float =0;

	}

	Numbers(int ival, float fval)
	{
		m_float = fval;
		m_int = ival;
		m_double =0;
		m_short = 0;

	}
	public String toString()
	{
		return "int = " + m_int + " short = " + m_short
		+ " double = " + m_double + " float = " +m_float;
	}
}
public class OverloadingDemo
{
	public static void main(String[] args)
	{
		Numbers number1 = new Numbers();
		Numbers number2 = new Numbers(1);
		Numbers number3 = new Numbers(1.1);
		Numbers number4 = new Numbers(2,3);
		Numbers number5 = new Numbers(2,4.4);
		System.out.println(number1);
		System.out.println(number2);
		System.out.println(number3);
		System.out.println(number4);
		System.out.println(number5);

	}

}
```

Deocamdata sa ignoram functia *toString()* care ajuta la o afisare rapida a membrilor clasei *Numbers*.

In exemplul de mai sus avem sase constructori, toti cu parametrii de tipuri diferite. Acestia in interiorul lor, initializeaza membrii, fie cu zero, fie cu valoarea corespunzatoare a parametrilor:

```
Numbers(int ival, float fval)
	{
		m_float = fval;
		m_int = ival;
		m_double =0;
		m_short = 0;


	}
```

De exemplu, daca am ales sa transmit un parametru *int* si unul *float*, *m_int* si *m_float* sunt initializati cu acele valori, iar restul cu zero. Evident logica poate fi schimbata si in cazul de fata toate variabilele intializate cu unul din parametrii constructorului. Ce este interesant, vom afla la apelul constructorilor in interiorul clasei *OverloadingDemo*. Constructorul *implicit* este cel fara parametrii si in cadrul lui toate variabilele se initializeaza cu zero. In cazul apelului `Numbers number2 = new Numbers(1);` ar trebui sa se apeleze constructorul cu parametrul de tip int, si asa se si intampla. Totusi in cazul celui de al treilea apel si anume: `Numbers number3 = new Numbers(1.1);` lucrurile devin neclare. Numarul transmis ca parametru poate fi interpretat de Java ca si *double* sau ca *float*. Aceeasi ambiguitate o avem si in apelurile ulterioare, pentru ca nu stim care dintre constructori vor fi apelati, deoarece parametrii pot fi interpretati fie ca *double*, fie ca *float* si asa mai departe. Aceasta este ceea ce interpretatorul java va „deduce”:

```
int = 0 short = 0 double = 0.0 float = 0.0
int = 1 short = 0 double = 0.0 float = 0.0
int = 0 short = 0 double = 1.1 float = 0.0
int = 2 short = 0 double = 0.0 float = 3.0
int = 2 short = 0 double = 4.4 float = 0.0
```

Pentru a controla bine aceste date avem doua cai: *cast specificat* de tipuri sau *cast* al valorilor.

Sa analizam prima metoda pentru instructiunea `Numbers number3 = new Numbers(1.1);` in care lucram fie cu float fie cu double. Pentru a interpreta numarul 1.1 ca *double* vom pune caracterul *d* la sfarsitul numarului: `Numbers number3 = new Numbers(1.1d);`.Asemenea, daca vrem sa formatam numarul ca *float* vom pune caracterul *f* la sfarsit:

```
Numbers number3 = new Numbers(1.1f);
```

A doua metoda pentru a constrange tipul de data al parametrului transmis este de a efectua *cast* explicit catre un anume tip de data:

```
Numbers number3 = new Numbers((short)1);
```

Astfel ceea ce vom obtine se schimba, pentru ca apelam un alt constructor:

```
int = 0 short = 0 double = 0.0 float = 0.0
int = 0 short = 1 double = 0.0 float = 0.0
int = 0 short = 0 double = 1.1 float = 0.0
int = 2 short = 0 double = 0.0 float = 3.0
int = 2 short = 0 double = 4.4 float = 0.0
```

Recursivitate

O metoda se poate apela pe sine (din interiorul ei), iar acest concept se numeste recursivitate.

In general recursivitatea este folosita atunci cand definim o expresie, sau termeni care sunt circulari prin definitie. Unul din exemplele clasice de recursivitate este factorialul, si anume produsul numerelor naturale din 1..*n*. Mai jos este codul care realizeaza calculul factorialului:

```
class Factorial
{
	int factR(int n)
	{
		int result;
		if(n==1) return 1;
			result = factR(n-1) * n;
		return result;
	}
}
class FactorialDemo
{
	public static void main(String args[])
	{
		Factorial f = new Factorial();
		System.out.println("Factorial de 5 este " + f.factR(5));
	}
}
```

In cadrul clasei Factorial este definita functia *factR* care returneaza o valoare de tip *int*. In corpul functiei are loc o conditie

```
if(n==1) return 1;
```

si anume conditia de terminare (a apelurilor recursive). Urmeaza dupa aceasta o conditie, o instructiune care duce la apelul functiei *factR*:

```
result = factR(n-1) * n;
```

Mai intai se va apela functia aceasta si iar se va executa conditia de mai sus, pana cand n devine 1. In acea clipa are loc primul *return* din functie si anume cu valoarea 1. Se revine in functia apelata penultima data inainte sa ajungem la *n* egal cu 1 si se evalueaza result ca fiind 1*2, si se returneaza valoarea 2. Mai departe se reevalueaza valoarea lui result ca fiind 2*3 si asa mai departe.

Atunci cand o metoda se apeleaza pe sine, alti parametrii locali si variabile sunt alocate pe stiva, metoda se executa cu acesti noi parametrii. Atunci cand se revine din apel, variabilele „vechi" si parametrii initiali sunt refacuti, si instructiunea se reia de acolo de unde inital apelasem functia. Dupa cum intuiti, in cazul mai multori apeluri, stiva se va „incarca" cu metode aditionale, ceea ce va duce la o eroare cauzata de lipsa memoriei: „stack overflow".

De ce o abordare recursiva?

Raspunsul este simplu, datorita faptului ca unii algoritmi (de exemplu QuickSort) pot fii implementati mai usor in mod recursiv decat iterativ. Vom reveni asupra acestui subiect, cu exemple diverse, in capitolele urmatoare.

Parametrii variabili

Un astfel de parametru este specificat prin trei puncte (...). De exemplu, iata o metoda ce preia un numar variabil de parametrii:

```
static void TestParam(int ... p)
{
      System.out.println("Numarul argumentelor: " + p.length);
      for(int i=0; i < p.length; i++)
            System.out.println(" argumentul " + i + ": " + p[i]);
      System.out.println();
}
```

Aceasta declaratie semnifica faptul ca metoda *TestParam* poate fi apelata cu oricati parametrii atata timp cat acestia sunt de tip int:

corect: `TestParam(1,2,3);`

incorect: `TestParam(1,2.5,3);`

In felul acesta variabila *p* va deveni un sir de date de tip *int* si va functiona ca atare.

Parametrii variabili se pot utiliza impreuna cu parametrii obisnuiti iar ca exemplu avem clasa de mai jos:

```
class TestParam
{
      static void TestParam(String amesage, double adouble,int ... p)
      {
            System.out.println("Numarul argumentelor: " + p.length);
            for(int i=0; i < p.length; i++)
                  System.out.println(" argumentul " + i + ": " + p[i]);
            System.out.println();
      }

      public static void main(String args[])
      {
            TestParam("Mesaj1",2.3,3,4);
      }
}
```

Acest exemplu este corect, deoarece parametrii variabili se declara la sfarsitul listei de parametri. Daca insa, inversam ordinea in care apar parametrii in lista ca mai jos:

```
static void TestParam(String amesage, int ... p, double adouble)
```

vom avea erori de compilare si anume:

```
Test.java:13: TestParam(int...) in TestParam cannot be applied to
(int,double,in
t)
                  TestParam(1,2.5,3);
                  ^
1 error
```

```
I:\Java\Curs3\Cod_curs3>javac Test.java
Test.java:3: ')' expected
        static void TestParam(String amesage, int ... p, double
adouble)
                                                          ^
Test.java:3: ';' expected
        static void TestParam(String amesage, int ... p, double
adouble)

^
2 errors
```

De asemenea nu este permis sa avem mai multi parametrii variabili in aceeasi lista.

Mostenire

Bazele mostenirii

Ca orice limbaj de programare orientat pe obiecte, Java implementeaza un mecanism de relationare al claselor si anume ca o clasa poate contine o alta clasa in declararea ei. Acesta se face prin cuvantul cheie **extends**. Aceasta inseamna ca o subclasa extinde functionalitatile unei superclase.

Pentru a intelege mai bine, sa luam un exemplu clasic si anume al unei superclase *Forma2D* ce va fi extinsa de cel putin doua subclase *Triunghi*, *Dreptunghi*.

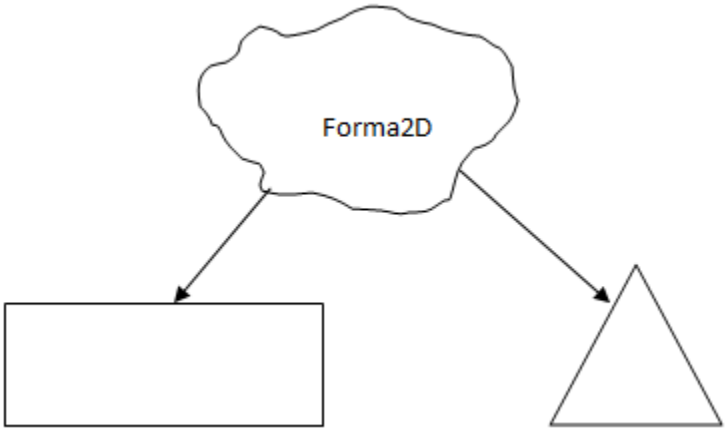

Figura 3.1 Extinderea clasei Forma2D

In ce consta extinderea acestei clase? In primul rand presupunem ca *Forma2D* contine doi membrii de tip *double* si anume *latime* si *inaltime* pe care ii vom folosi la calculul ariei formei.

```
class Forma2D
{
	double inaltime;
	double latime;
	void AfisezDimensiunile()
	{
		System.out.println("inaltimea este " +
		inaltime + " latimea este " + latime);
	}
}
```

Deocamdata avem o clasa, insa nici o posibilitate de a calcula aria unei forme din moment ce nu stim despre ce fel de forma este vorba. Ca atare, clasa *Forma2D* nu va contine nici o functie de tip *GetArea()* sau *CalcArea().*

In continuare vom implementa o clasa *Triunghi* ce mosteneste clasa *Forma2D*

```
class Triunghi extends Forma2D
{
	String tip_triunghi;
	double CalcArea()
	{
		return inaltime* latime/ 2;
	}
	void AfisezTip()
	{
		System.out.println("Triunghiul este " + tip_triunghi);

	}
}
```

Aceasta clasa va mosteni de la clasa *Forma2D* atat cei doi membrii *inaltime* si *latime* cat si metoda *AfisezDimensiunile()*. Ca atare putem folosi membrii in interiorul unei metode cum ar fi *CalcArea()*.
In continuare sa vedem cum pot fi folosite metodele – cele mostenite – si cele proprii:

```
class DemoMostenire{
	public static void main(String args[])
	{
		Triunghi t1 = new Triunghi ();
		Triunghi t2 = new Triunghi ();
		t1.latime = 8.0;
		t1.inaltime = 4.0;
		t1.tip_triunghi = "dreptunghic";
		t2.latime = 4.0;
		t2.inaltime = 6.0;
		t2.tip_triunghi = "isoscel";
```

```
            System.out.println("Informatiile despre t1: ");
            t1.AfisezTip();
            t1.AfisezDimensiunile();
            System.out.println("Aria " + t1.CalcArea());
            System.out.println();
            System.out.println("Informatiile despre t2: ");
            t2.AfisezTip();
            t2.AfisezDimensiunile();
            System.out.println("Aria " + t2.CalcArea());
      }
}
```

In functia *main* exista doua obiecte de tip *Triunghi.* Dupa cum se poate observa aceste obiecte contin, sau au acces la membrii *inaltime* si *latime* dar si la metoda *AfisezDimensiunile()*. Aceasta pentru ca clasa *Triunghi* mosteneste clasa *Forma2D* si implicit membrii ei.

Pe langa metodele si variabilele mentionate *t1* si *t2* au acces la variabila `tip_triunghi` de tip String, dar si la `AfisezTip()`si `CalcArea()`. Inainte nu puteam implementa o metoda de calcul al suprafetei pentru ca nu cunosteam tipul de forma (triunghi, cerc etc). In clipa aceasta, putem spune ca clasa *Forma2D* prinde contur prin extinderea ei in clasa *Triunghi*. Vom vedea in cele ce urmeaza ca expresia „prinde contur" poate avea mai multe intelesuri cand vom intra in detaliile polimorfismului in capitolele urmatoare.

Deocamdata spunem ca metodele *AfisezTip* si *CalAreasi* variabila *tip_triunghi* sunt membrii clasei Triunghi si numai ai clasei *Triunghi*, iar *inaltime*, *latime* si *AfisezDimensiunile()* sunt membrii clasei *Forma2D* si, datorita faptului ca *Triunghi* mosteneste *Forma2D*, sunt si membrii lui *Triunghi*.

Pentru a duce lucrurile mai departe si a exemplifica pluralismul mecanismului de mostenire vom mosteni din aceeasi clasa *Forma2D* si o clasa *Dreptunghi* ce seamana cu *Triunghi* dar are alta implementare.

```
class Dreptunghi extends Forma2D
{
      double CalcArea()
      {
            return inaltime* latime;
      }
      void AfisezTipDreptunghi()
      {
            if (latime == inaltime)
                  System.out.println("Dreptunghiul este patrat");
            else
                  System.out.println("Dreptunghiul este oarecare");
      }
}
class DemoMostenire{
      public static void main(String args[])
      {
            Dreptunghi d1 = new Dreptunghi ();
            Dreptunghi d2 = new Dreptunghi ();
```

```
            d1.latime = 4.0;
            d1.inaltime = 4.0;
            d2.latime = 4.0;
            d2.inaltime = 6.0;
            System.out.println("Informatiile despre d1: ");
            d1. AfisezTipDreptunghi ();
            d1.AfisezDimensiunile();
            System.out.println("Aria " + d1.CalcArea());
            System.out.println();
            System.out.println("Informatiile despre d2: ");
            d2. AfisezTipDreptunghi ();
            d2.AfisezDimensiunile();
            System.out.println("Aria " + d2.CalcArea());
      }
}
```

Dupa cum se observa clasa Dreptunghi nu mai are nevoie de un membru de gen `tip_triunghi` pentru ca putem sa calculam foarte simplu cu ce fel de dreptunghi lucram si acest lucru il facem in functia `AfisezTipDreptunghi()` a carei implementare este diferita fata de cea in *Triunghi* a functiei `AfisezTip()`.

Pentru a intelege mai bine mostenirea avem reprezentarile de mai jos:

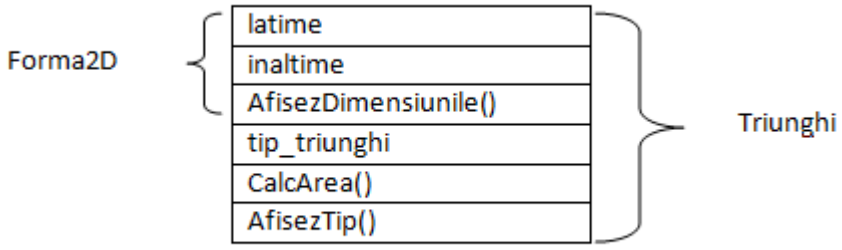

Figura 3.2 Clasa Triunghi mosteneste clasa Forma2D

Figura 3.3 Clasa Dreptunghi mosteneste clasa Forma2D

In general mostenirea se va declara in urmatorul fel:

```
class nume_subclasa extends nume_superclasa
{
      //corpul clasei
}
```

Java nu suporta mostenirea din mai multe superclase, spre deosebire de C++. Se poate totusi crea o ierarhie de mosteniri in care o subclasa devine superclasa pentru alta clasa. Desigur, o clasa nu poate fi propria superclasa.

Specificatori de acces

In Java exista la nivel de membrii ai clasei, trei tipuri de specificatori de acces: **public**, **private** si **protected**.

private inseamna ca acel membru nu poate fi accesat decat in interiorul clasei, de catre metodele din interiorul clasei.

protected inseamna ca nu poate fi accesat decat fie in interiorul clasei, de catre metodele clasei, fie din interiorul claselor ce mostenesc clasa de baza (in care membrii au fost declarati). In cadrul claselor din acelasi pachet general membrii declarati astfel sunt vizibili in aceste clase.

public inseamna ca membrii pot fi accesati si din afara clasei.

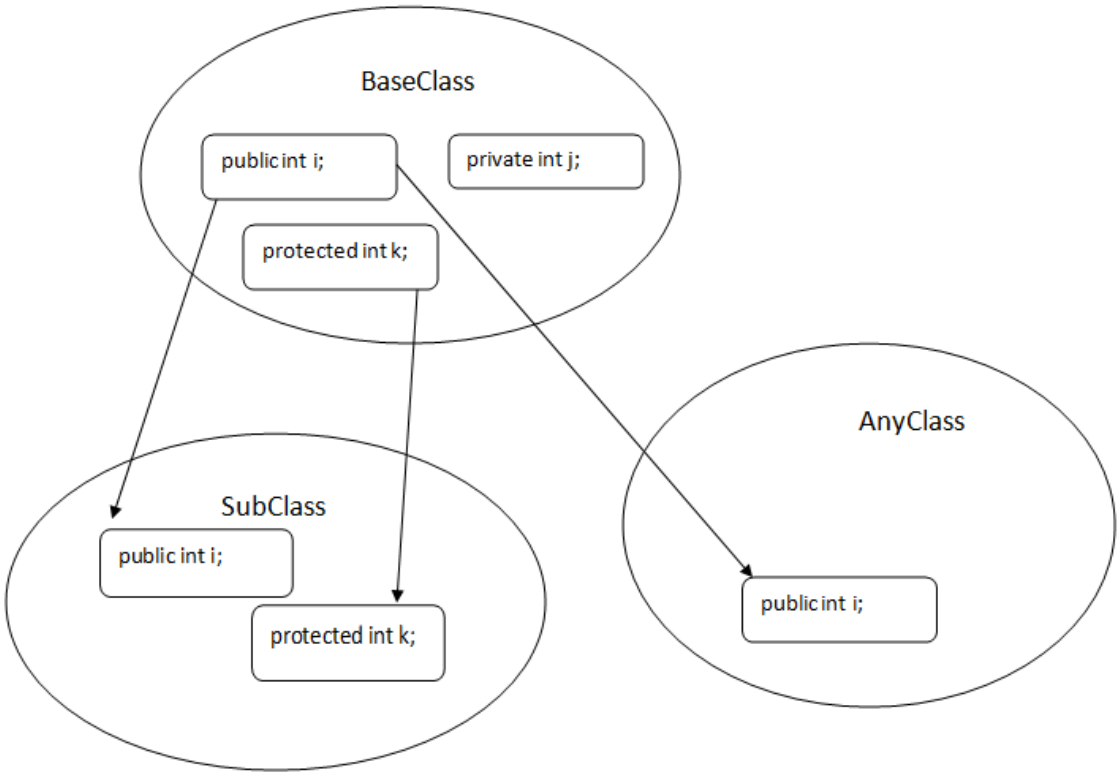

Figura 3.4 Specificatorii de acces

Mai jos avem implementarea claselor din aceasta diagrama

```
class BaseClass
{
	public int i;
	private int j;
	protected int k;
	void PrintMyVariables()
	{
		//membrii proprii
		System.out.println(i);
		System.out.println(j);
		System.out.println(k);
	}
}
class SubClass extends BaseClass
{
	double d;
	void PrintVariables()
	{
		//membrul propriu
		System.out.println(d);
		//membrii lui BaseClass
		System.out.println(i);
		System.out.println(k);
	}
}

class AnyClass
{
	float f;
	//Atentie in cadrul acestei clase
	//nu am acces direct asupra membrilor
	//celorlalte clase pentru ca nu exista
	//relatie de mostenire, ci prin instantiere
	//vom accesa membrii claselor
	void PrintVariablesOfBaseClass(BaseClass obj)
	{
		System.out.println(obj.i);
		//Acest apel ar duce la eroare
		//System.out.println(obj.j);
	}
}
class DemoAcces
{
	public static void main(String args[])
	{
		BaseClass objB = new BaseClass();
		objB.i = 20;
```

```
            objB.PrintMyVariables();
            SubClass objS = new SubClass();
            objS.d = 3.0;
            objS.i= 2;
            objS.k = 6;
            objS.PrintVariables();
            AnyClass objA = new AnyClass();
            objA.PrintVariablesOfBaseClass(objB);
      }
}
```

Membrii *i* si *k* ai clasei *BaseClass* vor fi vizibili atat in *SubClass* cat si in *AnyClass* dar din motive diferite. Clasa *SubClass* este derivata din *BaseClass,* automat mosteneste si membrii *public* si *protected* ai sai. De aceea putem face referire la membrii acestei clase fara a avea probleme. Problemele apar daca incercam sa facem referire la membrul *j* care este declarat *privat* in clasa de baza.

Pe de alta parte in clasa *AnyClass*, ce nu mosteneste nici o clasa, va trebui sa folosim membrii clasei de baza (pentru exemplificare) prin intermediul unui obiect de tip *BaseClass*. Acest obiect poate fi instantiat chiar in cadrul unei metode din clasa *AnyClass* sau poate fi transmis ca parametru unei functii din aceasta clasa. Evident al doilea caz l-am implementat si anume functia:

```
void PrintVariablesOfBaseClass(BaseClass obj)
```

Un alt specificator de acces ar fi cel implicit, si anume membrii declarati fara specificatori de acces isi extind domeniul de folosire in cadrul pachetului, adica sunt vizibili in orice clasa din pachetul clasei de baza.

Daca totusi vreau sa accesez un membru private?

Metode de get, set

In cazul in care se doreste expunerea „in exterior" a unei variabile declarata privata exista posibilitatea implementarii unor functii pentru aceasta. Acestea se numesc getter si setter si sunt folosite la returnarea valorii variabilei respectiv setarea valorii.

Mai mult, in cadrul acestor functii se pot verifica anumite conditii pentru evitarea aparitiei unor erori de logica.

```
class ExampleGetSet
{
      private String mystr;

      private double a;

      //initializam pe constructor cu string vid
      public ExampleGetSet()
      {
            mystr = "";
      }
```

```
	//functie de get cu acces public
	public String GetString()
	{
		return mystr;
	}
	//functie de set cu acces public

	public void SetString(String str)
	{
		if (str !=null)
			mystr = str;
		else
			mystr = "";
	}
	//functie de get cu acces public
	public double GetVal()
	{
		return a;
	}
	//functie de set cu acces public
	public void SetVal(Double val)
	{
		if (val !=Double.NaN)
			a = val;
		else
			a = 0;
	}

}
class DemoExampleGetSet
{
	public static void main(String args[])
	{
		ExampleGetSet obj = new ExampleGetSet();
		obj.SetString("Orice fel de sir");
		obj.SetVal(3.4);
		System.out.println(obj.GetVal());
		System.out.println(obj.GetString());
	}
}
```

In exemplul de mai sus metodele *SetVal* si *SetString* vor seta membrii privati ai clasei doar in cazul in care se „trece” de anumite conditii si anume ca String-ul transmis ca parametru sa fie diferit de *null* sau ca valoarea double transmisa ca parametru sa fie valida.

Asemenea si in cazul *GetString* sau *GetVal* se pot face unele validari, insa in general cand returnam din interior spre exterior, valorile ar trebui sa fie valide.

Nu exista regula de validare generala dar o functie *Get* ar trebui sa returneze un tip de data acelasi cu variabila pe care o expune iar *Set* are tipul *void,* iar ca parametru o variabila de tip de data acelasi cu membrul ce va fi modificat.

Nu exista reguli pentru cand o variabila trebuie declarata privata, dar sunt doua principii. Daca o variabila este folosita doar de metodele din interiorul clasei atunci ea trebuie declarata *private*. Daca valorile unei variabile trebuie sa fie intr-un interval atunci ea trebuie facuta *private*, altfel riscam ca apelata in exteriorul clasei sa genereze erori de logica sau alte erori.

Rolul constructorilor in mostenire

Intr-o ierarhie rezultata in urma mostenirii, este posibil ca atat superclasele cat si subclasele sa aiba constructorii lor proprii. Intrebarea este: care constructor se ocupa de instantierea obiectului subclasei? Este cel din superclasa, din subclasa, ambele? Raspunsul este urmatorul: constructorul superclasei va ajuta la instantierea portiunii de superclasa a obiectului si constructorul subclasei va instantia portiunea de subclasa. Pentru a fi mai explicit reluam exemplul cu *Forma2D* si *Triunghi*.

```
class Triunghi extends Forma2D
{
	String tip_triunghi;

	Triunghi(double a, double b, String tip)
	{
		inaltime =a;//initalizez portiunea legata de
		latime = b;//Forma2D adica superclasa
		tip_triunghi = tip; //instantiez portiunea de subclasa
	}
	double CalcArea()
	{
		return inaltime* latime/ 2;
	}
	void AfisezTip()
	{
		System.out.println("Triunghiul este " + tip_triunghi);
	}
}
class DemoMostenire
{
	public static void main(String args[])
	{
		Triunghi t1 = new Triunghi (4,8,"dreptunghic");
		Triunghi t2 = new Triunghi (4,6,"isoscel");
```

```
            System.out.println("Informatiile despre t1: ");
            t1.AfisezTip();
            t1.AfisezDimensiunile();
            System.out.println("Aria " + t1.CalcArea());
            System.out.println();
            System.out.println("Informatiile despre t2: ");
            t2.AfisezTip();
            t2.AfisezDimensiunile();
            System.out.println("Aria " + t2.CalcArea());
      }
}
```

Portiunea legata de superclasa este instatiata automat, apeland constructorul implicit al clasei *Forma2D*.

super

Pe langa cele prezentate mai sus, o subclasa poate apela constructorul superclasei prin utilizarea cuvantului cheie *super*. Utilizarea este:

```
super(lista de parametrii);
```

Pentru a vedea cum se utilizeaza acest apel vom modifica clasa de mai sus

```
class Forma2D
{
      double inaltime;
      double latime;
      public Forma2D(double a, double b)
      {
            inaltime =a;
            latime = b;
      }
      void AfisezDimensiunile()
      {
            System.out.println("inaltimea este " +
            inaltime + " latimea este " + latime);
      }
}

class Triunghi extends Forma2D
{
      String tip_triunghi;

      Triunghi(double a, double b, String tip)
      {
            super(a, b);
            tip_triunghi = tip;
      }
      double CalcArea()
      {
            return inaltime* latime/ 2;
      }
```

```
	void AfisezTip()
	{
		System.out.println("Triunghiul este " + tip_triunghi);

	}
}
```

Am definit un constructor in clasa Forma2D cu doi parametrii de tip double. In constructorul din Triunghi am inlocuit cele doua instructiuni ce initializau membrii din Forma2D cu apelul constructorului clasei parinte.

```
inaltime =a;//initalizez portiunea legata de
		latime = b;//Forma2D adica superclasa
```

Aceste doua instructiuni au fost inlocuite de `super(a, b);`

Aici clasa *Triunghi* apeleaza constructorul clasei *Forma2D* cu doi parametrii de tip double, in felul acesta nu mai initializeaza subclasa membrii superclasei.

```
class Forma2D
{
	double inaltime;
	double latime;
	public Forma2D()
	{
		inaltime =0; //in constructorul fara parametrii
		latime = 0;//initializez membrii cu zero
	}
	public Forma2D(double a, double b)
	{
		inaltime =a;
		latime = b;
	}
	void AfisezDimensiunile()
	{
		System.out.println("inaltimea este " +
		inaltime + " latimea este " + latime);
	}
}
class Triunghi extends Forma2D
{
	String tip_triunghi;

	Triunghi(String tip)
	{
		super(); //apelez Forma2D() si membrii superclasei vor fi zero
		tip_triunghi = tip;
	}

	Triunghi(double a, double b, String tip)
	{
		super(a, b); //apelez Forma2D(a,b)
		tip_triunghi = tip;
```

```
	}
	double CalcArea()
	{
		return inaltime* latime/ 2;
	}
	void AfisezTip()
	{
		System.out.println("Triunghiul este " + tip_triunghi);

	}
}
class DemoMostenire{
	public static void main(String args[])
	{
		Triunghi t1 = new Triunghi ("fara dimensiuni");
		Triunghi t2 = new Triunghi (4,6,"isoscel");

		System.out.println("Informatiile despre t1: ");
		t1.AfisezTip();
		t1.AfisezDimensiunile();
		System.out.println("Aria " + t1.CalcArea());
		System.out.println();
		System.out.println("Informatiile despre t2: ");
		t2.AfisezTip();
		t2.AfisezDimensiunile();
		System.out.println("Aria " + t2.CalcArea());
	}
}
```

In clasa Triunghi avem doi constructori dintre care unul doar cu un parametru de tip String. In acel constructor vom apela constructorul clasei Forma2D `super();` fara parametrii si anume a carui definitie este:

```
public Forma2D()
	{
		inaltime =0;
		latime = 0;
	}
```

Prin aceasta am exemplificat si supraincarcarea constructorului clasei parinte, si anume cu doi parametrii de tip double si fara parametrii, si mai mult apelarea diversilor constructori in functie de caz.

Mai mult, din subclasa se pot apela membrii (variabilele) superclasei prin intermediul *super*, in acelasi mod in care am utilizat *this*. De fapt putem spune ca *super* este *this*-ul superclasei, si anume instanta obiectului curent al parintelui.

Intrebarea vine in mod firesc: cine se executa primul, constructorul parintelui sau cel al copilului? Constructorii sunt apelati in ordinea derivarii, de la superclasa la subclasa.

Suprascrierea metodelor

Atunci cand o metoda dintr-o clasa copil are aceeasi semnatura (parametrii, nume si tip de data) ca si metoda din clasa parinte, atunci metoda din copil *suprascrie* metoda din clasa parinte. Atunci cand se apeleaza o metoda suprascrisa din subclasa, referirea se face doar la metoda din subclasa (ca si cum metoda din superclasa nu ar exista)

```
class A
{
	int i, j;
	A(int a, int b)
	{
		i = a;
		j = b;
	}
	//afisez i si j
	void show()
	{
		System.out.println("i si j: " + i + " " + j);
	}
}
class B extends A
{
	int k;
	B(int a, int b, int c)
	{
		super(a, b);
		k = c;
	}
	//afisez doar k
	void show()
	{
		System.out.println("k: " + k);
	}
}
class DemoSuprascriere
{
	public static void main(String args[])
	{
		B obj = new B(1, 2, 3);
		obj.show(); // se va apela show() din B

	}
}
```

In exemplul de mai sus evident, clasa B este copilul lui A. Metoda *show()* este suprascrisa in B, adica mai este redefinita in B, desi ea exista si in parinte si anume A. In clasa *DemoSuprascriere* cand se lucreaza cu obiectul *obj* de tip B, si se apeleaza metoda *obj.show(),* este ca si cum clasa A nu ar defini

metoda *show().* Pe de alta parte, daca comentam definitia metodei *show()* din clasa B, si efectuam apelul *obj.show(),* se vor afisa valorile lui *i* si *j* din superclasa A.

In clipa aceasta poate aparea confuzia legata de conceptul de supraincarcare. Supraincarcarea presupune ca aceeasi functie sa aiba parametrii diferiti, ceea ce in cazul de mai sus nu se intampla.

Totusi in cele ce urmeaza vom implementa o supraincarcare a metodei *show()*in B pentru a exemplifica diferenta.

```
class A
{
	int i, j;
	A(int a, int b)
	{
		i = a;
		j = b;
	}
	//afisez i si j
	void show()
	{
		System.out.println("i si j: " + i + " " + j);
	}
}
class B extends A
{
	int k;
	B(int a, int b, int c)
	{
		super(a, b);
		k = c;
	}
	void show(String str) //Metoda supraincarca show() din A
	{
		System.out.println(str);
	}
	//in clasa B nu mai am show() suprascrisa
}
class DemoSuprascriere
{
	public static void main(String args[])
	{
		B obj = new B(1, 2, 3);
		obj.show("Mesaj 1"); //apelez show() din B
		obj.show(); // se va apela show() din A

	}
}
```

Motivele pentru care se mentine acest mecanism de suprascriere sunt multiple, si este unul din elementele ce contribuie la polimorfism la momentul rularii. Polimorfismul este fundamental in OOP deoarece permite unei clase generale sa specifice metode ce vor fi aceleasi pentru toate clasele derivate din ea, in timp ce unora din clasele copii le permite sa aiba propriile implementari pentru acele metode.

final

Atunci cand nu se doreste ca o metoda sa poata fi suprascrisa intr-una din clasele copil derivate din clasa parinte, se poate specifica *final* pentru acea metoda. Acest program

```
class A
{
	int i, j;
	A(int a, int b)
	{
		i = a;
		j = b;
	}
	//afisez i si j
	final void show()
	{
		System.out.println("i si j: " + i + " " + j);
	}
}
class B extends A
{
	int k;
	B(int a, int b, int c)
	{
		super(a, b);
		k = c;
	}
	void show()
	{
		System.out.println(k);
	}
	//in clasa B nu mai am show() suprascrisa
}
class DemoSuprascriere
{
	public static void main(String args[])
	{
		B obj = new B(1, 2, 3);
		obj.show(); // se va apela show() din B

	}
```

```
}
```

Va returna urmatoarea eroare:

```
A.java:23: show() in B cannot override show() in A; overridden method is
final
        void show()
             ^
1 error
```

In cazul variabilelor cuvantul cheie final este asemanator cuvantului const din C++ si marcheaza o variabila ca fiind constanta, iar aceasta nu mai poate fi intializata cu alta valoare.

Clasa Object

Toate clasele in Java, extind direct sau indirect, clasa Object aflata in pachetul *java.lang*. Aceasta clasa defineste cateva metode importante de care trebuie sa tinem seama, pentru ca fiecare clasa pe care o scriem poate suprascrie sau folosi aceste metode.

Metoda	**Scop**
Object clone( )	Creeaza un obiect cu aceleasi proprietati ale obiectului clonat.
boolean equals(Object object)	Determina faptul ca un obiect este/ nu este egal cu altul.
void finalize( )	Apelul inainte ca obiectul sa fie distrus.
Class<? extends Object> getClass( )	Determina clasa din care obiectul face parte.
int hashCode( )	Returneaza un id specific fiecarui obiect.
void notify( )	Reia executia firului de executie aflat in asteptare.
void notifyAll( )	Reia executia firelor de executie aflate in asteptare.
String toString( )	Returneaza un sir de caractere ce descrie obiectul.
void wait( )	Suspenda firul de executie apelant.

Metodele *getClass(), notify(), notifyAll()* si wait sunt declarate *final*. Vom reveni asupra lor la momentul oportun. In cele ce urmeaza vom discuta despre celelalte metode.

clone( )

Metoda returneaza un obiect ce are membrii identici cu cel al obiectului curent. Metoda functioneaza doar daca clasa implementeaza interfata Cloneable (vom aborda subiectul legat de interfete in capitolul urmator). Metoda este declarata *protected* adica doar subclasele lui Object pot suprascrie aceasta metoda. Vom reveni cu un exemplu mai elaborat cand discutam despre interfete.

```
class AnyClass implements Cloneable
{
      int i;
      String s;
      public double d;
      public void InitVars()
      {
```

```
                i=3;
                s="Mesaj";
                d= 4.1;
        }
        protected AnyClass clone()
        {
                AnyClass newobj = new AnyClass();
                newobj.i=i;
                newobj.s = s;
                newobj.d = d;
                return newobj;
        }
}
class DemoObject extends AnyClass
{
        public static void main(String args[])
        {
                AnyClass obj1 = new AnyClass();
                obj1.InitVars();
                AnyClass obj2 = obj1.clone();
                obj2.i =1;
                System.out.println(obj1);
                System.out.println(obj2);
                System.out.println(obj1.i + obj1.s + obj1.d);
                System.out.println(obj2.i + obj2.s + obj2.d);
        }
}
```

Pentru moment vom spune ca *Cloneable* este un fel de clasa parinte pentru *AnyClass* si metoda *clone()* este mostenita si suprascrisa in *AnyClass*. Tipul de data returnat de *clone* este tot un AnyClass si obiectul returnat este unul nou instantiat, cu valorile preluate din obiectul curent. Metoda *clone()* poate fi suprascrisa in orice fel, putem returna chiar si instanta obiectului curent, insa nu acesta este scopul.

Atunci cand rulam programul vom observa doua referinte, si anume a doua obiecte diferite, chiar daca au initial aceleasi proprietati:

```
AnyClass@19821f
AnyClass@addbf1
3Mesaj4.1
1Mesaj4.1
```

Atunci cand rulam

```
System.out.println(obj1);
```

Rezultatul este unul oarecum ciudat si anume `AnyClass@19821f`.

In continuare vom lamuri aceasta „ciudatenie”

toString( )

Atunci cand apelam metoda de afisare a oricarui obiect in Java, implicit se apeleaza o metoda toString() mostenita din cadrul clasei Object. Implicit aceasta metoda returneaza un format de genul

urmator: *TipulClasei@hascode_al_clasei*. Astfel se explica de ce cand afisam direct obiectul ca mai sus obtinem acel rezultat.

Pentru a creea rezultate care sa insemne ceva ce poate fi interpretat de oameni, putem suprascrie metoda toString() pentru a afisa de exemplu membrii obiectului.

```
class AnyClass
{
	int i;
	String s;
	public double d;
	public void InitVars()
	{
		i=3;
		s="Mesaj";
		d= 4.1;
	}
	public String toString()
	{
		String str="";
		str += "i este " + i + "\n";
		str += "s este " + s + "\n";
		str += "d este " + d + "\n";
		return str;
	}
}
class DemoObject
{
	public static void main(String args[])
	{
		AnyClass obj1 = new AnyClass();
		obj1.InitVars();
		//afisam obj1 prin apelului lui toString
		System.out.println(obj1);
		//Alt mod de a afisa continutul lui obj1
		System.out.println(obj1.i + obj1.s + obj1.d);
	}
}
```

Dupa cum se poate observa apelul inlesneste afisarea obiectului obj1. Mai mult daca am avea membrii *private* metoda *toString()* expune valorile acestora.

equals( )

Operatorul „==” testeaza daca doua obiecte pointeaza catre aceeasi referinta. Pentru a vedea daca doua obiecte contin valori diferite, trebuie sa folosim *equals()*.

hashcode()

Atunci cand suprascriem metoda equals, trebuie sa suprascriem si metoda *hashcode()*. Aceasta metoda returneaza un intreg care este folosit de java pentru a diferentia doua obiecte. Este important ca doua obiecte egale conform metodei *equals()* sa aiba aceleasi hashcode-uri. De asemenea doua obiecte diferite in acelasi sens trebuie sa aiba hashcode-uri diferite.

In continuare avem un exemplu de suprascriere a celor doua metode mai sus mentionate:

```
class Circle
{
      private final int x, y, r;

      // Constructorul de baza
      public Circle(int x, int y, int r)
      {
            this.x = x; this.y = y; this.r = r;
      }

      //Constructor de copiere - alternativa la clone()
      public Circle(Circle original)
      {
            x = original.x;
              y = original.y;
              r = original.r;
      }

    // functii Get
      public int getX()
      {
            return x;
      }
      public int getY()
      {
            return y;
      }
      public int getR()
      {
            return r;
      }

    // suprascrierea equals
    @Override public boolean equals(Object o)
      {
            if (o == this) return true;                //referinte egale
            if (!(o instanceof Circle)) return false; //daca tipul de data nu
este corect
              Circle that = (Circle) o;                  //se aplica un cast
catre tipul corect
              if (this.x == that.x && this.y == that.y && this.r == that.r)
```

```
                return true;
              else
                return false;
    }


    @Override public int hashCode() {
        int result = 17;
        result = 37*result + x;
        result = 37*result + y;
        result = 37*result + r;
        return result;
    }
}
class CircleDemo
{
      public static void main(String args[])
      {
            Circle c1 = new Circle(1,1,4);
            System.out.println(c1.hashCode());
            Circle c2 = new Circle(1,2,4);
            System.out.println(c2.hashCode());

            System.out.println(c1.equals(c2));

      }
}
```

Specificatorul *@Override* intareste ideea ca metoda este suprascrisa.

In metoda equals pasii sunt tot timpul aceeasi:

1. Se verifica daca referintele sunt egale
2. Se verifica faptul ca obiectul cu care se va face comparatia sa aiba acelasi tip de data cu obiectul curent.
3. Se compara proprietatile celor doua obiecte si se returneaza true sau false.

HashCode este mai interesanta. In cadrul acestei metode este imperativa sa implementam un algoritm care sa produca un numar, care sa respecte conditiile mentionate la descrierea metodei *hasCode*. Acest algoritm, prin inmultirea cu un numar prim si insumarea membrilor obiectului asigura acest lucru.

O suma a lui *x*, *y* si *r* nu ar fi suficienta deoarece trei numere pot da aceeasi suma cu alte trei numere diferite insumate. De aceea un algoritm ar fi sa insumam cele trei numere inmultite cu numere prime pentru a diminua aceasta posibilitate: de exemplu 23*x+31*y+17*r. Cu cat algoritmul matematic este mai complex cu atat regulile *hashCode* sunt respectate in mai multe cazuri.