C8

Metode şi Moştenire

- Metode. Definiţie, semnătură, apel.
- Constructori
- Supraîncărcare.
- this
- Moştenire
- super

Obiective

Dupa parcurgerea acestui curs ar trebuie sa puteţi:

- ☐ să creaţi şi să operaţi cu metode;
- ☐ iniţilizaţi variabile de instanţă utilizând constructori
- ☐ intelege si opera cu supraîncărcarea în cazul metodelor
- ☐ inţelege şi defini ierarhii de clase pentru scrierea unei aplicatii orientate pe obiect

Metode

Operaţiile unei clase se definesc cu ajutorul metodelor. Metodele din Java sunt echivalentul procedurilor şi funcţiilor din alte limbaje de programare, cu excepţia ca ele trebuie scrie obligatoriu în interiorul definiţie de clasă (nu există conceptul de metodă globală în Java). Vizibilitatea controlează cine anume va putea apela metoda. Pentru public acesta va fi apelabilă din toată aplicaţia, pentru private numai de alte metode din clasa în care s-a definit. Dacă metoda întoarce o valoare, adică este o funcţie, ea va avea specificat unul dintre tipurile deja discutate şi o instrucţiune de salt necondiţionat `return expresie` ca va genera întoarcerea în codul apelant cu valoarea lui `expresie`. Dacă metoda nu întoarce valoare, adică este o procedură, tipul ei va fi obligatoriu `void` şi nu va conţine instrucţiune `return`. Pentru ca operaţia implementată la nivelul unei metode să aibă loc aceasta trebuie apelată. Prin apel, va se realiza rularea instrucţiunilor cuprinse în corpul metodei. Terminarea metodei se face la rularea unei instrucţiuni `return`, din corpul ei sau, în lipsa lui `return`, la atingerea acoladei de inchidera a corpului ciclului. La terminarea rulării unei metode rularea aplicaţiei va continua cu instrucţiunea imediat următoare celei de apel a metodei. Porţiune de cod din care se face apelul metodei se numeşte cod apelant, iar porţiune de cod corespunzătoare metodei apelate se numeşte cod apelat. Metoda poate avea zero sau mai mulţi parametri. Parametrii sunt valori transferate metodei în momentul apelului acesteia. Aceste valori, dacă există, vor putea fi prelucrate în corpul metodei pentru a produce rezultatele întoarse cu `return`.

Apelul metodelor şi metodele de transfer al parametrilor

Aplelul de metodă
Apelul unei metode se face se face cu ajutorul operatorului punct. Forma generală a unui apel de metodă este `refObiect.nume_met(argumente)`, în situaţia exemplului de mai sus apelul de metodă este `p1.setX(12)`.

Parametrii şi argumente
Metodele nu au nevoie de parametri. Folosirea parametrilor perminte însă o generalizare a meteodei în sensul că aceasta va putea opera cu mai multe date de intrare şi/sau în mai multe situaţii uşor distincte.

Termenul de **parametru** se foloseşte pentru o variabilă definită la nivelul metodei care primeste o valoare în momentul apelării metodei. Persistenţa şi vizibilitatea parametrilor este limitată la corpul metodei. Termenul de **argument** se foloseşte pentru o valoarea care se transferă metodei atunci când aceasta este apelată. Astfel, în exemplul prezentat `abscisa` este parametru, iar `12` este argument.

Apel prin valoarea
Exită două tipuri de parametri în Java: tipuri primitive şi referinţe la obiecte. În cazul ambelor categorii de parametri apelul se face întotdeauna prin valoare. Respectiv metoda primeşte o copie a argumentelor prin stivă şi nu va putea să modifice conţinutul iniţial al argumentelor ce se transferă în parametrii din interiorul metodei. În cazul parametrilor de tipul referinţă la obiect se poate modifica starea obiectului utilizând metodele puse la dispoziţie în acest scop, dar nu se pot modifica referinţele originale ale argumentelor de tipul referinţă la obiect în corpul metodei. Iată un exeplu:

Este imposibilă modificarea unui argument de tip primitiv prin parametrii metodei. Pentru aceasta s-a scris metoda `dubleazaX()` ce ar trebui să dubleze valorea argumentului `x` cu valoarea iniţială `5`.

Observaţi în rezultate că deşi în corpul metodei dublarea se face, la terminarea metodei şi revenirea din aceasta, valoarea dublată se pierde. Iată cum lucrează Java:

1. parametrul `a` este iniţializat cu o copie a valorii stocate în argumetul `x`, adică cu o copie a lui `5`;
2. `a` este dublat, adică devine `10`, dar `x` rămâne `5`;
3. metoda se termină, iar parametrul îşi încetează persistenţa, deci variabila `a` nu mai există.

În siuaţia în care parametrul (vezi metoda `schimbaX()`) este o referinţă la un obiect, Java lucrează astfel:

1. parametrul `x` primeşte o copie a referinţei către obiectul `p1`;
2. metoda `setX()` este aplicată asupra referinţei la obiect, obiectul `PunctOK` fiind referit în acest moment de `x` şi `p1`;

3. metoda se termină şi parametrul x încetează să mai existe.

Metoda poate modifica starea unui parametru obiect deoarece primeşte copie o referinţei la obiectul iniţial. Atât copia cât şi argumentul referă însă acelaşi obiect.

```
public class GraficaOK {
  public static void
main(String[] args) {
    double x = 5;
    PunctOK p1 = new PunctOK();
    PunctOK p2 = new
PunctOK(-1.,7.);

    System.out.println("in afara
lui dubleazaX x este: " + x);
     dubleazaX(x);
    System.out.println("in afara
lui dubleazaX x este: " + x);

    System.out.println("p1 = " +
p1);
    System.out.println("p2 = " +
p2);
    interschimba(p1, p2);
    System.out.println("p1 = " +
p1);
    System.out.println("p2 = " +
p2);

    schimbaX(p1);
    System.out.println("p1 = " +
p1);
    }

//modificarea valorii
argumentului
//NU se reflecta in codul
apelant
 public static void
dubleazaX(double a) {
    a = 2. * a;
  System.out.println("in
dubleazaX x este: " + a);
    }

//metoda de interschimbare NU
lucreaza
//ca urmare a transferului prin
valoare
 public static void
interschimba(PunctOK x, PunctOK
y) {
    PunctOK aux = x;
    x = y;
    y = aux;
 }
//lucreaza corect
 public static void schimbaX(PunctOK x) {
   x.setX(100);
 }
}
```

Rezultate:

```
in afara lui dubleazaX x este: 5.0
in dubleazaX x este: 10.0
in afara lui dubleazaX x este: 5.0
p1 = <0.0,0.0>
p2 = <-1.0,7.0>
p1 = <0.0,0.0>
p2 = <-1.0,7.0>
p1 = <100.0,0.0>
```

Constructori

Constructorii clasei `PunctOK` sunt prezentați în secvența de cod alăturată. Observați că există doi contructori ce poartă numele clasei, diferența între ei fiind parametrii. În măsura în care este uitată definirea lor, Java utilizează un constructor implicit (default constructor), fără argumente şi fără nici un fel de cod, ce permite, măcar crearea obiectelor din clasa respectivă şi inițializarea la valori implicite a variabilelor de instanță. Constructorul implicit poate fi scris şi explicit dacă inițializările implicite (toate datele numerice iau valoarea 0, `Boolean false` iar obiectele `null`) nu sunt cele dorite.

Constructorul implicit `PunctOK()` va fi apelat automat la creare unui obiect prin linia ce cod `PunctOK p1 = new PunctOK();`, iar constructorul `PunctOK(double x, double y)` va fi apelat automat la crearea unui obiect cu linia de cod `PunctOK p2 = new PunctOK(-1.,7.);`. Java identifică constructorul ce trebuie să-l apeleze pe baza numărului şi tipurilor parametrilor şi a argumentelor.

```
public class PunctOK {
//Campuri
 ...

//Constructori
  PunctOK() {
      setX(0);
      setY(0);
      distanta = 0;
  }

  PunctOK(double x, double y) {
      setX(x);
      setY(y);
      actualizareDistanta();
  }
 ...
}
```

Supraîncărcarea (overloading)

În exemplul prezentat clasa `PunctOK` are mai mulți constructori. Deşi constructorii au acelaşi nume, au număr şi/sau tipuri de parametrii distincți. Compilatorul determină care dintre constructori urmează să fie apelat prin compararea numărului şi a tipurilor de parametri din definiția constructorului cu numărul şi tipurile valorilor argumentelor folosite în apelul contructorului. Dacă o astfel de corespondență nu se poate stabili sau se pot stabili, simultan, mai mult de o singură corespondență biunivocă va genera o eroare legată de rezolvarea supraîncărcătrii (overloading resolution).

Supraîncărcarea în Java este posibilă atât la nivel de constructori cât şi la nivel de metode. Descrierea completă a unei metode se realizează pe baza numelui metodei împreună cu tipurile parametrilor acesteia. Aceste informații poarte denumire de semnătura metodei (method signature). Tipul întors de metodă nu face parte din semnătura metodei, atfel nu putem avea două metode cu acelați nume şi parametri, dar cu tip întors diferit.

Referința `this`

Obiectul curent poate fi referit în Java prin folosirea cuvântului cheie `this`. Toate metodele unei instanțe primesc ca argument implicit pe `this`. Obiectul curent este obiectul al cărei metode a fost apleată.

Exemplul următor prezintă modul de utilizare în cod a lui `this`. Dintre situațiile în care se doreşte folosirea explicită a lui `this` amintesc:

1. numele unei variabile de instanță este identic cu cel al unui parametru de metodă (vezi exemplul alăturat), în această situație accesul la variabila instanță se face prin `this.nume_var_instanta`;
2. este necesară transferarea unei referințe la obiectul curent ca argument, unei alte metode;
3. un constructor al unei clase apelează un alt constructor al aceleiaşi clase (permite evitarea reluării unor secvenţe de iniţializare).

```
public class Punct {
//variabile de instanta
  private double x;
  private double y;
  private double distanta;

...

// Metode
  public void setX(double x) {
      this.x = x;
      actualizareDistanta();
  }

  public void setY(double y) {
      this.y = y;
      actualizareDistanta();
  }
...
}
```

Moştenirea

Moştenirea permite crearea de noi clase pe baza unora existente. Noua clasă se numeşte subclasă şi se defnieşte prin extinderea unei clase deja exisntente numită superclasă. La definirea subclasei se vor specifica numai diferenţele faţă de superclasă. Cele mai generale variabile şi metode sunt plasate în superclasă iar cele mai specializate în subclasă. În situaţia în care unele metode ale superclasei nu sunt corerspunzătoare subclasei acestea pot şi suprascrise. Aplicaţia ce urmează îşi propune să creeze o bibliotecă grafică. Conceptul de bază aici este cel de cooronate pe care se va baza acela de punct. Clasa de bază va fi deci `Coordonate` iar cea derivată din aceasta este `Punct`. Noua clasă are adăugate variabila de instanţă `nume` , metoda `setNume()` şi suprascrisă metoda `toString()`. Deşi există două metode cu acelaşi nume `toString()` în superclasă şi însubclasă, Java va ştii să apeleze metoda dorită corect. Constructorii noi clase sunt `Punct()`, `Punct(double x, double y)` şi `public Punct(double x, double y, String nume)`. Fiecare subclasă poate accesa constructorul superclasei iar codul comun poate fi scris în superclasă şi apelat în subclasă prin folosirea lui `super`. Noua clasă însă nu moşteneşte constructorii clasei de bază ea beneficiind în mod direct numai de constructorul implicit. Deşi subclasa include toţi membrii superclasei, aceasta nu va putea membrii care au fost declaraţi cu `private`.

```
public class Coordonate {
    private double x;
    private double y;

    Coordonate(){
        setX(0);
        setY(0);
    }

    Coordonate(double x, double y){
        setX(x);
        setY(y);
    }

    public void setX(double x) {
        this.x = x;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public void setY(double y) {
        this.y = y;
    }

    public String toString() {
       return "(" + x + "," + y + ")";
    }
}
```

```
public class Punct extends Coordonate {
    private String nume;

    public Punct() {
        super();
        setNume("P");
    }

    public Punct(double x, double y) {
        super(x,y);
        setNume("P");
    }

    public Punct(double x, double y,
String nume) {
        this(x,y);
        setNume(nume);
    }

    public Punct(Coordonate c) {
        this(c.getX(),c.getY());
```

```
        setNume(nume);
    }

    public void setNume(String nume){
        this.nume = nume;
    }

    public String toString() {
       return nume +"(" + getX() +
"," + getY() + ")";
    }
}

public class Mostenire {
  public static void main(String[]
args) {
  Coordonate c1 = new Coordonate();
  Coordonate c2 = new
Coordonate(1,2);

  Punct p1 = new Punct();
  Punct p2 = new Punct(1,1);
  Punct p3 = new Punct(1,2,"P3");
  Punct p4 = new Punct(c1);

  System.out.println(c1);
  System.out.println(c2);
  System.out.println(p1);
  System.out.println(p2);
  System.out.println(p3);
  System.out.println(p4);

  c1 = p1;
  System.out.println(c1);
  c1 = c2;
  System.out.println(c1);

 }
}
```

Rezultate:

```
(0.0,0.0)
(1.0,2.0)
P(0.0,0.0)
P(1.0,1.0)
P3(1.0,2.0)
P(0.0,0.0)
P(0.0,0.0)
(1.0,2.0)
```

Referința `super`

Aplicația anterioară prezintă modul de utilizare a lui `super`. Asemenea lui `this`, ce avea două semnificații:

1. o referință de tip argument implicit la metodele obiectului;
2. o referință la obiectul în cauză prin care se pot apela alți construcori ai aceleiaşi clase;

şi cuvântul cheie `super` are două semnificaţii asigurând:

3. apelul unei metode din superclasă;
4. apelul constructorilor superclasei.

De exemplu, în codul următor, metoda `toString()` a clasei derivate `Punct` poate fi rescrisă utilizând pe super pentru apelul `toString()` din clasa de bază `Coordonate` astfel:

```
public String toString() {
   return nume + super.toString();
   //return nume +"(" + getX() + "," + getY() + ")";
}
```

C8- Întrebari

1. Care metode necesita return?
2. Cand folosim numele de parametru dar cel de argument de metoda?
3. Ce este apelul prin valoare?
4. Ce rol au constructorii?
5. Ce este superclasa si in ce relatie este cu subclasa?