

Programare Orientată pe Obiecte

Pachete

Dr. Petru Florin Mihancea

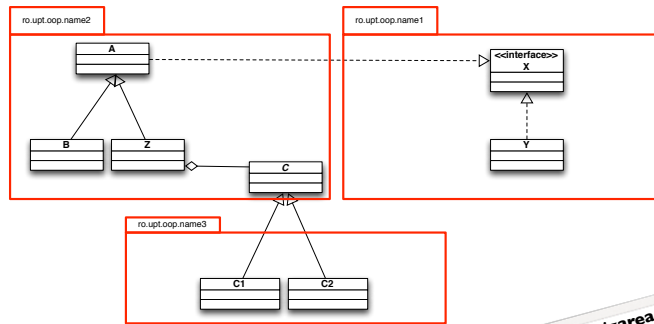
V20180924

Programare Orientată pe Obiecte

Programare Orientată pe Obiecte

Programare Orientată pe Obiecte

pachetul



suport pentru organizarea
unui program în subsisteme
(module)

Elemente de bază legate de pachete

Dr. Petru Florin Mihailescu

```
package ro.upt.oop.curs.ceasuri;

public interface ClockType {
    public void setTime(int h, int n, int s);
    public String toString();
}

abstract class AbstractClock implements ClockType {

    private int hour, minute, second;

    public AbstractClock() {
        hour = minute = second = 0;
    }

    public void setTime(int h, int m, int s) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
        second = (s >= 0) && (s < 60) ? s : 0;
    }

    public String toString() {
        return "Current time " + hour + ":" +
            minute + ":" + second;
    }
}
```

```
package ro.upt.oop.curs.ceasuri;

public class Clock extends AbstractClock {
    public String toString() {
        return "Normal clock - " + super.toString();
    }
}
```

package

nume pachet
(sucesiune de identificatori de forma
identif1.identif2. ... identifN)

```
package ro.upt.oop.curs;

...

public class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

unități de compilare
(fișiere); toate clasele/interfețele
conținute vor aparține pachetului
specificat

```
public interface ClockType {
    public void setTime(int h, int m, int s);
    public String toString();
}
```

```
hour = (h >= 0) && (h < 24) ? h : 0;
minute = (m >= 0) && (m < 60) ? m : 0;
second = (s >= 0) && (s < 60) ? s : 0;
}

public String toString() {
    return "Current time " + hour + ":" +
           minute + ":" + second;
}
}
```

```
public class Clock extends AbstractClock {
    public String toString() {
        return "Normal clock - " + super.toString();
    }
}
```

nume pachet
(sucesiune de identificatori de forma
identif1.identif2. ... identifN)

```
***
public class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

unități de compilare
(fișiere); toate clasele/interfețele
conținute vor aparține pachetului
specificat

```

package ro.upt.oop.curs.ceasuri;
public interface ClockType {
    public void setTime(int h, int n, int s);
    public String toString();
}
abstract class AbstractClock implements ClockType {

    private int hour, minute, second;

    public AbstractClock() {
        hour = minute = second = 0;
    }
    public void setTime(int h, int m, int s) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
        second = (s >= 0) && (s < 60) ? s : 0;
    }
    public String toString() {
        return "Current time " + hour + ":" +
            minute + ":" + second;
    }
}

```

Un pachet reprezintă și un spațiu distinct de nume pentru clase/interfețe; și de exemplu putem avea clase cu același nume simplu în pachete diferite

nume complet calificat

Putem referi clase/interfețe din alte pachete folosind numele complet calificat **numePachet.numeClasa**

```

package ro.upt.oop.curs;
public class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}

```

In același pachet NU avem ClockType !!!
 Ceasornicar.java:4: error: cannot find symbol
 public void regleaza(ClockType x) {
 ^
 symbol: class ClockType
 location: class Ceasornicar
 1 error

```
package ro.upt.oop.curs.ceasuri;
public interface ClockType {
    public void setTime(int h, int n, int s);
    public String toString();
}
abstract class AbstractClock implements ClockType {

    private int hour, minute, second;

    public AbstractClock() {
        hour = minute = second = 0;
    }
    public void setTime(int h, int m, int s) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
        second = (s >= 0) && (s < 60) ? s : 0;
    }
    public String toString() {
        return "Current time " + hour + ":" +
            minute + ":" + second;
    }
}
```

nume **complet** calificat

Putem referi clase/interfațe din alte
pachete folosind numele
complet calificat
numePachet.numeClasa

```
package ro.upt.oop.curs;
public class Ceasornicar {
    public void regleaza(ro.upt.oop.curs.ceasuri.ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

Un pachet reprezintă și un spațiu distinct
de nume pentru clase/interfețe; și de
exemplu putem avea clase cu același nume
simplu în pachete diferite

Dr. Petru Florin Mihailescu

Quiz

```
package ro.upt.oop.curs.ceasuri;  
public class Clock extends AbstractClock {  
    public String toString() {  
        return "Normal clock - " + super.toString();  
    }  
}
```

```
package ro.upt.oop.curs;  
class Test {  
    public Clock create() {  
        return new Clock();  
    }  
}
```

Trebuie să folosim numele complet la fiecare declarare de referință, la crearea unui object, etc.

```
package ro.upt.oop.curs;  
class Test {  
    public ro.upt.oop.curs.ceasuri.Clock create() {  
        return new ro.upt.oop.curs.ceasuri.Clock();  
    }  
}
```

În același pachet NU avem ClockType !!!
Ceasornicar.java:11: error: cannot find symbol
 public Clock create() {
 ^
symbol: class Clock
location: class Test
Ceasornicar.java:12: error: cannot find symbol
 return new Clock();
 ^
symbol: class Clock
location: class Test
2 errors


```
package ro.upt.oop.curs.ceasuri;
public interface ClockType {
    public void setTime(int h, int n, int s);
    public String toString();
}
abstract class AbstractClock implements ClockType {
    private int hour, minute, second;
    public AbstractClock() {
        hour = minute = second = 0;
    }
    public void setTime(int h, int m, int s) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
        second = (s >= 0) && (s < 60) ? s : 0;
    }
    public String toString() {
        return "Current time " + hour + ":" +
            minute + ":" + second;
    }
}
```

Alternativa

La început de unitate de compilare
(după package dacă există)
import numePachet.NumeClasă;
import numePachet.*;

În unitatea de compilare curentă:
Varianta 1 - putem referii prin nume
scurt doar clasa menționată
Varianta 2 - putem referii prin nume
toate clasele din pachetul menționat

```
package ro.upt.oop.curs;
import ro.upt.oop.curs.ceasuri.ClockType;
public class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

1

Presupunem că mai avem un pachet **tmp**
ce conține și el o clasă **Clock**

```
import ro.upt.oop.curs.ceasuri.*;
import tmp.*;

...
Clock c;//Eroare de ambiguitate
...
```

Atenție

Trebuie folosit numele complet
sau importăm explicit clasa dorită

2

Import este folosit exclusiv de compilator pentru rezolvarea de nume de clasă. **NU** are nici o legătură/similaritate cu include-ul din C

3

Ca dovadă și fără legătură cu pachetele, există (din java 1.5)

import static numePachet.NumeClasa.*;

sau

import static numePachet.NumeClasa.NumeMembruStatic;

având ca efect posibilitatea de a utiliza (în acel fișier) membrii statici
implicați fără a specifica la referirea lor și clasa din care fac parte

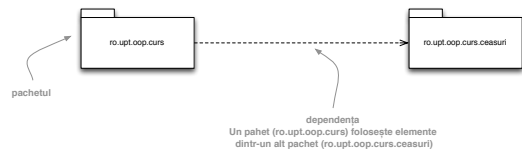
Quiz

```
public class Automat {  
    public static void main(String[] args) {  
    }  
}
```

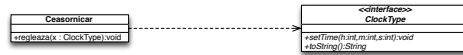
Clasa **String** e în pachetul **java.lang** și cu toate astea am referit clasa fără să folosim numele complet ori import. Oare de ce compilează?

Pachetul **java.lang** se consideră importat automat tot timpul :)

În UML



**Dependența apare și
la nivel de clase**



**Ceasornicar-ul
NU are ClockType-uri (NU e compunere),
NU este un fel de ceas (NU e moștenire),
dar folosește referințe / metode din ClockType și deci depinde de el**

Specificatori de acces pentru vizibilitatea conținutului pachetelor

Vizibilitatea **conținutului** pachetelor

public

respectiva clasă/interfață poate fi accesată **și din exteriorul pachetului** de care ea aparține

Atenție

clasa trebuie să se găsească într-un fișier (unitate de compilare) ce are același nume ca și clasa (urmat de sufixul .java); altfel e o eroare de compilare

când clasa/interfața face parte
din "interfața" pachetului

default (fără nimic - **NU** există cuvânt cheie) **package access**

respectiva clasă/interfață poate fi accesată numai în interiorul pachetului de care ea aparține

când clasa/interfața este un
detaliu de implementare a
pachetului

Exemple

```
package pachetA;  
public class A {  
    ...  
}  
  
interface B {  
    public void aMethod();  
}
```

Fișierul **trebuie** să se numească **A.java**

```
package pachetA;  
  
class Whatever implements B {  
    public void aMethod() {  
        A a;  
        B b;  
        ...  
    }  
}
```

```
package pachetB;  
  
public class Client implements pachetA.B { //Eroare  
  
    public void doSomething() {  
        pachetA.B x; //Eroare  
        pachetA.A a;  
    }  
}
```

Specificatorii de acces pentru membrii claselor în contextul pachetelor

**vom presupune că (virgulă) clasa ce
conține membrul e accesibilă din
alte pachete**

Modificatori/Specificatori de acces

private

respectivul membru al clasei (câmp/metodă) poate fi accesat **doar în interiorul clasei**

public

respectivul membru al clasei (câmp/metodă) poate fi accesat **de oriunde**

protected

respectivul membru al clasei (câmp/metodă) poate fi accesat **din interiorul clasei, din subclasele sale (pe this) sau din același pachet (pe orice obiect)**

default (fără nimic - **NU** există cuvânt cheie) package access

respectivul membru al clasei (câmp/metodă) poate fi **accesat doar din interiorul aceluiași pachet (de oriunde din interiorul pachetului)**

Vizibilitatea în UML
- private
+ public
protected
~ access de tip package

Exemple

```
package pachet1;  
  
public class A1 {  
    private int x;  
    public int y;  
    protected int z;  
    int t;  
}
```

```
package pachet1;  
  
class B1 {  
    public void metodaB1() {  
        A1 ob = new A1();  
        ob.x = 1; //Eroare  
        ob.y = 1; //Corect  
        ob.z = 1; //Corect  
        ob.t = 1; //Corect  
    }  
}
```

```
package pachet2;  
class A2 {  
    public void metodaA2() {  
        pachet1.A1 ob = new pachet1.A1();  
        ob.x = 1; //Eroare  
        ob.y = 1; //Corect  
        ob.z = 1; //Eroare  
        ob.t = 1; //Eroare  
    }  
}  
  
class B2 extends pachet1.A1 {  
    public B2() {  
        x = 1; //Eroare  
        y = 1; //Corect  
        z = 1; //Corect  
        t = 1; //Eroare  
    }  
    public void metodaB2() {  
        pachet1.A1 ob = new pachet1.A1();  
        ob.x = 1; //Eroare  
        ob.y = 1; //Corect  
        ob.z = 1; //Eroare (Da! E eroare!)  
        ob.t = 1; //Eroare  
    }  
}
```



Organizarea codului

(sursă / mașină)

Dr. Petru Florin Pălinescu

Organizarea codului

(sursă / mașină)

Dr. Petru Florin Mihailescu

```
package ro.upt.oop.curs.ceasuri;
public interface ClockType {
    public void setTime(int h, int n, int s);
    public String toString();
}

abstract class AbstractClock implements ClockType {

    private int hour, minute, second;

    public AbstractClock() {
        hour = minute = second = 0;
    }

    public void setTime(int h, int m, int s) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
        second = (s >= 0) && (s < 60) ? s : 0;
    }

    public String toString() {
        return "Current time " + hour + ":" +
            minute + ":" + second;
    }
}
```

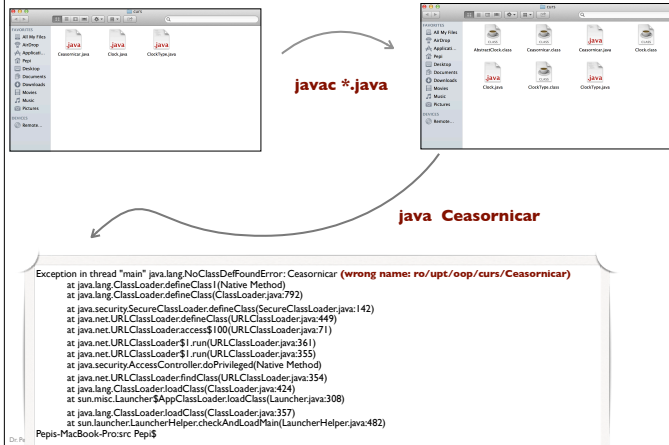
```
package ro.upt.oop.curs.ceasuri;
public class Clock extends AbstractClock {
    public String toString() {
        return "Normal clock - " + super.toString();
    }
}
```

**Presupunem că
avem acest program**

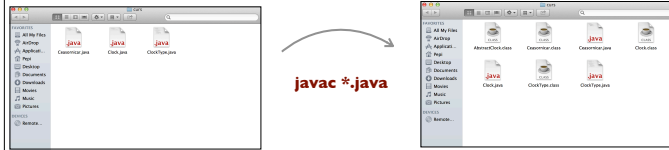
```
package ro.upt.oop.curs;
import ro.upt.oop.curs.ceasuri.*;
public class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }

    public static void main(String argv[]) {
        Clock c = new Clock();
        Ceasornicar om = new Ceasornicar();
        om.regleaza(c);
        System.out.println(c);
    }
}
```

Compile & Rulare (I)



Compile & Run (I)



`javac *.java`

`java ro.upt.oop.curs.Cesornicar`

Error: Could not find or load main class ro.upt.oop.curs.Cesornicar

Fişierele **class** pe disc

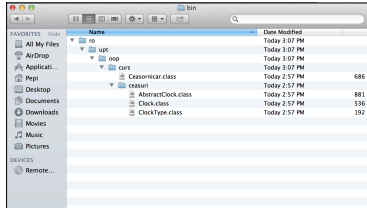
Într-o unitate de compilare avem:

```
package nume1.nume2.nume3;  
class X {  
...  
}  
public class Y {  
...  
}
```

Pe disc trebuie să avem:

Folder **bin** (sau oricum) undeva pe disc

- **nume1** (folder)
 - **nume2** (subfolder **nume1**)
 - **nume3** (subfolder **nume2**)
 - **X.class**
 - **Y.class**



Compilatorul poate face distribuția în mod automat:

```
javac -d cale_catre_bin *.java
```

Calea către clase (classpath)

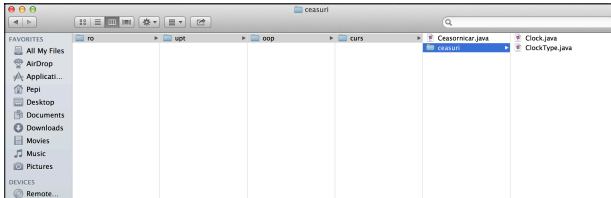
Uzual argument al compilatorului și mașinii virtuale (-cp)

- o listă de căi către folderele în care am distribuit fișierele class
- elementele listei se separă cu : în sisteme unix și ; sub windows
ex. cale1:cale2:cale3:...:caleN
- din aceste foldere se va începe căutarea bytecode-ului unei clase când este necesar; pe baza numelui complet al clasei se caută corespunzător sub-folderele și fișierul class
- în această listă pot apare și arhive java (fișiere jar)

java -cp listă_căi_către_bin ro.upt.oop.curs.Ceasornicar

Compilare (II)

La fel ca și fișierele class, sursele (fișierele .java) se distribuie și ele pe foldere/sub-foldere conform numelui pechetului în care sunt amplasate



Compilare (II)

Compilăm individual fiecare pachet în
ordinea inversă a **dependențelor**



1. `javac -d cale_către_bin cale_către_sursele_pachetului1/*.java`

2. `javac -d cale_către_bin cale_către_sursele_pachetului2/*.java`

```
src/ro/upit/oop/curs/Ceasornicar.java:2: error: package
ro.upit.oop.curs.ceasuri does not exist
import ro.upit.oop.curs.ceasuri.*;
^
src/ro/upit/oop/curs/Ceasornicar.java:4: error: cannot find
symbol
  public void regleaza(ClockType x) {
                ^
symbol:   class ClockType
location: class Ceasornicar
```

Compilare (II)

Compilăm individual fiecare pachet în ordinea inversă a **dependențelor**



1. `javac -d cale_către_bin cale_către_sursele_pachetului I/*.java`

2. `javac -d cale_către_bin -cp listă_căi_către_bin_deja_compilate cale_către_sursele_pachetului/*.java`

dacă avem dependențe circulare e problematic și este un semn că organizarea pe pachete e deficitară

Calea către surse (sourcepath)

Argument al **compilerului**

- o listă de căi către folderele în care am distribuit fișierele sursă
- dacă compilăm o unitate de compilare ce depinde de clase necomilate încă, în aceste foldere se caută **sursele** claselor necesare urmărind subfolderele după numele pachetelor respectivelor clase