

Programare Orientată pe Obiecte

Relația de Moștenire

Dr. Petru Florin Mihancea

V20180924



Moștenirea

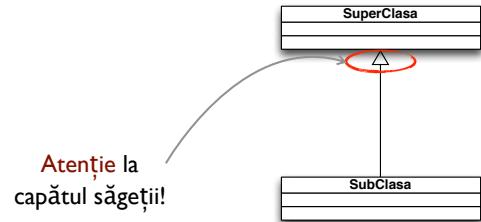
Booch - OO Analysis and Design

... este o relație între clase prin care o clasă (superclasă / clasă de bază) pune la dispoziția altor clase (subclase / clase derivate) structura și comportamentul definite de ea

Booch - OO Analysis and Design

În cod Java și în UML

```
class SuperClasa {  
}  
...  
class SubClasa extends SuperClasa {  
}  
...
```



Definiție

Programarea orientată pe obiecte
este o metodă de **implementare a programelor**
în care acestea sunt organizate ca și
colecții de obiecte ce cooperează între ele [...],
fiecare obiect fiind o instanță a unei clase, și
fiecare clasă fiind membră a unei ierarhii de clase
[clase unite prin relații de moștenire]

Booch - OO Analysis and Design

Dr. Petru Florin Mihăescu

Semantica - două “arome”

a. Moștenire de clasă / de implementare

b. Moștenire de tip / de interfață

de multe ori **același** mecanism/construcție de limbaj (ex. extends între clase) le furnizează pe amândouă

1

Moștenirea de clasă / implementare

Problema

```
class Clock {  
    private int hour, minute, second;  
  
    public Clock() {  
        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;  
    }  
}
```

După un timp, vrem să avem
și ceasuri care indică milisecunda

```
class EnhancedClock {  
    private int hour, minute, second, millisecond;  
  
    public EnhancedClock() {  
        hour = minute = second = millisecond = 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 void setTime(int h, int m, int s, int ms) {  
        setTime(h, m, s);  
        millisecond = (ms >= 0) && (ms < 1000) ? ms : 0;  
    }  
  
    public String toString() {  
        return "Current time " + hour + ":" +  
            minute + ":" + second + ":" + millisecond;  
    }  
}
```

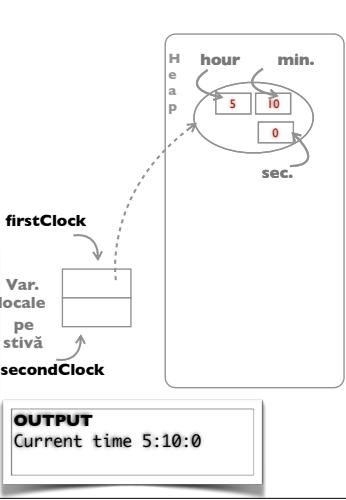
Programarea în stil copy-paste
(e.g., copiezi și bug-uri, dacă e
necesară o modificare, trebuie
modificat și acolo unde ai dat
paste, etc.)

Alternativa

```
class EnhancedClock extends Clock {  
    private int millisecond;  
  
    public EnhancedClock() {  
        millisecond = 0;  
    }  
  
    public void setTime(int h, int m, int s, int ms) {  
        setTime(h, m, s);  
        millisecond = (ms >= 0) && (ms < 1000) ? ms : 0;  
    }  
  
    public String toString() {  
        return super.toString() + ":" + millisecond;  
    }  
}
```

```
class Clock {  
    private int hour, minute, second;  
    public Clock() {  
        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;  
    }  
}
```

```
class Main {  
    public static void main(String argv[]) {  
        Clock firstClock;  
        EnhancedClock secondClock;  
        firstClock = new Clock();  
        firstClock.setTime(5, 10, 0);  
        System.out.println(firstClock);  
    }  
}
```



OUTPUT
Current time 5:10:0

```

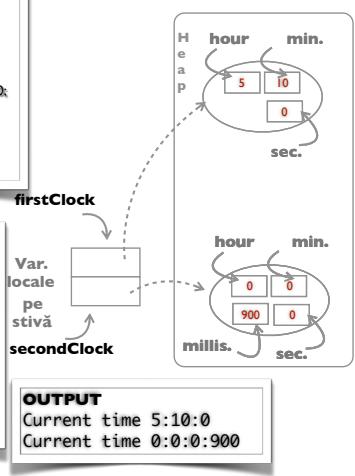
class EnhancedClock extends Clock {
    private int millisecond;
    public EnhancedClock() {
        millisecond = 0;
    }
    public void setTime(int h, int m, int s, int ms) {
        setTime(h, m, s);
        millisecond = (ms >= 0) && (ms < 1000) ? ms : 0;
    }
    public String toString() {
        return super.toString() + ":" + millisecond;
    }
}

```

```

class Main {
    public static void main(String args[]) {
        Clock firstClock;
        EnhancedClock secondClock;
        firstClock = new Clock();
        firstClock.setTime(5, 10, 0);
        System.out.println(firstClock);
        secondClock = new EnhancedClock();
        secondClock.setTime(19, 30, 45);
        secondClock.setTime(0, 0, 900);
        System.out.println(secondClock);
    }
}

```

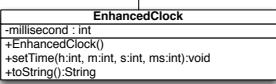
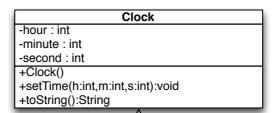


OUTPUT
 Current time 5:10:0
 Current time 0:0:0:900

Dr. Perla Florin Marinescu

Moștenirea (de clasă)

Un mecanism simplu de reutilizare de cod :)



Nu neapărat cel mai adecvat pt.
a rezolva o problemă !

A

Specificatori de acces, constructori, etc. în contextul moștenirii

Dr. Petru Florin Mihance

Modifieri/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)

Vizibilitatea în UML

- private
- + public
- # protected

Exemple

```
class SubClasa extends SuperClasa {
    public void metoda(SuperClasa x) {
        super_a = 1; //Corect
        super_b = 2; //Eroare de compilare
        super_c = 3; //Corect
        x.super_a = 1; //Corect
        x.super_b = 2; //Eroare de compilare
        x.super_c = 3; //Corect (daca superclasa/subclasa in celasi pachet)
    }
}

class Client {
    public void metoda() {
        SuperClasa sp = new SuperClasa();
        SubClasa sb = new SubClasa();
        sp.super_a = 1; //Corect
        sp.super_b = 2; //Eroare de compilare
        sp.super_c = 3; //Corect (daca superclasa/subclasa in celasi pachet)
        sb.super_a = 1; //Corect
        sb.super_b = 2; //Eroare de compilare
        sp.super_c = 3; //Corect (daca superclasa/subclasa in celasi pachet)
    }
}
```

```
class SuperClasa {
    public int super_a;
    private int super_b;
    protected int super_c;
}
```

Uzual, sunt folosiți pt. a aduce un obiect nou creat în starea inițială (atribuirea unor valori inițiale la variabilele instantă)

În contextul moștenirii, cine ar fi cel mai în măsură să initializeze câmpurile moștenite de la o clasă de bază?

Constructori

Constructorul acelei superclase!

prima instrucție dintr-un constructor e fie apel la alt constructor al aceleiași clase, fie apel la un constructor din superclasa directă

Constructori (II)

```
class Clock {  
    ...  
    public Clock() {  
        hour = minute = second = 0;  
    }  
    ...  
}
```



```
class EnhancedClock extends Clock {  
    ...  
    public EnhancedClock() {  
        millisecond = 0;  
    }  
    ...  
}
```

Dacă în superclasă există un **constructor no-arg** (default sau nu) compilatorul introduce automat un apel la acel constructor ca primă instrucțiune

Constructori (III)

```
class Clock {  
    ...  
    public Clock(int h, int m, int s) {  
        hour = h;  
        minute = m;  
        second = s;  
    }  
    ...  
}
```

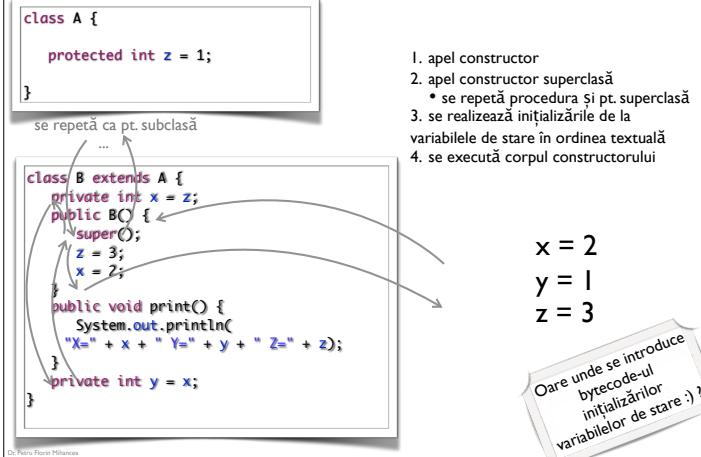
Dacă avem numai constructori cu argumente în clasa de bază, în constructorul subclaserilor trebuie apelat **explicit** un constructor din superclasă

```
class EnhancedClock extends Clock {  
    ...  
    public EnhancedClock(int h,  
                         int m, int s, int ms) {  
        super(h,m,s);  
        millisecond = ms;  
    }  
    ...  
}
```

și dacă singurul rol al constructorului din subclasă e acest apel :)

nu e o raritate ca argumentele constructorului subclasei să fie necesare ca argumente pt. apelul la constructorul clasei de bază

Ordinea inițializărilor (aprox)



1. apel constructor
2. apel constructor superclăsă
 - se repetă procedura și pt. superclăsă
3. se realizează inițializările de la variabilele de stare în ordinea textuală
4. se execută corpul constructorului

x = 2
y = 1
z = 3

Dr. Raluca Florica Mihăescu

Alte elemente importante

O clasă poate extinde direct cel mult o clasă

NU există moștenire multiplă în Java

(între clase, după extends apare o singură clasă)

Clasa Object

Automat superclasă pt. orice clasă care nu extinde ceva

Implicit, e moștenită (direct ori indirect) de orice clasă Java

Acesta e motivul pt. care metodele `equals`, `toString`, etc. (vezi capitol anterior) pot fi apelate pe instanțele oricărei clase ;)

Dr. Petru Florin Mihăescu

B

Redefinirea metodelor (method overriding)

(method overriding)

Dr. Petru Florin Mihance

Redefinirea (overriding)

uneori, implementarea moștenită a unei operații nu e adekvată/suficientă pentru o subclasă

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

într-o subclasă putem redefini (override) o metodă instantă accesibilă

implementarea moștenită se poate accesa apelând metoda cu super în față

modifierii de acces pot fi schimbați dar trebuie să ofere cel puțin la fel de multă vizibilitate iar tipul returnat poate fi schimbat cu un subtip

```
class EnhancedClock extends Clock {  
    ...  
    public String toString() {  
        return super.toString() + ":" + millisecond;  
    }  
}
```

în capitolele anterioare, când reimplementam equals, toString ... din clasa Object făceam overriding

Putem controla :)

```
class ClassName {  
    final public void doSomething()  
    {  
        ...  
    }  
}
```

Este o eroare de compilare dacă cineva încearcă să redefinăască metoda

```
final class ClassName {  
    ...  
}
```

Este o eroare de compilare dacă cineva extinde clasa ClassName

overriding vs. overloading

```
class Clock {  
    private int hour, minute, second;  
    public Clock() { 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;  
    }  
}  
  
class EnhancedClock extends Clock {  
    private int millisecond;  
    public EnhancedClock() { millisecond = 0; }  
    public void setTime(int h, int m, int s, int ms) {  
        setTime(h, m, s);  
        millisecond = (ms >= 0) && (ms < 1000) ? ms : 0;  
    }  
    public String toString() {  
        return super.toString() + ":" + millisecond;  
    }  
}
```

suprascriere /
redefinire / overriding
același nume, aceeași
semnătură

?
supraîncărcare
(overloading)
același nume,
semnăruri diferite

Ascunderea (hiding) variabilelor instanță

Dr. Petya Florin Milance

Ascunderea var. instantă

într-o subclasă putem avea variabile instanță cu același nume ca și alte variabile instanță vizibile dar declarate în superclase

```
class Point {  
    protected int x = 2;  
}  
  
class MyPoint extends Point {  
    protected double x = 4.7;  
  
    public void printBoth() {  
        System.out.println(x + " " + super.x);  
    }  
  
    public static void main(String[] args) {  
        MyPoint ex = new MyPoint();  
        ex.printBoth();  
        System.out.println(ex.x + " " + ((Point)ex).x);  
    }  
}
```

se spune că declarația lui x din MyPoint ascunde declarația lui x din Point

Important deoarece trebuie să fim atenți la ce accesăm :)

Din subclasă, folosind super la accesul variabilei instanță

Se poate și așa dar atenție (să nu umbăr la date din afara claselor implicate)

Există și alte “ascunderi”

ascunderea metodelor statice cu alte metode statice

ascunderea câmpurilor statice cu alte câmpuri statice



Pentru detalii vedeti
Java Language Specification

Hiding vs. Overriding în Java

Metode cu aceeasi semnatură	Metodă instantă superclasă	Metodă statică superclasă
Metodă instantă subclasă	Overriding / Redefinire	Eroare compilare
Metodă statică subclasă	Eroare compilare	Hiding / Ascundere
Metodele instantă pot fi ascunse (NU pot fi redefinite și implicit mecanismele bazate pe overriding NU merg la metodele statice)		Cele statice doar

D

Când folosim?

Moștenirea de clasă vs. compunerea obiectelor (compunerea - object composition)

Dr. Petru Florin Mihaiță

**Euristică Importantă a
Programării Orientate pe Obiecte**

**Nu folosiți moștenirea doar pentru
a reutiliza codul unei superclase**

(Nu folosiți moștenirea exclusiv ca moștenire de clasă; folosiți-o numai
când e folosită și ca moștenire de tip)

**Favorizează compunerea
obiectelor în locul
moștenirii de clasă**

GOF

Dr. Petru Florin Milăneș

Studiu de caz

Vector
...
+add(o: Object) : boolean
+get(index: int) : Object
+isEmpty() : boolean
+removeElementAt(index:int) : void

"Simulează" un tablou a cărui capacitate se poate modifica după necesitățile execuției.
Presupunem că e **deja implementat**.

```
class Stack extends Vector {  
    public Object push(Object o) {  
        this.add(o);  
        return o;  
    }  
    public Object pop() {  
        Object r = this.get(this.size() - 1);  
        this.removeElementAt(this.size() - 1);  
        return r;  
    }  
    ...  
}
```

Stack
...
+push(o: Object) : Object
+pop() : Object
+peek() : Object
+empty() : boolean

Structură de date LIFO (Last-In-First-Out)
cu operațiile uzuale
push - adăugare în vârful stivei
pop - scoaterea elementului din vârful stivei.
Presupunem că **trebuie implementată**.

Studiu de caz

Vector

```
+add(o: Object) : boolean  
+get(index: int) : Object  
+isEmpty() : boolean  
+removeElementAt(index:int) : void  
...
```

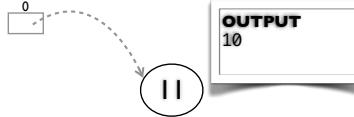


Stack

```
+push(o:Object) : Object  
+pop() : Object  
+peek() : Object  
+empty() : boolean
```

```
class Main {  
    public static void main(String args[]) {  
        Stack stk = new Stack();  
        stk.push(new Integer(5));  
        stk.push(new Integer(10));  
        Object p = stk.pop();  
        System.out.println(p);  
  
        stk.add(new Integer(11));  
        // cu add (?) ce operatie este asta pt. notiunea de stivă ?  
        // adică cum (!!!) că într-o stivă trebuie să pot  
        // accesa doar ultimul element introdus  
        ...  
    }  
}
```

În obiectul stivă (vârful e cea mai din dreapta intrare)

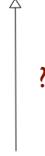


Handwriting practice area (15 horizontal lines for notes)

Studiu de caz

Vector

```
+add(o: Object) : boolean  
+get(index: int) : Object  
+isEmpty() : boolean  
+removeElementAt(index:int) : void  
...
```



Stack

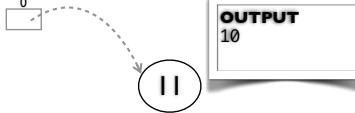
```
+push(o: Object) : Object  
+pop() : Object  
+peek() : Object  
+empty() : boolean
```

```
class Main {  
    public static void main(String args[]) {  
        Stack stk = new Stack();  
        stk.push(new Integer(5));  
        stk.push(new Integer(10));  
        Object p = stk.pop();  
        System.out.println("Obiectul din stivă este " + p);  
    }  
}
```

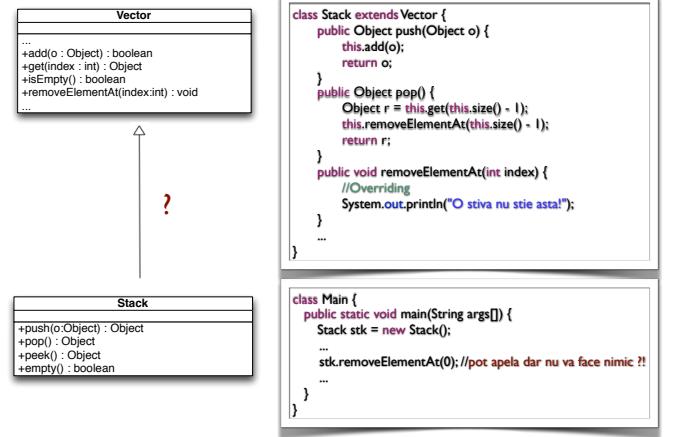
Operații care nu caracterizează notiunea de stivă pot fi folosite pe o stivă !?

Exemplu: `System.out.println("Obiectul din stivă este " + p);` că într-o stivă trebuie să pot // accesa doar **ultimul** element introdus

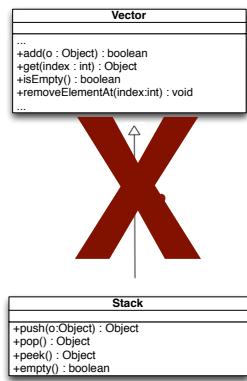
În obiectul stivă (vârful e cea mai din dreapta intrare)



Asta NU e soluția ...



Asta NU e soluția ...



```
class Stack extends Vector {  
    public Object push(Object o) {  
        this.add(o);  
        return o;  
    }  
  
    public Object pop() {  
        Object r = this.get(this.size() - 1);  
        this.removeElementAt(this.size() - 1);  
        return r;  
    }  
  
    public void removeElementAt(int index) {  
        //Overriding  
        System.out.println("O stivă nu stie asta!");  
    }  
}
```



```
class Main {  
    public static void main(S  
    Un obiect/clasă pune împreună  
    datele și operațiile ce operează  
    pe acele date ... iar metoda  
    noastră nu face nimic deci ce  
    căută în interfață obiectului ?  
    !!
```

Compunerea obiectelor

În esență, amplasarea de referințe la obiecte ca variabile instanță într-o clasă
(inclusiv când referințele sunt într-un tabou variabilă instanță)

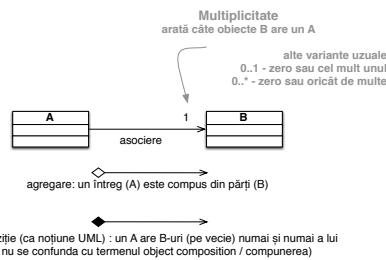
```
class Car {  
    private Engine engine;  
    public Car(Engine e) {  
        engine = e;  
    }  
    public void accelerate() {  
        engine.increaseFuelFlow();  
    }  
}
```

```
class Engine {  
    public void increaseFuelFlow() {  
        ...  
    }  
}
```

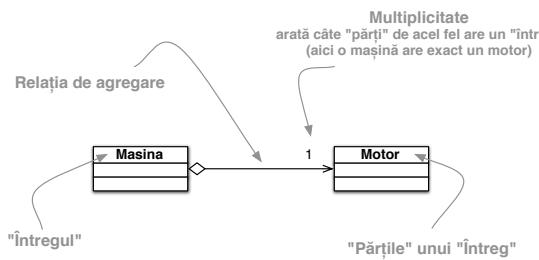
Uzual spune că o mașină are un motor.
Compunerea este o relație de tip has-a

Reprezentarea în diagrame UML de clase

```
class A {  
    private B bx;  
    ...  
}
```



Reprezentarea în diagrame UML de clase (II)



De ce compunerea ?

... folosită de exemplu când dorim ca feature-ul unei clase să fie folosit într-o clasă pe care o implementăm dar NU și în interfața obiectelor definite de noua clasă

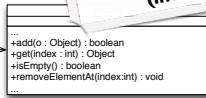
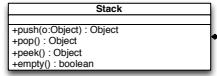


Stiva corectată ...

```
class Stack {  
    private Vector v = new Vector();  
    public Object push(Object o) {  
        v.add(o);  
        return o;  
    }  
    public Object pop() {  
        Object r = v.get(v.size() - 1);  
        v.removeElementAt(v.size() - 1);  
        return r;  
    }  
}
```

```
class Main {  
    public static void main(String args[]) {  
        Stack stk = new Stack();  
        stk.push(new Integer(5));  
        stk.push(new Integer(10));  
        Object p = stk.pop();  
        System.out.println(p);  
  
        stk.add(new Integer(11)); // eroare compilare  
        stk.removeElementAt(0); // eroare execuție  
        ...  
    }  
}
```

... în plus putem schimba
Vectorul cu altceva fără să
afectăm clientii stivei
(inclusiv la rulare)



o stivă are un vector (numai și numai al ei)

Componere vs. Moștenire

Componerea este o relație has-a

- o mașină **are un** motor
- o stivă **are un** vector în care își ține elementele
- dă naștere unei ierarhii de obiecte

Moștenirea este o relație is-a

- un EnhancedClock **este un** fel de Clock
- un Triunghi **este un** fel de FigurăGeometrică
- o Pisică **este un** fel de Felină
- dă naștere unei ierarhii de clase

B is-a A : B se și comportă ca un A

```
class Clock {  
    private int hour, minute, second;  
    public Clock() {  
        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;  
    }  
}
```

comportamental știe să:

- 1) își seteze ora, minutul, secunda
- 2) se reprezintă ca sir de caractere

```
class EnhancedClock extends Clock {  
    private int millisecond;  
    public EnhancedClock() {  
        millisecond = 0;  
    }  
    public void setTime(int h, int m, int s, int ms) {  
        setTime(h, m, s);  
        millisecond = (ms >= 0) && (ms < 1000) ? ms : 0;  
    }  
    public String toString() {  
        return super.toString() + ":" + millisecond;  
    }  
}
```

comportamental știe să:

- 1) își seteze ora, minutul, secunda
- 2) se reprezintă ca sir de caractere
- 3) își seteze ora, minutul, secunda, milisecunda

M-ar deranja dacă eu văș cere un
Clock și voi mi-ăti da un
EnhancedClock?

Nu, pt. că pot face cu un
EnhancedClock tot ce mi-am propus cu
un ceas normal.

Invers nu e adevărat!

B is-a A : B se și comportă ca un A

```
class Clock {  
    private int hour, minute, second;  
    public Clock() {  
        hour = minute = second = 0;  
    }  
    public void setTime(int h, int m, int s) {  
        hour = (h >= 0) && (h <= 23) ? h : 0;  
        minute = (m >= 0) && (m <= 59) ? m : 0;  
        second = (s >= 0) && (s <= 59) ? s : 0;  
    }  
    public String toString() {  
        return "Current time:  
                " + hour + ":" + minute + ":" + second;  
    }  
}
```

comportamental știe să:

- 1) își seteze ora, minutul, secunda
- 2) se reprezintă ca sir de caractere

M-ar deranja dacă eu văd cere un
Clock și voi mi-ți da un
EnhancedClock?

Nu, pt. că pot face cu un
EnhancedClock tot ce mi-am propus cu
un ceas normal.

comportamental știe să:
1) își seteze ora, minutul, secunda
2) se reprezintă ca sir de caractere
3) își seteze ora, minutul, secunda, milisecunda

```
class EnhancedClock extends Clock {  
    private int millisecond;  
    public EnhancedClock() {  
        millisecond = 0;  
    }  
    public void setTime(int h, int m, int s, int ms) {  
        hour = (h >= 0) && (h <= 23) ? h : 0;  
        minute = (m >= 0) && (m <= 59) ? m : 0;  
        second = (s >= 0) && (s <= 59) ? s : 0;  
        millisecond = (ms >= 0) && (ms < 1000) ? ms : 0;  
    }  
    public String toString() {  
        return super.toString() + ":" + millisecond;  
    }  
}
```

Se spune că EnhancedClock este un
subtip de-al lui Clock și deci
mostenirea e folosită aici nu numai ca
mostenire de clasă. Hai să vedem
mostenirea de tip :

subtip

mostenirea de tip :

mostenirea de tip :

Invers nu e adevărat!

2

Moștenirea de de tip / interfață și polimorfismul

(de subtíp)

Noțiunea de tip

Un tip de date (abstract) definește o mulțime de valori / "obiecte" (abstracte) complet caracterizate de operațiile disponibile peste ele

Barbara Liskov

Similare sunt și tipurile de date primitive ne interesează cum creem "obiecte" ex. int într-un program și care sunt operațiile prin care le putem prelucra

Moștenirea de tip

**Se referă la o relație între tipuri
în sensul că ...**

... un tip (subtip**) moștenește
operațiile unui alt tip (**supertip**)**

Tipuri vs. Clase

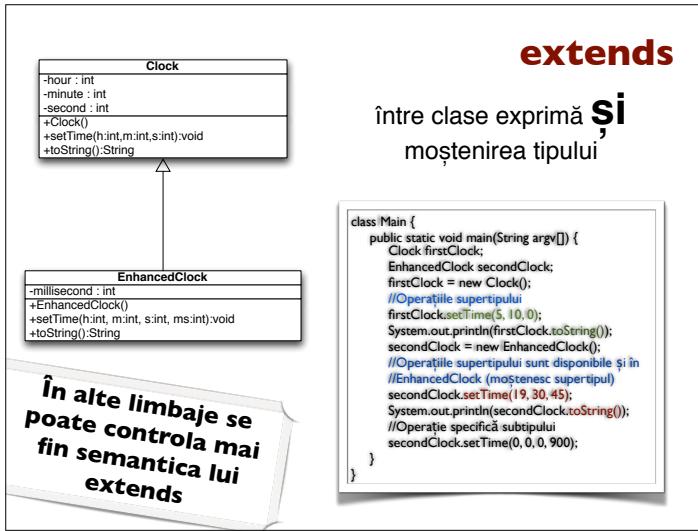
Clock
- hour : int
- minute : int
- seconds : int
+ Clock(h : int, m : int, s : int)
+ setTime(h : int, m : int, s : int) : void
+ toString() : String

interfața obiectului
reprezintă setul
de operații ce se pot efectua
pe un obiect și deci
denotă **tipul**
aceluia obiect

Conceptual clasa diferă de noțiunea de tip

- o clasă reprezintă de fapt **implementarea** unui tip
- tipul e dat doar de **declarațiile** operațiilor publice din clasă

... dar des folosite ca sinonime, deoarece o clasă specifică și
operațiile publice, deci și interfața și deci și tipul



extends

între clase exprimă **și**,
moștenirea tipului

```

class Main {
    public static void main(String args[]) {
        Clock firstClock;
        EnhancedClock secondClock;
        firstClock = new Clock();
        //Operatiile supertipului
        firstClock.setTime(5, 10, 0);
        System.out.println(firstClock.toString());
        secondClock = new EnhancedClock();
        //Operatiile supertipului sunt disponibile și în
        //EnhancedClock (moștenesc supertipul)
        secondClock.setTime(19, 30, 45);
        System.out.println(secondClock.toString());
        //Operatiile specifică subtipului
        secondClock.setTime(0, 0, 900);
    }
}

```

Nu folosiți moștenirea doar pentru a reutiliza codul unei superclase. Favorizează compunerea obiectelor în locul moștenirii de clasă.

Să ne amintim ...

class Main {

Vector

```
+add(o : Object) : boolean  
+get(index : int) : Object  
+isEmpty() : boolean  
+removeElementAt(index:int) : void  
...
```

X

Stack

```
+push(o:Object) : Object  
+pop() : Object  
+peek() : Object  
+empty() : boolean
```

Tipul Stivă nu e caracterizat de operațiile tipului Vector, deci nu e subtip de-al lui Vector !

**stk.add(new Integer(1));
// cu add () ce operatie e asta pt. noțiunea de stivă ?
stk.removeElementAt(0);
// adică cum (!!!!) că într-o stivă trebuie să pot
// accesa doar **ultimul** element introdus**

punând extends am folosi doar o "aromă" a lui (adică moștenirea de clasă) iar moștenirea de tip nu, încălcând prima euristică :(

A

Clase și metode abstracte

(ce, de ce și pentru ce?)

Dr. Petru Florin Milăneș

Cum “declarăm” un tip ?

... fără niciun fel de detaliu de implementare

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

```
(abstract)  
ClockType  
+setTime(h:int,m:int,s:int).void  
+toString().String
```

numele clasei/metoda se
scrie italic (se adaugă **abstract**)
când sunt scrise de mână

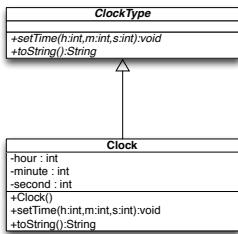
Metodă instantă **abstractă**
• **NU** are corp de instrucțiuni
• rămâne ca o subclasă să-i dea o
implementare prin **overriding**
• clasa în care apare trebuie
declarată **abstractă**

```
class Main {  
    public static void main(String argv[]) {  
        //Putem declara referințe  
        ClockType aClock;  
  
        ...  
        //Este o eroare de compilare dacă încercăm  
        //să instanțiem o clasă abstractă (ex.mai jos)  
        aClock = new ClockType(); // oare de ce?  
    }  
}
```

Cum implementăm tipul ?

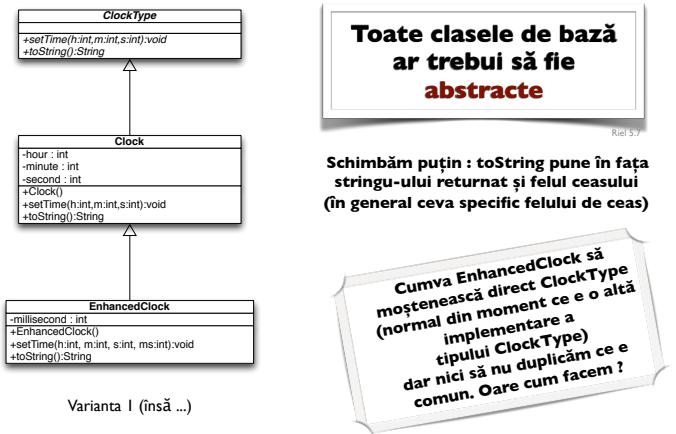
```
abstract class ClockType {  
    public abstract void setTime(int h, int m, int s);  
    public abstract String toString();  
}
```

```
class Clock extends ClockType {  
    private int hour, minute, second;  
    public Clock() {  
        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;  
    }  
}
```

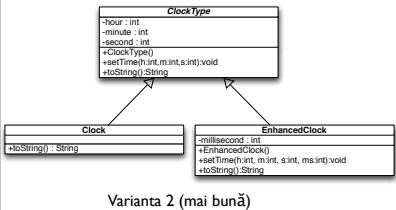


Notă - dacă într-o clasă nu dăm implementare pt. o metodă abstractă dintr-o superclasă, clasa trebuie declarată abstractă (altfel eroare de compilare)

Exemplul cu ceasurile ...



Exemplul cu ceasurile ... (II)



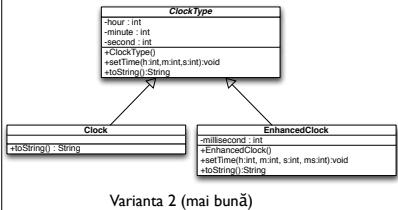
Varianta 2 (mai bună)

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

```
abstract class ClockType {  
    private int hour, minute, second;  
    public ClockType() {  
        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;  
    }  
}
```

Obs. - o clasă poate fi declarată **abstractă** și dacă nu are metode abstractive (și poate avea câmpuri, constructori pt. a-i inițializa, etc.). Prin urmare poate și factoriza codul comun.

Exemplul cu ceasurile ... (II)



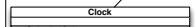
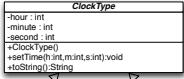
Varianta 2 (mai bună)

```
class EnhancedClock extends ClockType {
    private int millisecond;
    public EnhancedClock() {
        millisecond = 0;
    }
    public void setTime(int h, int m, int s, int ms) {
        setTime(h, m, s);
        millisecond = (ms >= 0) && (ms < 1000) ? ms : 0;
    }
    public String toString() {
        return "Enhanced clock - " + super.toString() + ":" +
            millisecond;
    }
}
```

```
abstract class ClockType {
    private int hour, minute, second;
    public ClockType() {
        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;
    }
}
```

Obs. - o clasă poate fi declarată abstractă și dacă nu are metode abstractive (și poate avea câmpuri, constructori pt. a-i inițializa, etc.). Prin urmare poate și factoriza codul comun.

Exemplul cu ceasurile ... (III)



Am mixat în superclasa abstractă declararea tipului cu factorizarea implementării comune

Varianta 2 (mai bună)

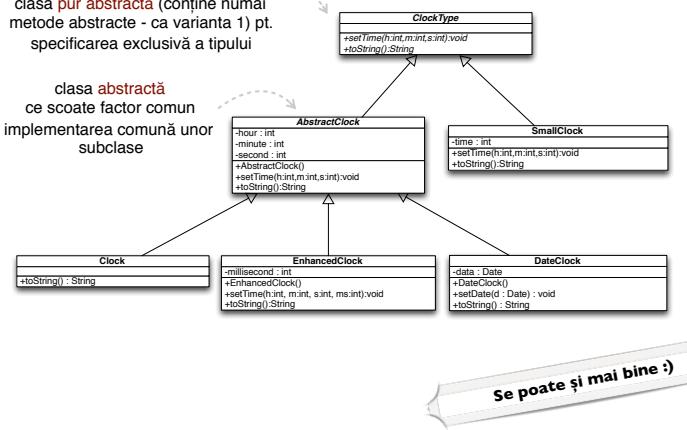
Dacă apar noi implementări ale tipului ClockType unde le punem în ierarhie ?
ex. **SmallClock** care codifică ora, minutul și secunda în octeți separați ai aceluiași int ?

Nasol,
dar se poate mai bine :)

Exemplul cu ceasurile ... (IV)

clasa pur abstractă (conține numai metode abstracte - ca varianta 1) pt. specificarea exclusivă a tipului

clasa abstractă
ce scoate factor comun
implementarea comună unor
subclase



Se poate și mai bine :)

abstract

Metoda abstractă

- doar declarăția metodei, specificând o operație a unui tip
- sunt implementate prin overriding în subclase

Clasa abstractă

- NU poate fi instantiată
- pt. a) specificarea tipului + b) reutilizarea de cod comun
 - abstractă pură (doar metode abstracte) - a
 - câmpuri comune/implementări comune de metode - a,b
 - putem avea și metode abstracte (pe lângă concrete) - a,b
 - NU forțați "implementări comune" inexistente; lasați metoda abstractă dacă nu există ceva comun de pus în ea

Dacă o superclasă are rolul de a specifica tipul / codul comun faceți-o abstractă și dacă are numai metode concrete (obiectele ei oricum nu au sens în problema de rezolvat)

B

Polimorfism. Legare dinamică

(+ de ce avem nevoie să declarăm tipuri)

Dr. Petru Florin Milăneș

Efectul moștenirii de tip

Polimorfism

O variabilă referință declarată de un anumit tip (clasa) poate să refere obiecte a aceluiași tip (clase) și a oricărui alt subtip (subclase) de-al său

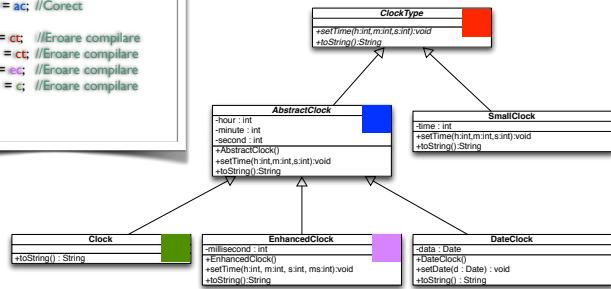
```

class Main {
    public static void main(String[] args) {
        ClockType ct;
        AbstractClock ac;
        Clock c;
        EnhancedClock ec;
        c = new Clock();
        ec = new EnhancedClock();
        ...
        ct = c; //Eroare compilare
        ac = ct; //Corect
        ac = c; //Corect
        ac = ec; //Corect
        ct = ac; //Corect
        ...
        c = ct; //Eroare compilare
        ac = ct; //Eroare compilare
        c = ec; //Eroare compilare
        ec = c; //Eroare compilare
        ...
    }
}

```

Adică ...

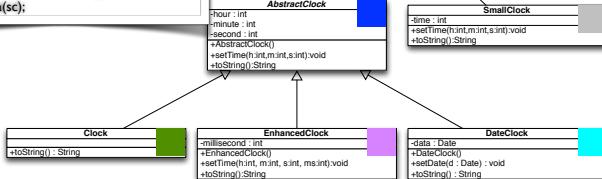
O variabilă referință declarată de un anumit tip (clasă) poate să refere obiecte a acelui tip (clase) și a oricărui alt subtip (subclase) de-al său



Nu știm exact: poate fi **Clock**,
EnhancedClock, **DateClock** sau
SmallClock

```
class Ceasornicar {
    public void regleaza(ClockType x) {
        ...
    }
}
```

```
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
om.regleaza(ec);
om.regleaza(new DateClock());
om.regleaza(sc);
```



Întrebare

Spre ce fel/clasă/tip concret de obiect referă variabila x când se execută metoda regleză ?

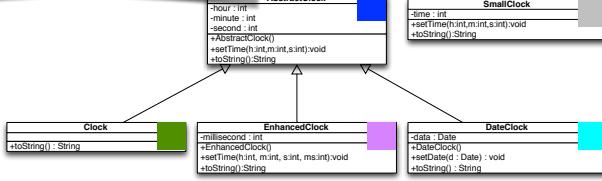
Nu știm exact: poate fi **Clock**,
EnhancedClock sau **DateClock**

Enhanced

Întrebare

**Spre ce fel/clasă/tip concret
de obiect referă variabila x
când se execută metoda
regleză ?**

```
CeasornicarMaiSlabut om = new CeasornicarMaiSlabut();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
om.regleaza(ec);
om.regleaza(new DateClock());
om.regleaza(sc); // Eroare compilare
```



Quizz

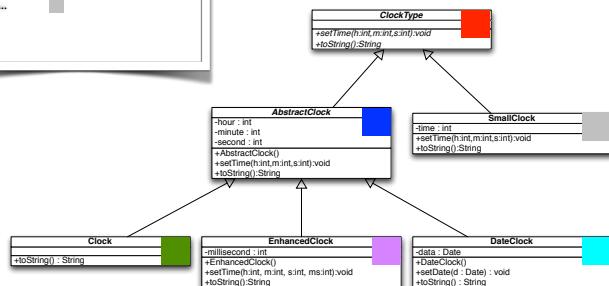
Spre ce fel concret de obiecte poate referi o referință declarată ca fiind de tip **Object** ?

) - vă amintiți de
metoda equals(Object) din
clasa Object ?

instanceof

```
class Ceasornicar {  
    public void regleaza(ClockType x) {  
        if(x instanceof AbstractClock) {  
            ...  
        }  
        if(x instanceof Clock) {  
            ...  
        }  
        if(x instanceof SmallClock) {  
            ...  
        }  
    }  
}
```

Operatorul întoarce true dacă variabila referă un obiect al clasei date ori a unei subclase de-a ei



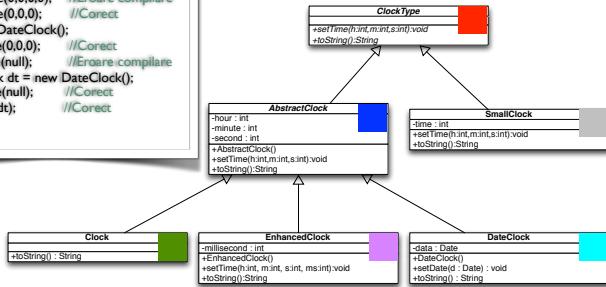
```

class Main {
    public static void main(String[] args) {
        ClockType ct;
        AbstractClock ac;
        Clock c;
        EnhancedClock ec;
        ...
        ct.setTime(0,0); //Corect
        ac.setTime(0,0); //Corect
        c.setTime(0,0); //Corect
        ec.setTime(0,0); //Corect
        ec.setTime(0,0,0); //Corect
        ac = new EnhancedClock();
        ac.setTime(0,0,0); //Eroare compilare
        ac.setTime(0,0);
        ct = new DateClock();
        ct.setTime(0,0);
        ct.setDate(null); //Eroare compilare
        DateClock dt = new DateClock();
        dt.setDate(null); //Corect
        dt.equals(dt);
    }
}

```

Invokeare corectă (la compilare)

Pe o variabilă referință putem invoca orice metodă instată declarată în tipul/clasa referinței sau în unul din supertipurile/supercările sale



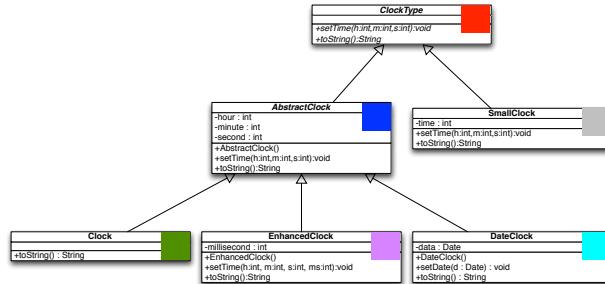
```

class Main {
    public static void main(String[] args) {
        ClockType ct;
        EnhancedClock ec = new EnhancedClock();
        ct = ec; //Up Cast
        ct.setTime(12,0,0); //Eroare compilare
        ((EnhancedClock)ct).setTime(12,0,0);
    }
}

```

(Down) Cast

Necesar dacă trebuie să apelăm metode specifice unui subtip dar avem referințe de un supertip de-al său



(Down) Cast

```
class Ceasornicar {
    public void regleaza(ClockType x) {
        ((EnhancedClock)x).setTime(12.0,0,0);
    }
}
```

Necesar dacă trebuie să apelăm metode specifice unui subtip dar avem referințe de un supertip de-al său

```
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
om.regleaza(c);
```

Exception in thread "main" java.lang.ClassCastException:
 at var3.Ceasornicar.regleaza(Ceasornicar.java:5)
 at var3.main(var3.java:5)

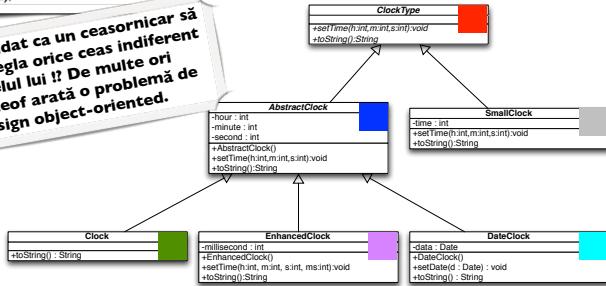
(Down) Cast

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

```
Ceasornicar om = new Ceasornicar();  
Clock c = new Clock();  
om.regleaza(c);
```

Deși e ciudat ca un ceasornic să nu știe să regleze orice ceas indiferent de felul lui ! De multe ori instanceof arată o problemă de design object-oriented.

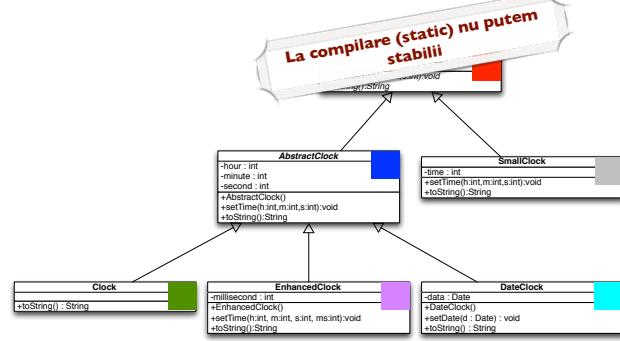
Necesar dacă trebuie să apelăm metode specifice unui subtip dar avem referințe de un supertip de-al său



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

Legarea dinamică

Care implementare ?

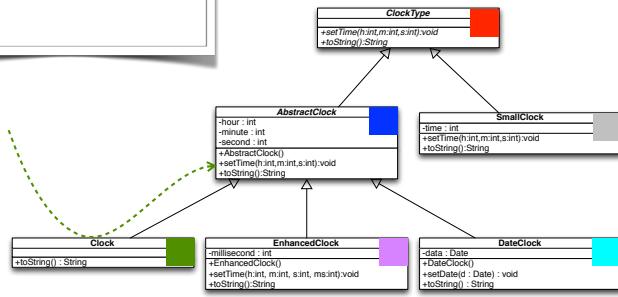


```
class Ceasnicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
Ceasnicar om = new Ceasnicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
```

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**

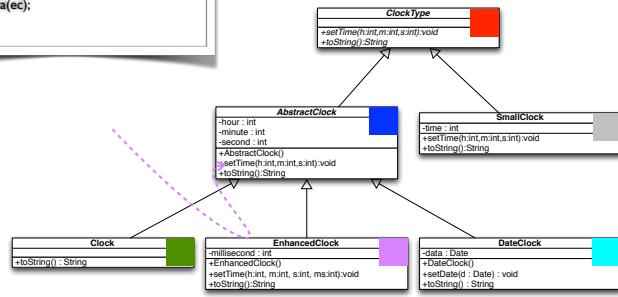


```
class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
om.regleaza(ec);
```

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**

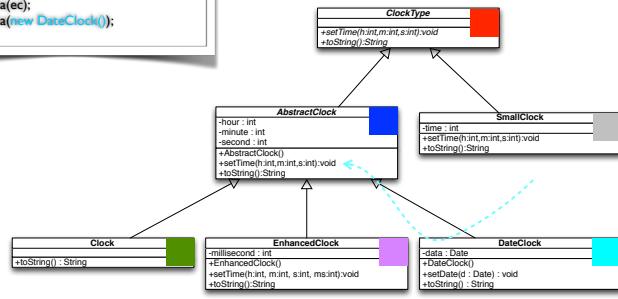


```
class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
om.regleaza(ec);
om.regleaza(new DateClock());
```

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**

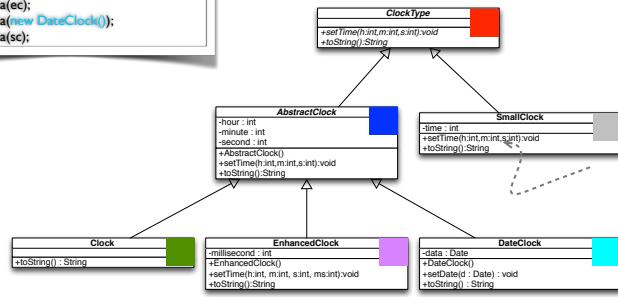


```
class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
om.regleaza(ec);
om.regleaza(new DateClock());
om.regleaza(sc);
```

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**

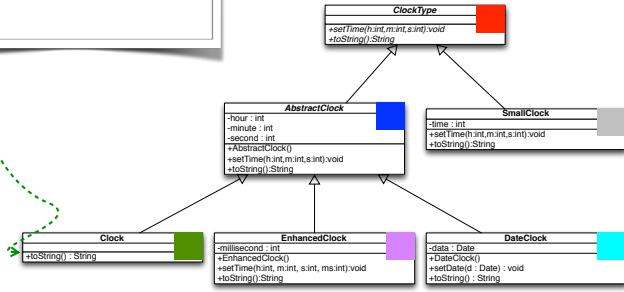


```
class Ceasornicar {  
    public void print(ClockType x) {  
        System.out.println(x.toString());  
    }  
}
```

```
Ceasornicar om = new Ceasornicar();  
Clock c = new Clock();  
EnhancedClock ec = new EnhancedClock();  
SmallClock sc = new SmallClock();  
om.print(c);
```

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**

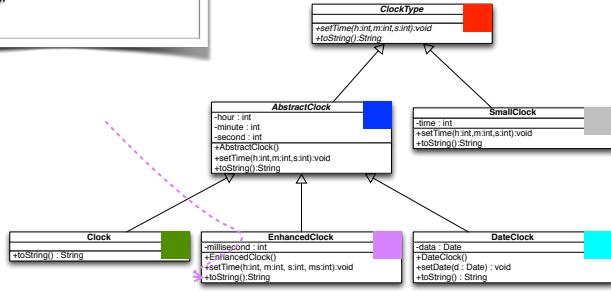


```
class Ceasornicar {
    public void print(ClockType x) {
        System.out.println(x.toString());
    }
}
```

```
Ceasornicar cm = new Ceasornicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
cm.print(c);
cm.print(ec);
```

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**

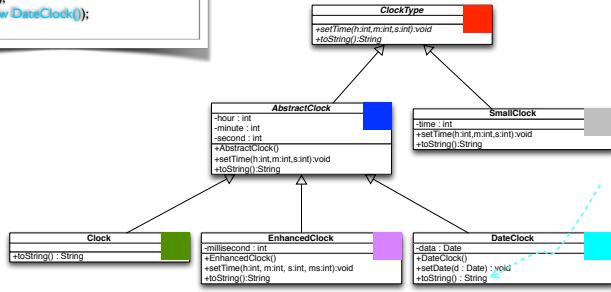


```
class Ceasornicar {
    public void print(ClockType x) {
        System.out.println(x.toString());
    }
}
```

```
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.print(c);
om.print(ec);
om.print(new DateClock());
```

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**

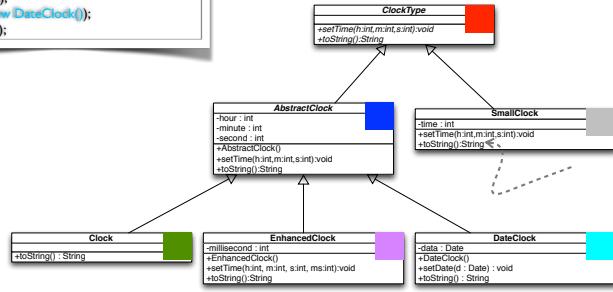


```
class Ceasornicar {
    public void print(ClockType x) {
        System.out.println(x.toString());
    }
}
```

```
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.print(c);
om.print(ec);
om.print(new DateClock());
om.print(sc);
```

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**



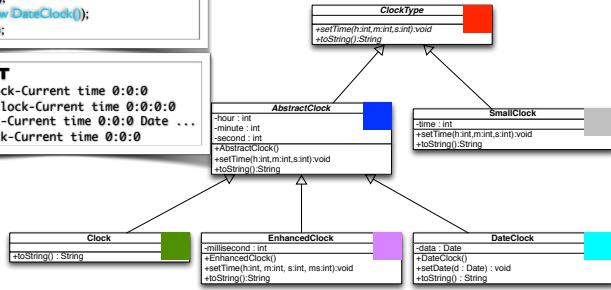
```
class Ceasornicar {
    public void print(ClockType x) {
        System.out.println(x.toString());
    }
}
```

```
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.print(c);
om.print(ec);
om.print(new DateClock());
om.print(sc);
```

OUTPUT
Normal clock-Current time 0:0:0
Enhanced clock-Current time 0:0:0:0
Date clock-Current time 0:0:0 Date ...
Small clock-Current time 0:0:0

Legarea dinamică

În cazul metodelor instanță, putând fi **overridden/redef.**, se stabilește numai la rularea programului (deci **dinamic**) care implementare se apelează **funcție de felul concret al obiectului referit la acel moment de referință din apel**



Atenție ...

**Se aplică dacă e vorba de
overriding**

**NU se aplică atunci când e vorba de
hiding**

**deci niciodată la apeluri de metode statice
deci niciodată la accesarea de câmpuri**

Dr. Petru Florin Milneusca

Quizz

Cât apare pe ecran ?

```
class A {  
    protected int x = 0;  
    public A() {  
        x = doSomething();  
    }  
    public int doSomething() {  
        return 10;  
    }  
    public String toString() {  
        return "" + x;  
    }  
    public static void main(String args[]) {  
        System.out.println(new B());  
        System.out.println(new A());  
    }  
}  
  
class B extends A {  
    public int doSomething() {  
        return 20;  
    }  
}
```

... dar aici ?

```
class C {  
    protected int y = 0;  
    public C() {  
        y = doSomethingElse();  
    }  
    public static int doSomethingElse() {  
        return 10;  
    }  
    public String toString() {  
        return "" + y;  
    }  
    public static void main(String args[]) {  
        System.out.println(new D());  
        System.out.println(new C());  
    }  
}  
  
class D extends C {  
    public static int doSomethingElse() {  
        return 20;  
    }  
}
```

1

De ce toate astea ?

sau cum să fim Harry Potter când organizăm programe OO :)

Programele evoluează

Programele trebuie adaptate în mod continuu la noi cerințe, altfel devin progresiv tot mai nesatisfăcătoare

Una din legile lui Lehman cu privire la evoluția programelor

Dr. Petru Florin Milăneș

Principiul Open-Closed

Entitățile software (ex. clase, metode) să fie deschise la extensii dar închise la modificări

deschise la extensii
să putem extinde (refolosi) funcționalitatea lor
închise la modificări
să le putem extinde dar fără să le modificăm codul

da, sigur ...



Bertrand Meyer
(restated by Robert Martin)

Dr. Petru Florin Mihăescu

Exemplul I

```
class Painter {  
    public void drawAll(Object[] figs) {  
        for(Object aFig : figs) {  
            if(aFig instanceof Circle) {  
                ((Circle)aFig).drawCircle();  
            } else {  
                ((Square)aFig).drawSquare();  
            }  
        }  
    }  
}
```

Un program ce lucrează cu
figuri geometrice

Circle	Square
... +drawCircle() : void	... +drawSquare() : void

După un timp vrem să extindem programul
cu triunghiuri

Exemplul I

```
class Painter {  
    public void drawAll(Object[] figs) {  
        for(Object aFig : figs) {  
            if(aFig instanceof Circle) {  
                ((Circle)aFig).drawCircle();  
            } else {  
                ((Square)aFig).drawSquare();  
            }  
        }  
    }  
}
```

Un program ce lucrează cu
figuri geometrice

Eroare de execuție -
ClassCastException !!!
Compilatorul nu te poate ajuta.

Circle	Square	Triangle
... +drawCircle() : void	... +drawSquare() : void	... +drawTriangle() : void

1. Adăugăm clasa corespunzătoare
2. Peste tot unde am făcut distincție între diverse feluri de figuri, mai adăugăm probabil un **if-instanceof-else**

Painter nu respectă principiul pt.
că trebuie să-l modificăm codul

Dr. Petru Florin Mihăescu

Exemplul I

```
class Painter {  
    public void drawAll(Object[] figs) {  
        for(Object aFig : figs) {  
            if(aFig instanceof Circle) {  
                ((Circle)aFig).drawCircle();  
            } else if(aFig instanceof Square) {  
                ((Square)aFig).drawSquare();  
            } else {  
                ((Triangle)aFig).drawTriangle();  
            }  
        }  
    }  
}
```

Un program ce lucrează cu
figuri geometrice

Circle	Square	Triangle
... +drawCircle() : void	... +drawSquare() : void	... +drawTriangle() : void

1. Adăugăm clasa corespunzătoare
2. Peste tot unde am făcut distincție între diverse feluri de figuri, mai adăugăm probabil un **if-instanceof-else**

După un timp vrem să extindem programul cu triunghiuri

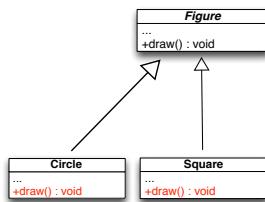
Painter nu respectă principiul pt.
că trebuie să-l modificăm codul

Dr. Petru Florin Mihăescu

Exemplul I

Un program ce lucrează cu
figuri geometrice

```
class Painter {  
    public void drawAll(figure[] figs) {  
        for(figure aFig : figs){  
            aFig.draw();  
        }  
    }  
}
```



După un timp vrem să extindem programul
cu triunghiuri

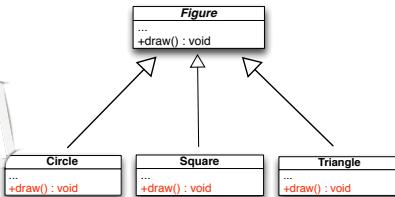
Dr. Petru Florin Mihăescu

Exemplul I

Un program ce lucrează cu
figuri geometrice

```
class Painter {  
    public void drawAll(figure[] figs) {  
        for(figure aFig : figs) {  
            aFig.draw();  
        }  
    }  
}
```

Desenează corect și triunghiuri; în
general legarea dinamică apelează
implementarea corespunzătoare a
operatiei draw



După un timp vrem să extindem programul
cu triunghiuri

I. Adăugăm subclasa corespunzătoare

Si gata!

Painter respectă principiul

Dr. Petru Florin Mihăescu

Exemplul 2

```
class Ceasornicar {
    public void regleaza(Object x) {
        if(x instanceof Clock) {
            ((Clock) x).setTime(12, 0, 0);
        } else if (x instanceof EnhancedClock) {
            ((EnhancedClock) x).setTime(12, 0, 0);
        }
    }
}
```

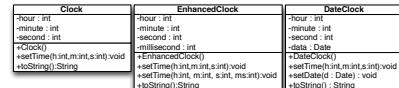
Ceasurile și ceasornicarul ...

Clock	EnhancedClock
-hour : int -minute : int -second : int +Clock() +setTime(int, int, int, int) void +toString() String	-hour : int -minute : int -second : int -milliseconds : int +EnhancedClock() +setTime(int, int, int, int) void +setTime(int, int, int, int, int) void +toString() String

```
class Ceasornar {
    public void regleaza(Obiect x) {
        if(x instanceof Clock) {
            ((Clock) x).setTime(12, 0, 0);
        } else if (x instanceof EnhancedClock) {
            ((EnhancedClock) x).setTime(12, 0, 0);
        } else if (x instanceof DateClock) {
            ((DateClock) x).setTime(12, 0, 0);
        }
    }
}
```

Exemplul 2

Ceasurile și ceasornicarul ...



```
class Ceasornicar {
    public void regleaza(Object x) {
        if(x instanceof Clock) {
            ((Clock)x).setTime(12, 0, 0);
        } else if (x instanceof EnhancedClock) {
            ((EnhancedClock)x).setTime(12, 0, 0);
        } else if (x instanceof DateClock) {
            ((DateClock)x).setTime(12, 0, 0);
        } else if (x instanceof SmallClock) {
            ((SmallClock)x).setTime(12, 0, 0);
        }
    }
}
```

Exemplul 2

Ceasurile și ceasornicarul ...

Clock	EnhancedClock	DateClock	SmallClock
-hour : int -minute : int -second : int +Clock() +setTime(int,m,int,s,int);void +toString();String	-hour : int -minute : int -second : int +EnhancedClock() +setTime(int,m,int,s,int);void +setTime(int,m,int,s,int);void +setTime(int,m,int,s,int);void +setTime(int,m,int,s,int);void	-hour : int -minute : int -second : int +DateClock() +setTime(int,m,int,s,int);void +setTime(int,m,int,s,int);void	-time : int +setTime(int,m,int,s,int);void +toString();String

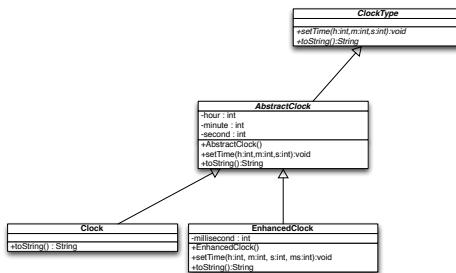
Ceasornicarul nu respectă
principiul pt. că trebuie să-i
modificăm codul + multă
duplicare de cod

Dr. Petru Florin Milăneș

Exemplul 2

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

Ceasurile și ceasornicarul ...

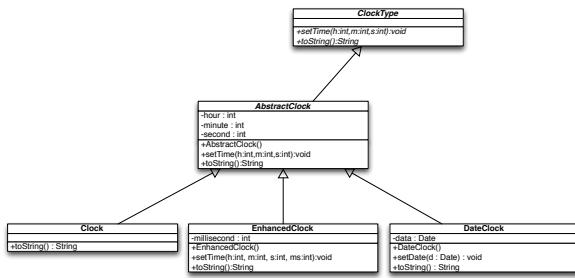


Dr. Petru Florin Mihăescu

Exemplul 2

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

Ceasurile și ceasornicarul ...



Dr. Petru Florin Mihăescu

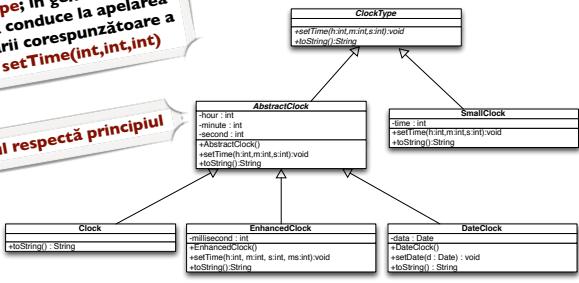
Exemplul 2

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

Ceasurile și ceasornicarul ...

Regleză orice ceas ce aderă la tipul ClockType; în general legarea dinamică va conduce la apelarea implementării corespunzătoare a operației setTime(int,int,int)

Ceasornicarul respectă principiul



Dr. Petru Florin Mihăescu

Quizz

Spre ce fel de obiecte poate referii o intrare din tabloul următor ?

ClockType[] t = new ClockType[10];

... dar tabloul următor ?

ClockType[] t = new Clock[10];

- 
1. Orice obiect de tip **ClockType**
 2. Numai obiecte **Clock**
(la compilare aparent ok, dar la rulare eroare de execuție dacă e altfel)

D

Altă variantă pt. declarare și implementare de tipuri

Dr. Petru Florin Mihai

Cum “declarăm” un tip ?

... fără niciun fel de detaliu de implementare

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

... sau folosim conceptul de
interfață din Java

De la Java 8 este
puțin altfel
(nu folosiți) !

... altă variantă

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

```
<<interface>>  
ClockType  
+setTime(h : int, m : int, s : int) : void  
+toString() : String
```

Automat - fie că le declarăm explicit așa, fie că nu
toate metodele sunt **abstrakte**
toate metodele sunt **publice**
orice câmp este **public static final**
nu pot fi instantiată

Seamană foarte mult cu o
clasă pur abstractă

Dacă încercăm altfel, eroare de
compilare

implements

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

```
<<interface>>  
ClockType  
+setTime(h : int, m : int, s : int) : void  
+toString() : String
```

```
class SmallClock implements ClockType {  
    private int time;  
    public void setTime(int h, int m, int s) {  
        if(h >= 0 && h < 24) {  
            time = time & 0x0000FFFF; time = time | (h << 16);  
        }  
        if(m >= 0 && m < 60) {  
            time = time & 0xFFFF00FF; time = time | (m << 8);  
        }  
        if(s >= 0 && s < 60) {  
            time = time & 0x00FF0000; time = time | s;  
        }  
    }  
    public String toString() {  
        return "SmallClock - " + ((time & 0xFFFF0000) >> 16) + ":"  
            + ((time & 0x0000FF00) >> 8) + ":"  
            + (time & 0x000000FF);  
    }  
}
```

```
SmallClock  
-time : int  
+setTime(h:int,m:int,s:int):void  
+toString():String
```

Denotă și moștenirea tipului
deci nu trebuie să ne mirăm că o
referință de tipul interfeței va
putea referi obiecte a oricărui
clase ce implementează acea
interfață

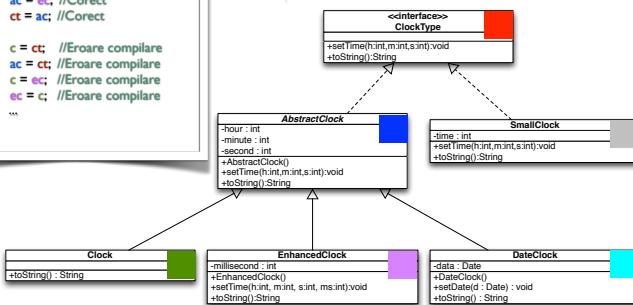
```

class Main {
    public static void main(String[] args) {
        ClockType ct;
        AbstractClock ac;
        Clock c;
        EnhancedClock ec;
        c = new Clock();
        ...
        ct = c; //Eroare compilare
        ac = c; //Eroare compilare
        c = ec; //Eroare compilare
        ec = c; //Eroare compilare
        ...
    }
}

```

Ceasurile

Polimorfism, legare dinamică,
verificarea apelurilor, downcast
ul sunt la fel ca în versiunea
anterioară



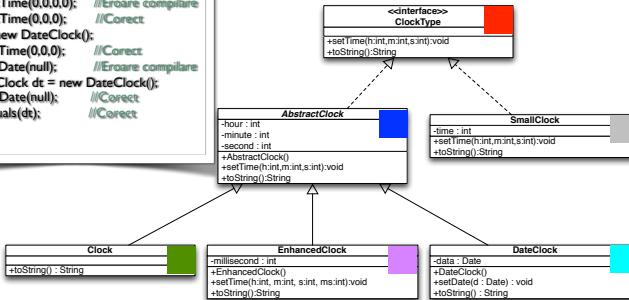
```

class Main {
    public static void main(String[] args) {
        ClockType ct;
        AbstractClock ac;
        Clock c;
        EnhancedClock ec;
        ...
        ct.setTime(0.0); //Correct
        ac.setTime(0.0); //Correct
        c.setTime(0.0); //Correct
        ec.setTime(0.0); //Correct
        ec.setTime(0.0,0); //Correct
        ac = new EnhancedClock();
        ac.setTime(0.0,0); //Eroare compilare
        ac.setTime(0.0); //Correct
        ct = new DateClock();
        ct.setTime(0.0); //Correct
        ct.setDate(null); //Eroare compilare
        DateClock dc = new DateClock();
        dc.setDate(null); //Correct
        dc.equals(dt); //Correct
    }
}

```

Ceasurile

Polimorfism, legare dinamică,
verificarea apelurilor, downcast
ul sunt la fel ca în versiunea
anterioară

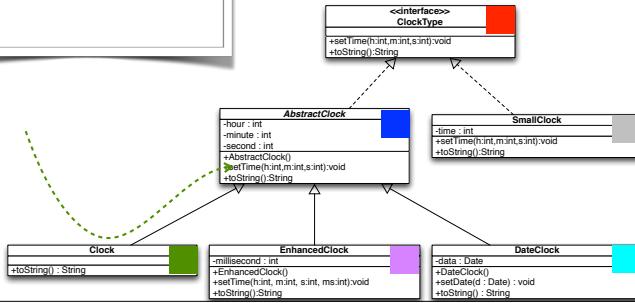


```
class Ceasnicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
Ceasnicar om = new Ceasnicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
```

Ceasurile

Polimorfism, legare dinamică,
verificarea apelurilor, downcast
ul sunt la fel ca în versiunea
anterioară

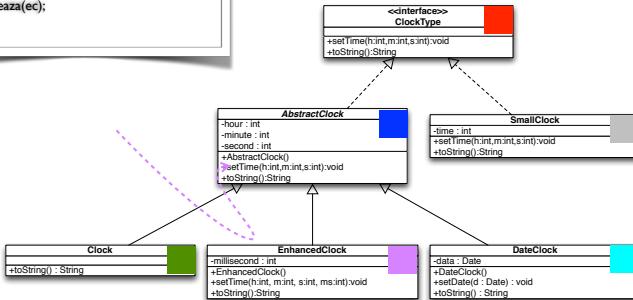


```
class Ceasnicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
Ceasnicar om = new Ceasnicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
om.regleaza(ec);
```

Ceasurile

Polimorfism, legare dinamică,
verificarea apelurilor, downcast
ul sunt la fel ca în versiunea
anterioară

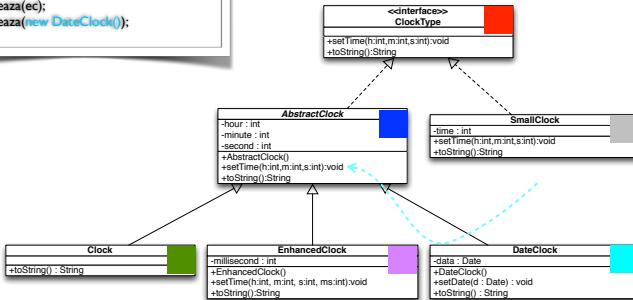


```
class Ceasnicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
Ceasnicar om = new Ceasnicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
om.regleaza(ec);
om.regleaza(new DateClock());
```

Ceasurile

Polimorfism, legare dinamică,
verificarea apelurilor, downcast
ul sunt la fel ca în versiunea
anterioară

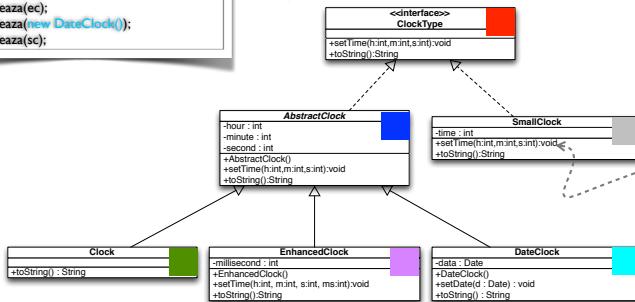


```
class Ceasnicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
Ceasnicar om = new Ceasnicar();
Clock c = new Clock();
EnhancedClock ec = new EnhancedClock();
SmallClock sc = new SmallClock();
om.regleaza(c);
om.regleaza(ec);
om.regleaza(new DateClock());
om.regleaza(sc);
```

Ceasurile

Polimorfism, legare dinamică,
verificarea apelurilor, downcast
ul sunt la fel ca în versiunea
anterioară



Ceva problema?

Quizz

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

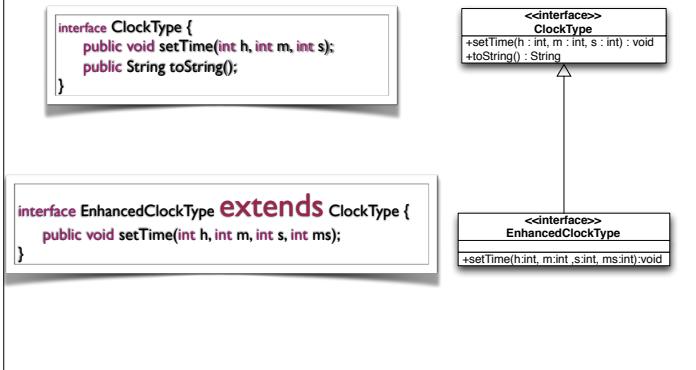
class SomeClock implements ClockType {
    private int h,m,s;

    public SomeClock(int h, int m, int s) {
        this.h = h;
        this.m = m;
        this.s = s;
    }

    public String toString() {
        return "Some Clock - Current time " + h + ":" + m + ":" + s;
    }
}
```

SomeClock trebuie să fie abstractă
(pt. că nu implementează setTime)

extends între interfețe



chiar și mai multe ...

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

```
<<interface>>  
ClockType  
+setTime(h : int, m : int, s : int) : void  
+toString() : String
```

```
interface RadioType {  
    public void startPlay();  
    public void stopPlay();  
    public void searchNext();  
}
```

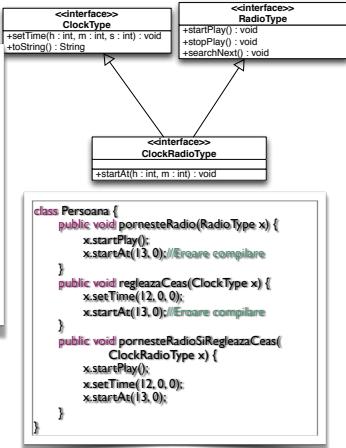
```
<<interface>>  
RadioType  
+startPlay() : void  
+stopPlay() : void  
+searchNext() : void
```

```
interface ClockRadioType extends ClockType, RadioType {  
    public void startAt(int h, int m);  
}
```

Putem modela tipurile/
subtipurile (inclusiv putem
combi într-un subtip două
tipuri distincte)

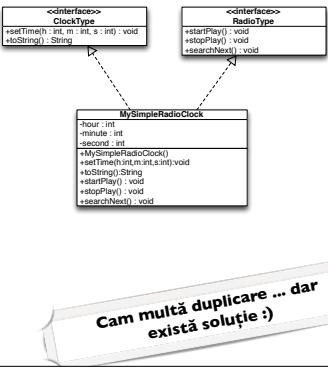
Atenție la ce putem apela

```
class Main {  
    public static void main(String[] args) {  
        ClockRadioType rct;  
        RadioType rt;  
        ClockType ct;  
        ...  
        Persoana om = new Persoana();  
        om.pornesteRadio(rct);  
        om.pornesteRadio(rt);  
        om.pornesteRadio(ct); //EROARE compilare  
        om.regleazaCeaș(c);  
        om.regleazaCeaș(rt); //EROARE compilare  
        om.porneșteRadioSiRegleazăCeaș(rct); //EROARE compilare  
        om.porneșteRadioSiRegleazăCeaș(ct); //EROARE compilare  
        om.porneșteRadioSiRegleazăCeaș(rt); //EROARE compilare  
    }  
}
```



implements again

```
class MySimpleRadioClock implements  
    ClockType, RadioType {  
  
    private int hour, minute, second;  
    public MySimpleRadioClock() {  
        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 "MySimpleRadioClock - Current time " +  
            hour + ":" + minute + ":" + second;  
    }  
    public void startPlay() {  
        System.out.println("Start");  
    }  
    public void stopPlay() {  
        System.out.println("Stop");  
    }  
    public void searchNext() {  
        System.out.println("Search");  
    }  
}
```

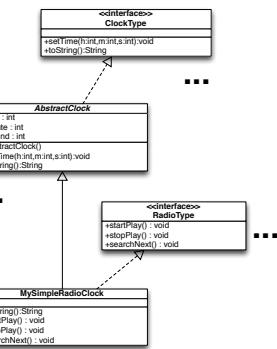


Cam multă duplicare ... dar
există soluție :)

implements again

```
class MySimpleRadioClock extends AbstractClock
    implements RadioType {
    public String toString() {
        return "MySimpleRadioClock - " + super.toString();
    }
    public void startPlay() {
        System.out.println("Start");
    }
    public void stopPlay() {
        System.out.println("Stop");
    }
    public void searchNext() {
        System.out.println("Search");
    }
}
```

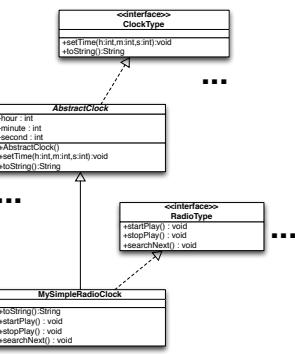
în general, o clasă poate extinde
o clasă și poate implementa mai
multe interfețe



Putem face asta !

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

```
Ceasornicar om = new Ceasornicar();  
MySimpleRadioClock rc = new MySimpleRadioClock();  
om.regleaza(rc);
```



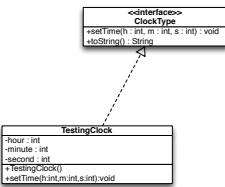
... și evident, oriunde e nevoie
de un obiect `RadioType` putem
folosi un `MySimpleRadioClock`

De ce o fi ok asta ?

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

```
class TestingClock implements ClockType {  
    private int h,m,s;  
    public void setTime(int h, int m, int s) {  
        this.h = h;  
        this.m = m;  
        this.s = s;  
    }  
}
```

Nu există implementare în
TestingClock pt. toString și
totuși compilează

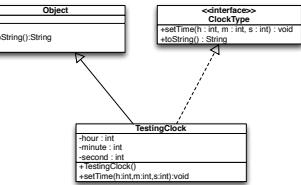


De ce o fi ok asta ?

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

```
class TestingClock implements ClockType {  
    private int h, m, s;  
    public void setTime(int h, int m, int s) {  
        this.h = h;  
        this.m = m;  
        this.s = s;  
    }  
}
```

Nu există implementare în
TestingClock pt. `toString` și
totuși compilează



În esență, implementarea unei
metode dintr-o interfață poate
veni și de la o superclăsă pe care
clasa o extinde
(e considerat overriding)

Potențiale probleme

```
interface A {  
    public void get();  
}  
  
interface B {  
    public int get();  
}  
  
interface AB extends A, B {  
    //Eroare compilare deoarece metodele  
    //diferă doar prin tipul returnat  
}
```

```
interface A {  
    public void get();  
}  
  
abstract class AbstractA {  
    public int get() {return 0;}  
}  
  
class ImplementationA extends AbstractA implements A {  
    //Eroare compilare deoarece metodele  
    //diferă doar prin tipul returnat  
}
```

Ar fi OK dacă tipul returnat al
unei declarații de metodă e
subtip al tipului returnat de
ceață declaratie
(permis de overriding)

clase abstracte vs. interfețe

Clasă abstractă

- Putem **mixa** definirea tipului cu factorizarea codului comun
- O clasă poate extinde o singură altă clasă (în Java)



Interfață

- **Nu** putem factoriza codul comun diverselor implementări
 - ... dar am putea să-l **factorizăm** într-o clasă intermediară ori "laterală"
- O clasă poate implementa mai multe interfețe
- O interfață poate extinde mai multe interfețe

Quizz

Poate extinde o interfață o clasă ?

Poate implementa o interfață o clasă ?



Quizz

```
interface A {  
    String CONST = "A";  
}  
  
interface B {  
    String CONST = "B";  
}  
  
class ClassAB implements A,B {  
    public String toString() {  
        return CONST;  
    }  
}
```

Situatie de ambiguitate! Trebuie
mentionat explicit A.CONST
sau B.CONST!