

Relația de Moștenire

Dr. Petru Florin Mihancea

V20180924

Moștenirea



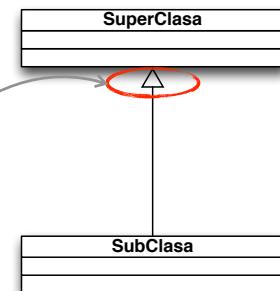
Booch - OO Analysis and Design

... 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 {  
}  
...
```



Atenție la
capătul săgeții!

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

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 este extrem de dăunătoare (e.g., copiezi și bug-uri, dacă e necesară o modificare trebuie să modifici 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;
    }
}

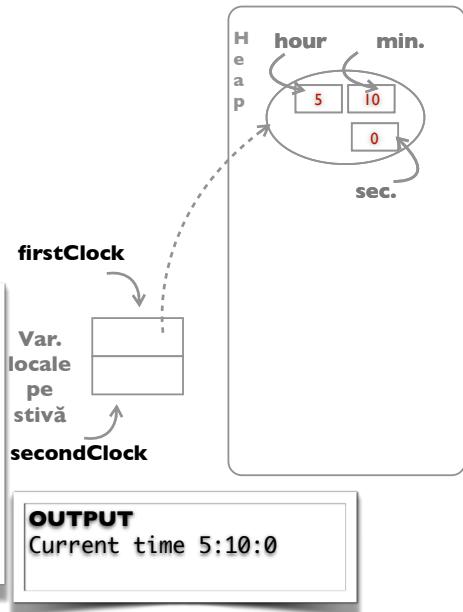
```

```

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

```

Dr. Petru Florin Mihăeș



```

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;
    }
}

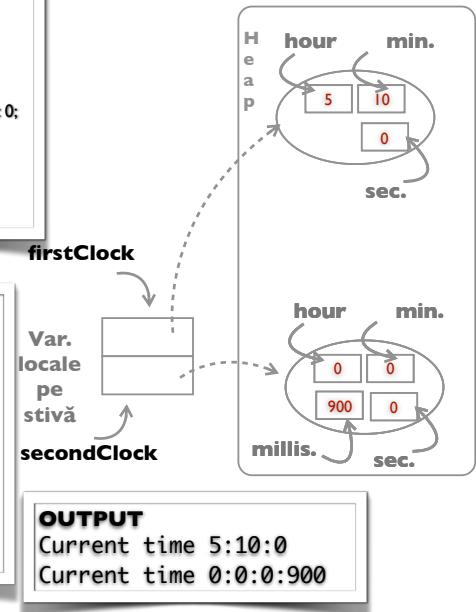
```

```

class Main {
    public static void main(String argv[]) {
        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, 0, 900);
        System.out.println(secondClock);
    }
}

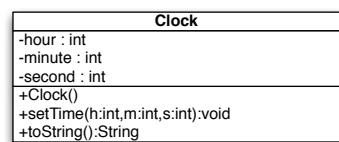
```

Dr. Petru Florin Mihăeș

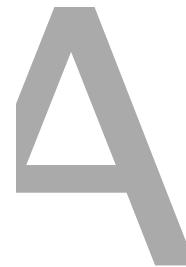
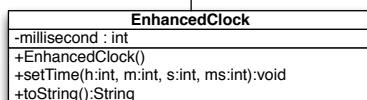


Mostenirea (de clasă)

Un mecanism simplu de reutilizare de cod :)



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



Specificatori de acces,
constructori, etc. în contextul
moștenirii

Dr. Petru Florin Mihăeș

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)

Vizibilitatea în UML

- private
- + public
- # protected

Dr. Petru Florin Mihăncă

Uzual, sunt folositi pt. a aduce un obiect nou creat în starea inițială (atribuirea unor valori initiale la variabilele instanțăi)

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

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;  
}
```

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;  
    }  
    ...  
}
```

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

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

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

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 superclăsa 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 :

Ordinea inițializărilor (aprox)

```
class A {  
    protected int z = 1;  
}
```

se repetă ca pt. subclasă

```
...  
class B extends A {  
    private int x = z;  
    public B() {  
        super();  
        z = 3;  
        x = 2;  
    }  
    public void print() {  
        System.out.println(  
            "X=" + x + " Y=" + y + " Z=" + z);  
    }  
    private int y = x;  
}
```

Dr. Petru Florin Mihăescu

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

x = 2
y = 1
z = 3

Oare unde se introduce bytecode-ul initializărilor variabilelor de stare ?

B

Redefinirea metodelor

(method overriding)

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

modificatorii 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

În capitolile anterioare, când reimplementam equals, toString ... din clasa Object făceam overriding

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

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;  
    }  
}
```

suprascriere / redefinire / overriding același nume, aceeași semnătură

```
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;  
    }  
}
```

supraîncărcare (overloading) același nume, semnături diferite

Putem controla :)

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

Este o eroare de compilare dacă cineva încearcă să redefinească metoda

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

Este o eroare de compilare dacă cineva extinde clasa ClassName



Ascunderea (hiding) variabilelor instantă

Ascunderea var. instanță

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 aceeași semnătură	Metodă instanță superclasă	Metodă statică superclasă
Metodă instanță subclasă	Overriding / Redefinire	Eroare compilare
Metodă statică subclasă	Eroare compilare	Hiding / Ascundere

Metodele instanță pot fi doar redefinite !!!

ascunse (NU pot fi redefinite și implicit mecanismele bazate pe overriding NU merg la metodele statice)

D

Când folosim ?
Moștenirea de clasă vs. compunerea obiectelor
(compunere - object composition)

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 Mihancă

Studiu de caz

Vector

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



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 e asta pt. notiunea de stiva ?
    stk.removeElementAt(0);
    // adica cum (????) ca intr-o stiva trebuie sa pot
    // accesa doar ultimul element introdus
}
```

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



Stack

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

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
```



Studiu de caz

"Simulează" un tablou a căruia 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;
    }
    ...
}
```

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
```

Main

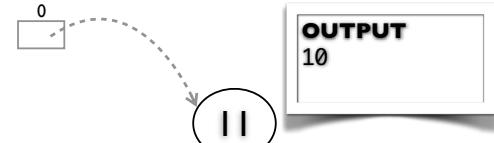
```
Stack stk = new Stack();
stk.push(new Integer(5));
stk.push(new Integer(10));
Object p = stk.pop();
System.out.println(p);

// cu add (!!!!) ca intr-o stiva trebuie sa pot
// accesa doar ultimul element introdus
}
```

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

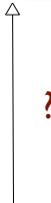
Interviu (!!!!) că într-o stivă trebuie să pot accesa doar **ultimul element introdus**

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



Asta NU e soluția ...

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

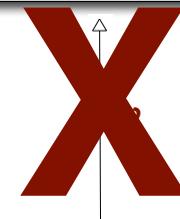


```
Stack
...
+push(o:Object) : Object
+pop() : Object
+peek() : Object
+empty() : boolean
```

```
class 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 stiva nu stie asta!");
    }
}
```

```
class Main {
    public static void main(String args[]) {
        Stack stk = new Stack();
        ...
        stk.removeElementAt(0); //pot apela dar nu va face nimic !?
        ...
    }
}
```

```
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
```

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 stiva nu stie asta!");
    }
}
```



Un obiect/clasă pune împreună datele și operațiile ce operează pe acele date ... iar metoda noastră nu face nimic deci ce caută în interfața obiectului ?

Componerea obiectelor

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

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

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

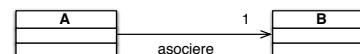
Uzual spune că o mașină are un motor.
Componerea este o relație de tip has-a

Reprezentarea în diagrame UML de clase

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

Multiplicitate
arată câte obiecte B are un A

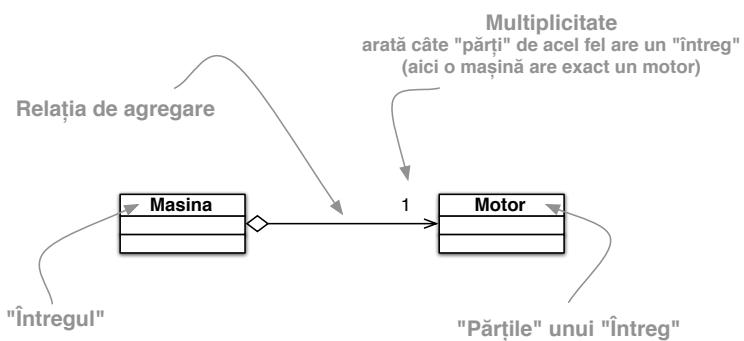
alte variante uzuale
0..1 - zero sau cel mult unul
0..* - zero sau oricără de multe



agregare: un întreg (A) este compus din părți (B)

compoziție (ca noțiune UML) : un A are B-uri (pe vecie) numai și numai a lui
(a nu se confunda cu termenul obiect composition / componerea)

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

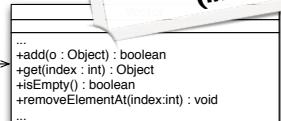
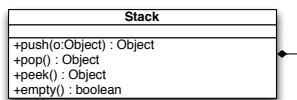
Prin urmare
e un alt mod de
reutilizare de cod

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 compilare  
    }  
}
```

... în plus putem schimba
Vectorul cu altceva fără să
afectăm clienții 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

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

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

M-ar deranja dacă eu v-ăs 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!

```
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

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;  
    }  
}
```

```
class EnhancedClock extends Clock {  
    private int millisecond;  
    public EnhancedClock() {  
        millisecond = 0;  
    }  
    public void setTime(int h, int m, int s, int ms) {  
        Se spune că EnhancedClock este un  
        subtip de-al lui Clock și deci  
        moștenirea e folosită aici nu numai ca  
        moștenire de clasă. Hai să vedem  
        moștenirea de tip :)  
        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 v-ăs 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!

2

Moștenirea de de tip / interfață și polimorfismul

(de subtip)

Notiunea de tip

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

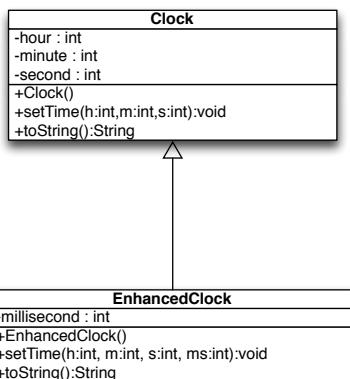
Barbara Liskov

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**)

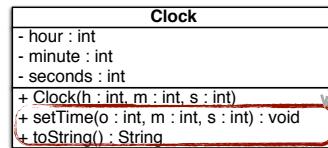


În alte limbi se poate controla mai fin semantica lui extends

extends

între clase exprimă **și**, moștenirea tipului

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



Tipuri vs. Clase

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

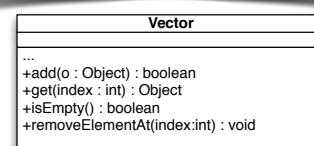
Conceptual clasa diferă de **noțiunea de tip**

- 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

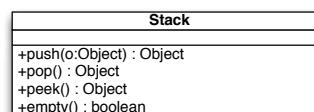
Să ne amintim ...

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



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

```
stk.add(new Integer(11));
// cu add (?) ce operație 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, încalcând prima heuristică :(

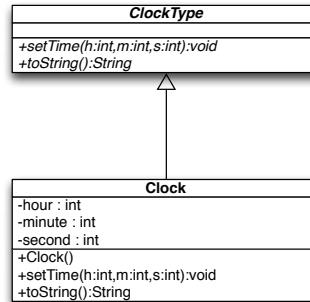
A

Clase și metode abstracte

(ce, de ce și pentru ce ?)

Dr. Petru Florin Mihăneanu

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



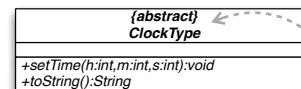
```
class Clock extends ClockType {
    private int hour, minute, second;
    public Clock() {
        hour = minute = second = 0;
    }
    public void setTime(int h, int m, int s) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m > 0) && (m < 60) ? m : 0;
        second = (s >= 0)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

Cum “declarăm” un tip ?

... fără niciun fel de detaliu de implementare

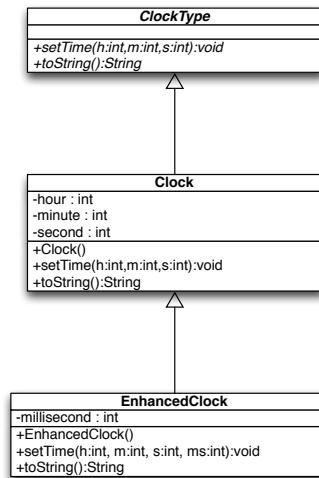
```
abstract class ClockType {
    public abstract void setTime(int h, int m, int s);
    public abstract String toString();
}
```



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

```
class Main {
    public static void main(String args[]) {
        //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 ?
    }
}
```

Exemplul cu ceasurile ...



Varianta I (însă ...)

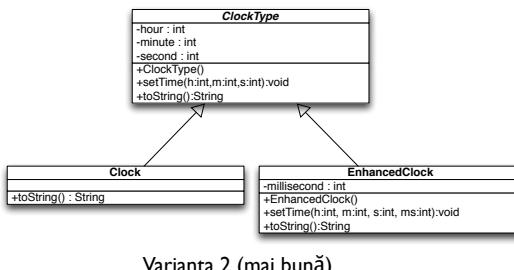
Toate clasele de bază ar trebui să fie abstracte

Schimbăm puțin : **toString** pune în față string-ului returnat și felul ceasului (în general ceva specific felului de ceas)

Cumva EnhancedClock să moștenească direct ClockType (normal din moment ce e o altă implementare a tipului ClockType) dar nici să nu duplicăm ce e comun. Oare cum facem ?

Riel 5.7

Exemplul cu ceasurile ... (II)

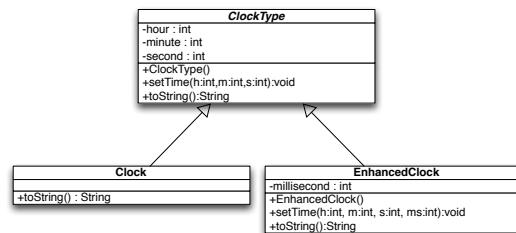


```

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 abstracte (și poate avea câmpuri, constructori pt. a-i inițializa, etc.). Prin urmare poate și factoriza codul comun.

Exemplul cu ceasurile ... (II)

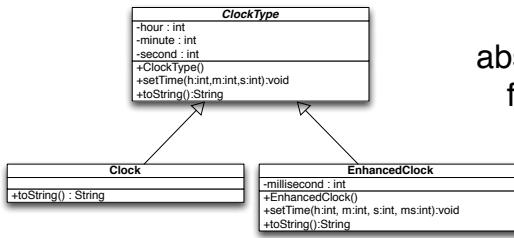


```

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;
    }
}
  
```

Obs. - o clasă poate fi declarată **abstractă** și dacă nu are metode abstracte (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

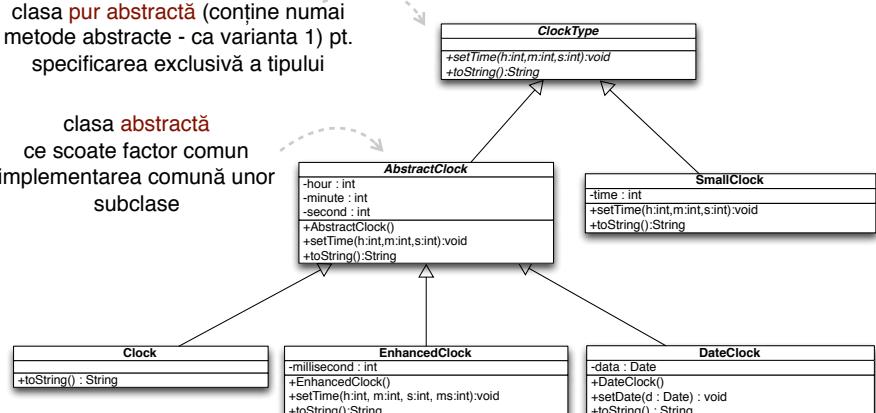
Dacă apar noi implementări ale tipului **ClockType** unde le punem în ierarhie ?
ex. **SmallClock** care codifică ora, minutul și secunda în octeți separați ai aceluiasi int ?

Nasol,
dar se poate mai bine :)

Exemplul cu ceasurile ... (IV)

clasa **pur abstractă** (contine 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 declaratia metodei, specificand o operatie a unui tip
- sunt implementate prin overriding in 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 faceti-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 Mihăncea

Efectul moștenirii de tip

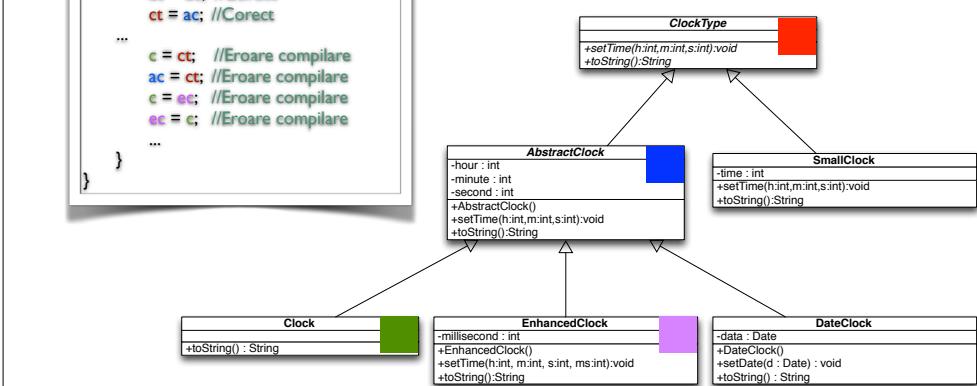
Polimorfism

O variabilă referință declarată de un anumit tip (clasa) poate să refere obiecte a aceluui 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; //Corect  
        ct = ec; //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 (clasa) poate să refere obiecte a aceluui tip (clase) și a oricărui alt subtip (subclase) de-al său



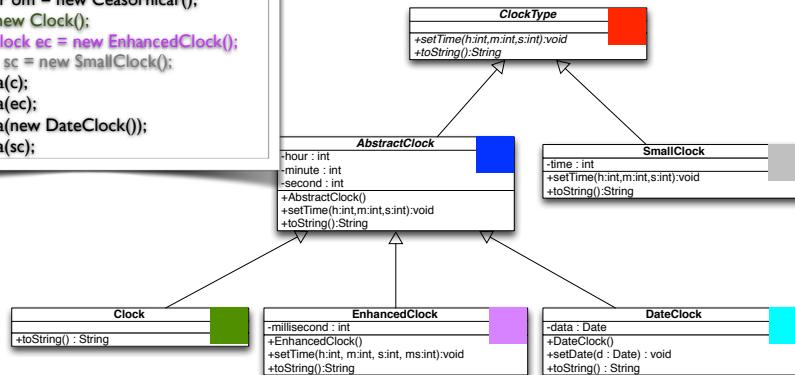
Nu știm exact: poate fi Clock,
EnhancedClock, DateClock sau
SmallClock

```
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 ?



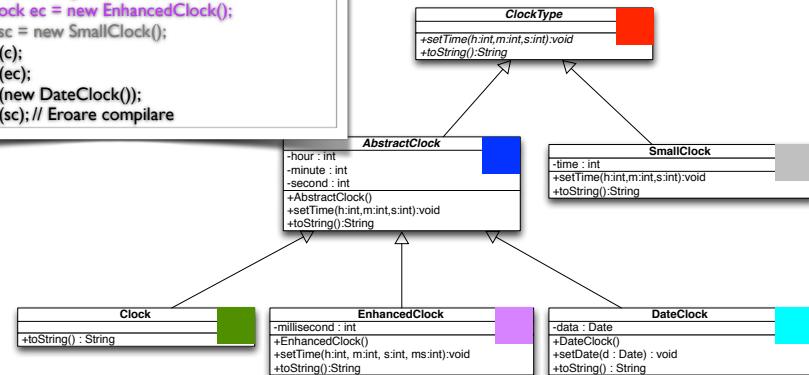
Nu știm exact: poate fi Clock,
EnhancedClock sau DateClock

```
class CeasornicarMaiSlabut {
    public void regleaza(AbstractClock x) {
        ...
    }
}
```

```
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
```

Întrebare

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



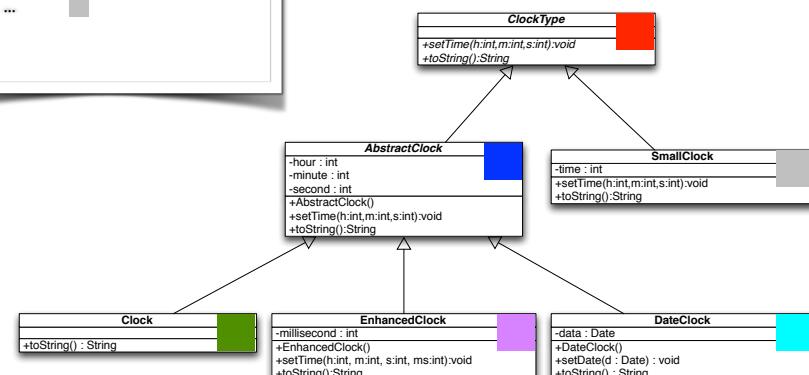
Spre ce fel concret de obiecte poate referi o referință declarată ca fiind de tip Object ?

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

Quizz

```
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



instanceof

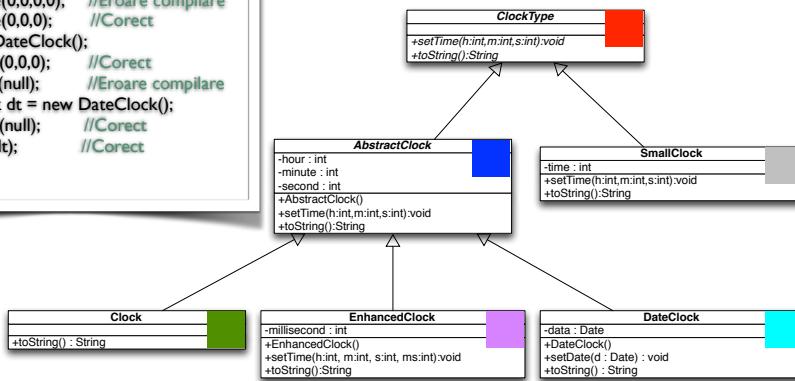
```

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

```

Invocare corectă (la compilare)

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



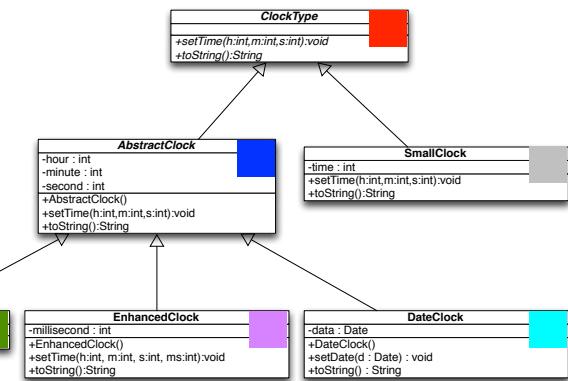
```

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



```

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

```

```

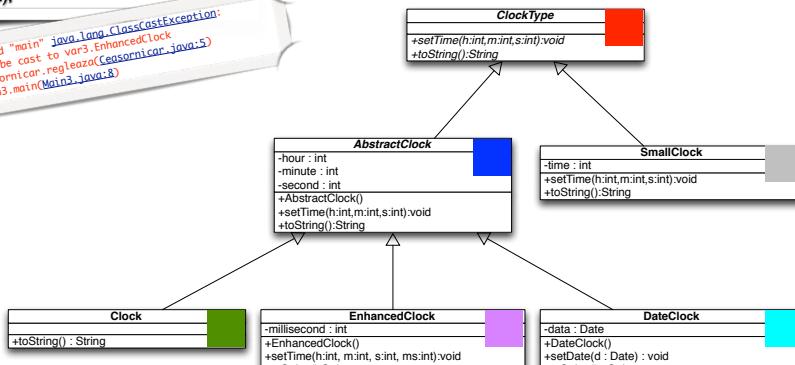
Ceasornicar om = new Ceasornicar();
Clock c = new Clock();
om.regleaza(c);

```

Exception in thread "main" java.lang.ClassCastException:
var3.Clock cannot be cast to var3.EnhancedClock
at var3.Ceasornicar.regleaza(Ceasornicar.java:5)
at var3.Main.main(Main.java:8)

(Down) Cast

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) {
        if(x instanceof EnhancedClock) {
            ((EnhancedClock)x).setTime(12,0,0);
        }
    }
}

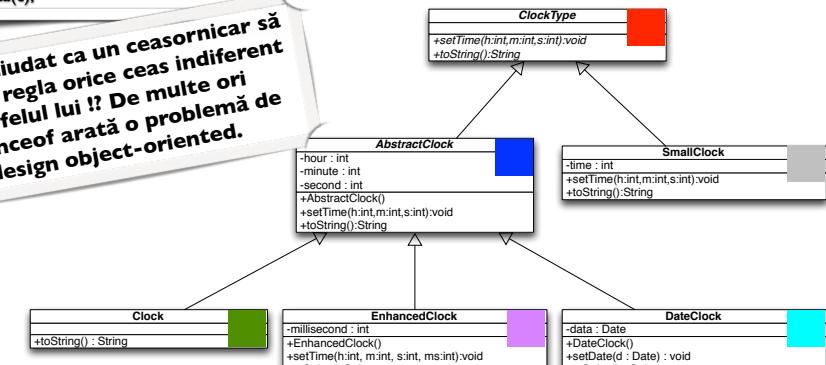
```

```

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

```

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



(Down) Cast

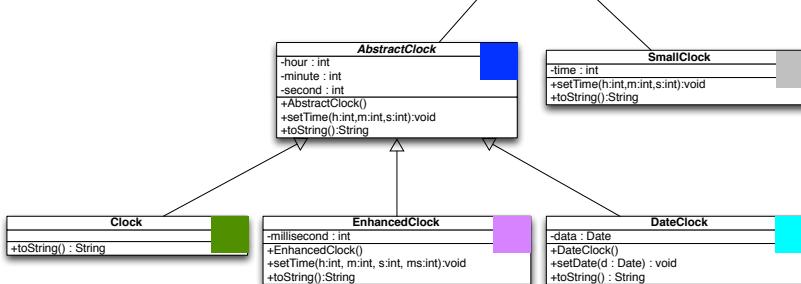
Necesar dacă trebuie să apelăm metode specifice unui subtip dar avem referințe de un supertip de-al său

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

Legarea dinamică

Care implementare ?

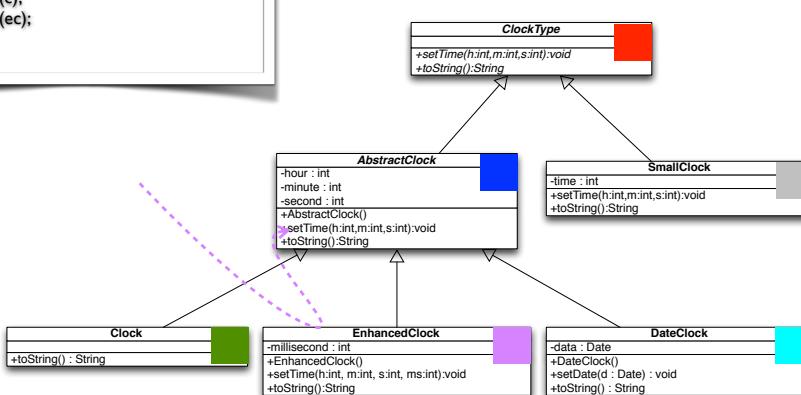
La compilare (static) nu putem stabili



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

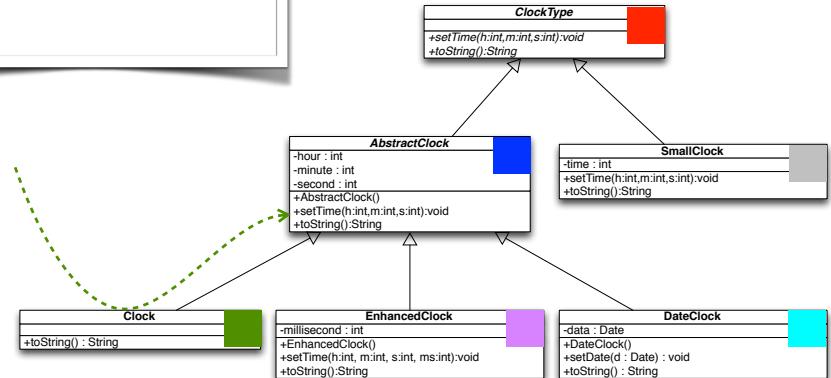


```
class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

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

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

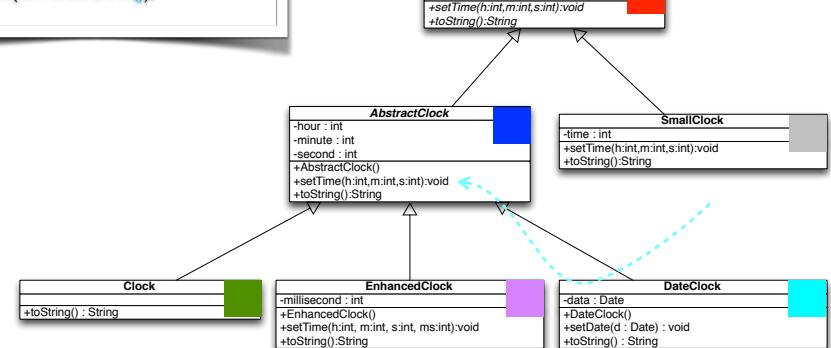


```
class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

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

```
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());
```

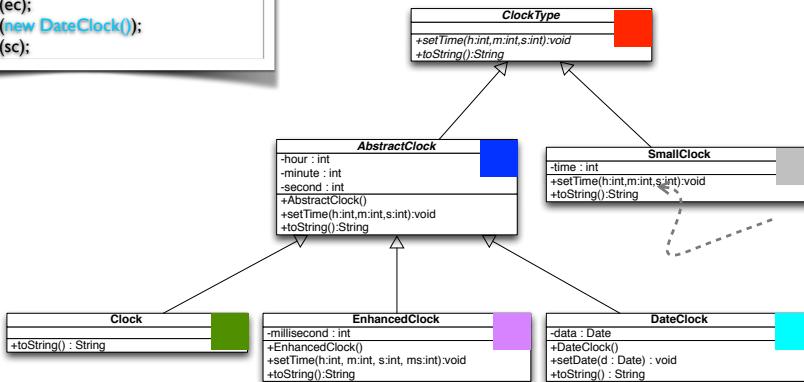


```
class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
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 instantă, 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

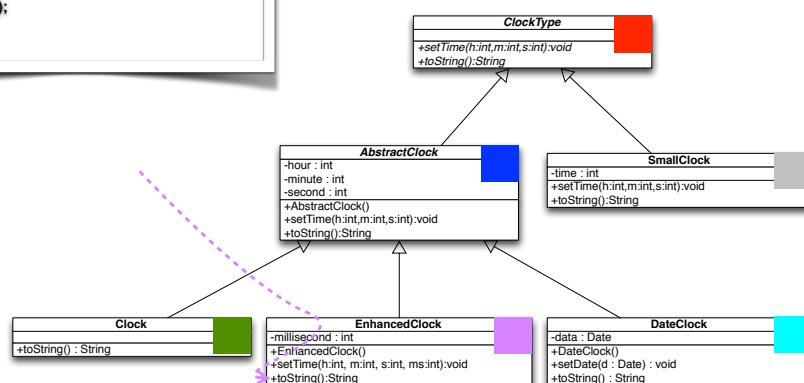


```
class Ceasornicar {
    public void print(ClockType x) {
        System.out.println(x.toString());
    }
}
```

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

Legarea dinamică

În cazul metodelor instantă, 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

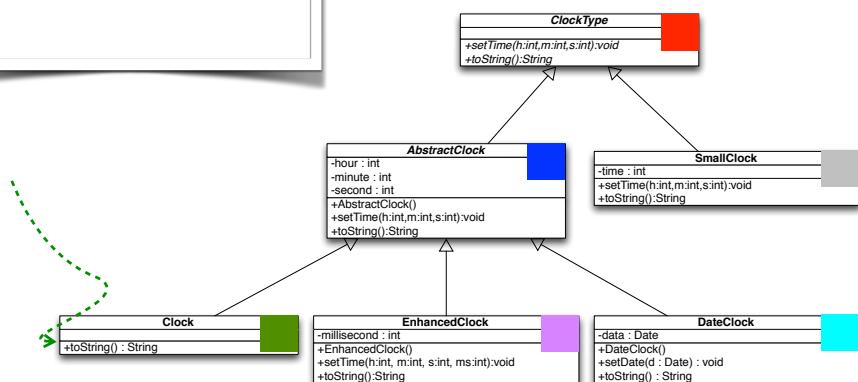


```
class Ceasornicar {
    public void print(ClockType x) {
        System.out.println(x.toString());
    }
}
```

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

Legarea dinamică

În cazul metodelor instantă, 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

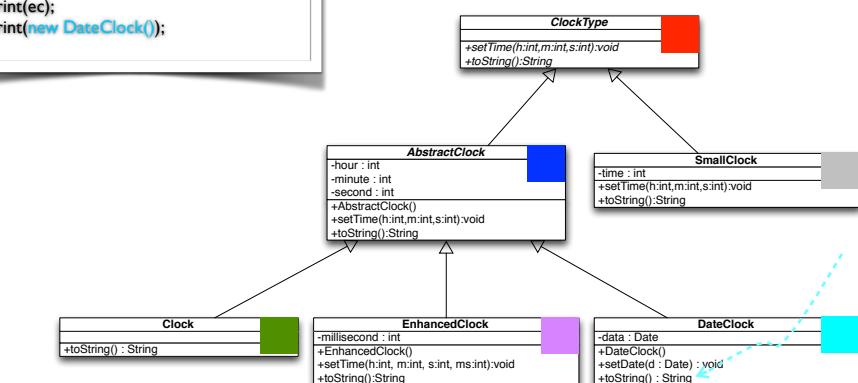


```
class Ceasornicar {
    public void print(ClockType x) {
        System.out.println(x.toString());
    }
}
```

```
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 instantă, 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

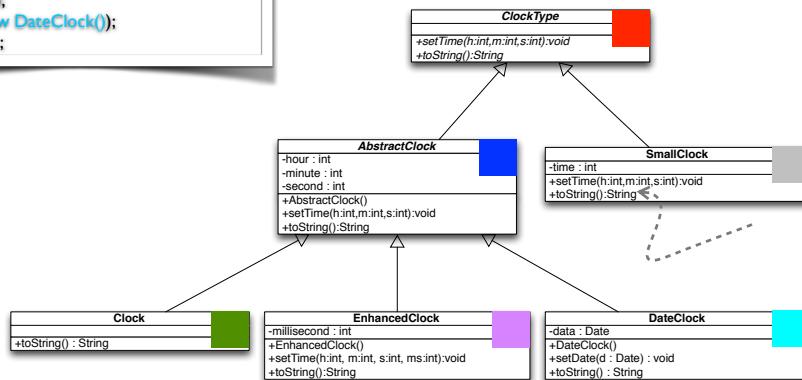


```
class Ceasornicar {
    public void print(ClockType x) {
        System.out.println(x.toString());
    }
}
```

```
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 instantă, 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**



Se aplică dacă e vorba de overriding

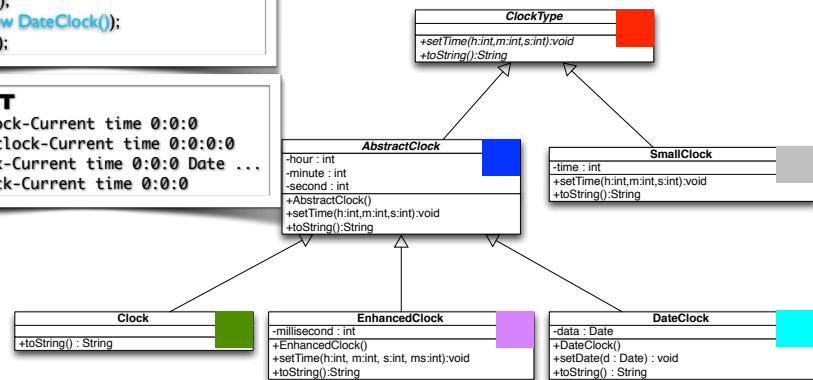
NU se aplică atunci când e vorba de hiding
deci niciodată la apeluri de metode statice
deci niciodată la accesarea de câmpuri

```
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
Date clock-Current time 0:0:0 Date ...
Small clock-Current time 0:0:0



Atenție ...

Quizz

Cât apare pe ecran ?

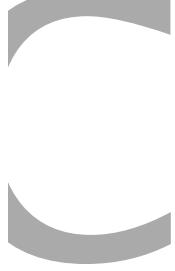
```
class A {
    protected int x = 0;
    public A() {
        x = doSomething();
    }
    public int doSomething() {
        return 10;
    }
    public String toString() {
        return "" + x;
    }
    public static void main(String argv[]) {
        System.out.println(new B());
        System.out.println(new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

class D extends C {
    public static int doSomethingElse() {
        return 20;
    }
}
```



De ce toate astea ?

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

Dr. Petru Florin Mihancea

Principiul Open-Closed

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

Bertrand Meyer
(restated by Robert Martin)

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

da, sigur ...



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 Mihancea

Exemplul I

Un program ce lucrează cu figuri geometrice

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

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

După un timp vrem să extindem programul cu triunghiuri

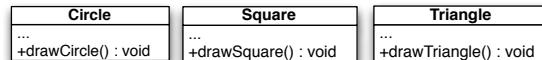
Dr. Petru Florin Mihancea

Exemplul I

Un program ce lucrează cu figuri geometrice

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

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



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-i modificăm codul

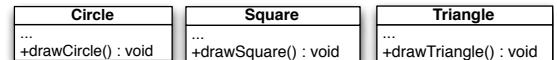
Dr. Petru Florin Mihăescu

După un timp vrem să extindem programul cu triunghiuri

Exemplul I

Un program ce lucrează cu figuri geometrice

```
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();  
            }  
        }  
    }  
}
```



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-i modificăm codul

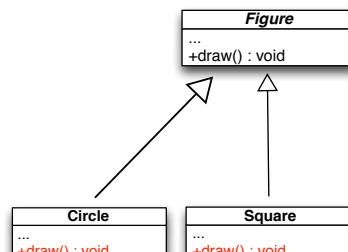
Dr. Petru Florin Mihăescu

După un timp vrem să extindem programul cu triunghiuri

Exemplul I

Un program ce lucrează cu figuri geometrice

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

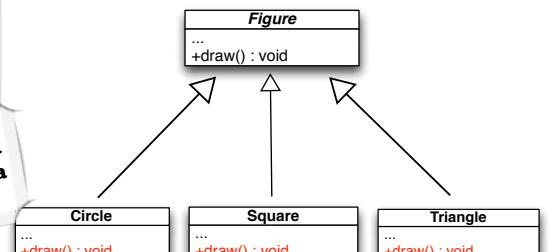


După un timp vrem să extindem programul cu triunghiuri

Dr. Petru Florin Mihăescu

```
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 operației draw



După un timp vrem să extindem programul cu triunghiuri

1. Adăugăm subclasa corespunzătoare
și gata!

Painter respectă principiul

Dr. Petru Florin Mihăescu

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);
        }
    }
}
```

Clock	EnhancedClock
-hour : int	-hour : int
-minute : int	-minute : int
-second : int	-second : int
+Clock()	+EnhancedClock()
+setTime(h:int,m:int,s:int);void	+setTime(h:int,m:int,s:int);void
+toString():String	+setTime(h:int,m:int,s:int,ms:int);void +toString():String

Dr. Petru Florin Mihăescu

```
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);
        }
    }
}
```

Exemplul 2

Ceasurile și ceasornicarul ...

Clock	EnhancedClock	DateClock
-hour : int	-hour : int	-hour : int
-minute : int	-minute : int	-minute : int
-second : int	-second : int	-second : int
+Clock()	+EnhancedClock()	+DateClock()
+setTime(h:int,m:int,s:int);void	+setTime(h:int,m:int,s:int);void	+setTime(h:int,m:int,s:int);void
+setTime(h:int,m:int,s:int,ms:int);void	+setTime(h:int,m:int,s:int,ms:int);void	+setDate(d : Date); void
+toString():String	+toString():String	+toString(): String

Dr. Petru Florin Mihăescu

```
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);
        }
    }
}
```

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

Dr. Petru Florin Mihăescu

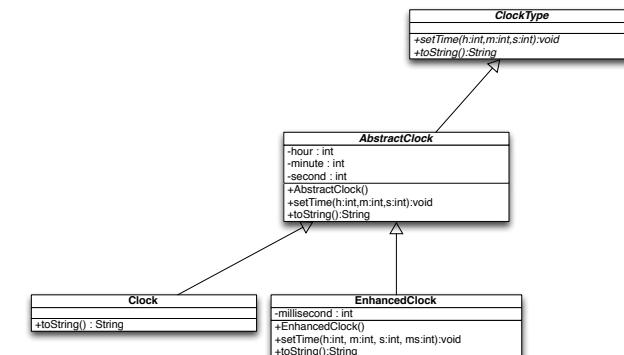
Exemplul 2

Ceasurile și ceasornicarul ...

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

Exemplul 2

Ceasurile și ceasornicarul ...



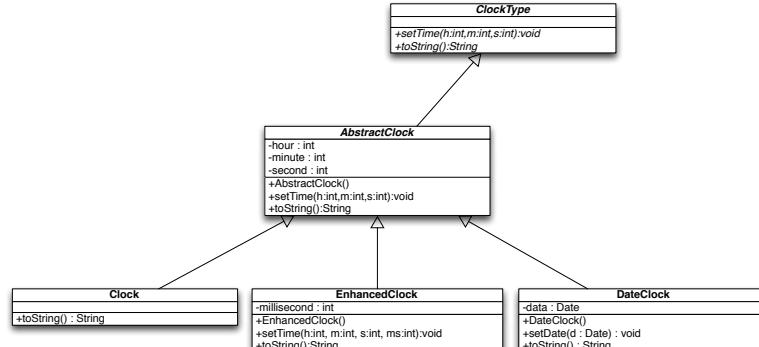
Ceasornicar nu respectă
principiul pt. că trebuie să-i
modificăm codul + multă
duplicare de cod

Dr. Petru Florin Mihăescu

Exemplul 2

Ceasurile și ceasornicarul ...

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



Dr. Petru Florin Mihancea

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)

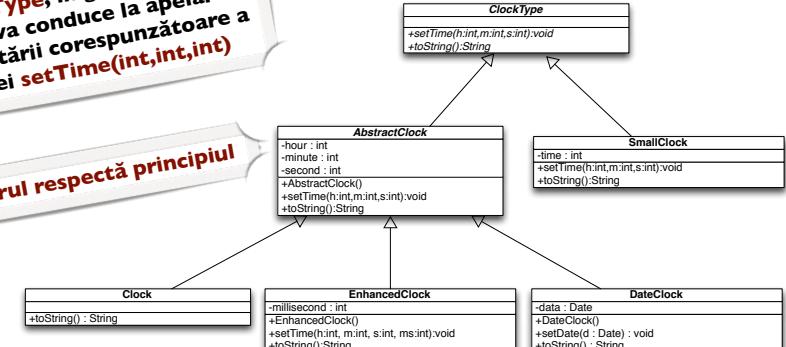
Exemplul 2

Ceasurile și ceasornicarul ...

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

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 Mihancea

D

Altă variantă pt. declarare și implementare de tipuri

Dr. Petru Florin Mihancea

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 instanțiate

Seamană foarte mult cu o clasă **pur abstractă**

Dacă încercăm altfel, eroare de compilare

```
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

SmallClock

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

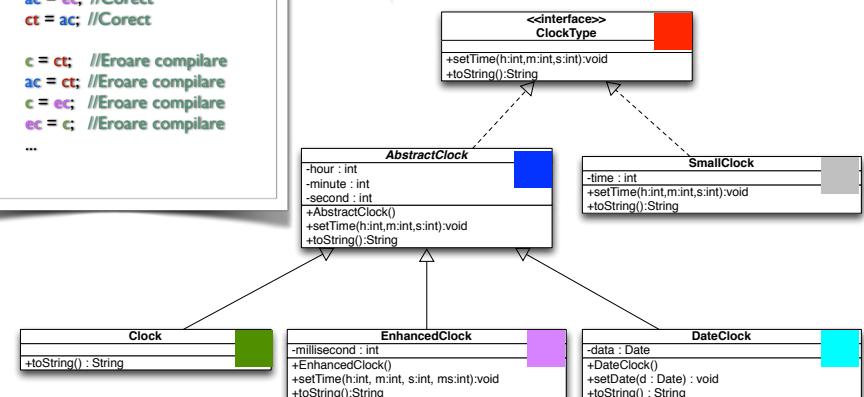
```
class SmallClock implements ClockType {
    private int time;
    public void setTime(int h, int m, int s) {
        if(h >= 0 && h < 24) {
            time = time & 0x0000FFFF; time = time | (h << 16);
        }
        if(m >= 0 && m < 60) {
            time = time & 0x00FF00FF; time = time | (m << 8);
        }
        if(s >= 0 && s < 60) {
            time = time & 0x0000FF00; time = time | s;
        }
    }
    public String toString() {
        return "SmallClock - " + ((time & 0x00FF0000) >> 16) + ":" +
               ((time & 0x0000FF00) >> 8) + ":" +
               (time & 0x000000FF);
    }
}
```

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

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

Ceasurile

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



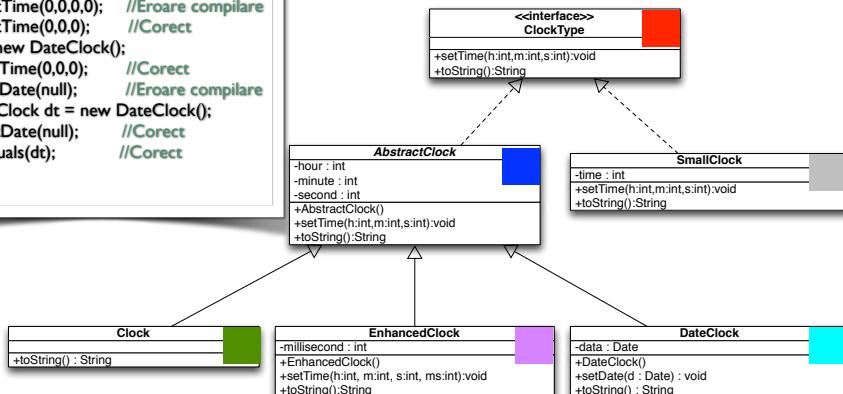
```

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

```

Ceasurile

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



```

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

```

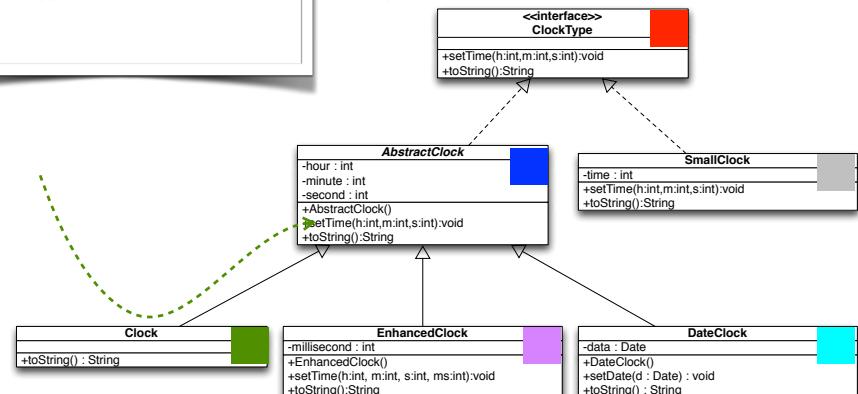
```

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

```

Ceasurile

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



```

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

```

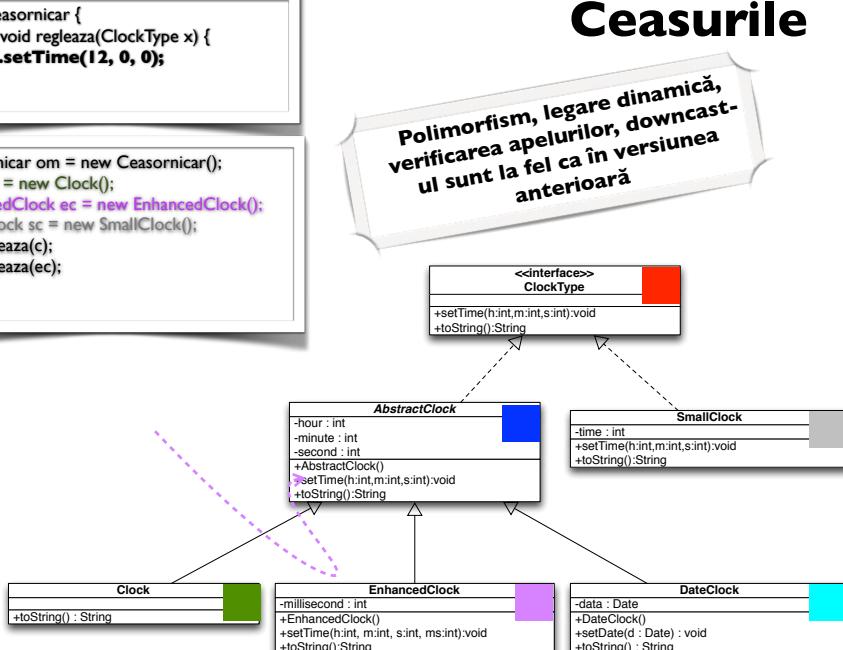
```

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());

```

Ceasurile

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



```

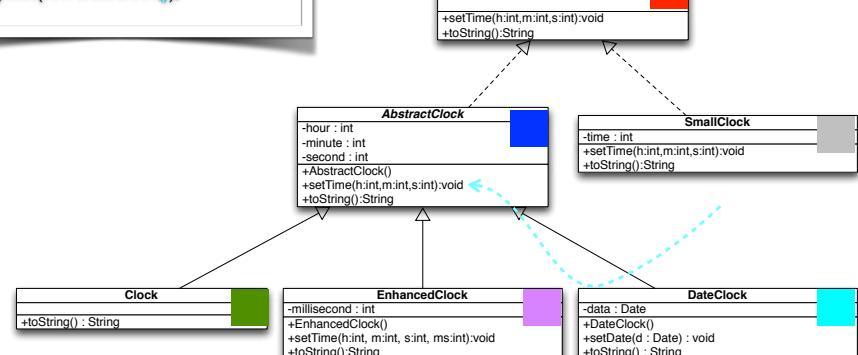
class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}

```

```

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());

```

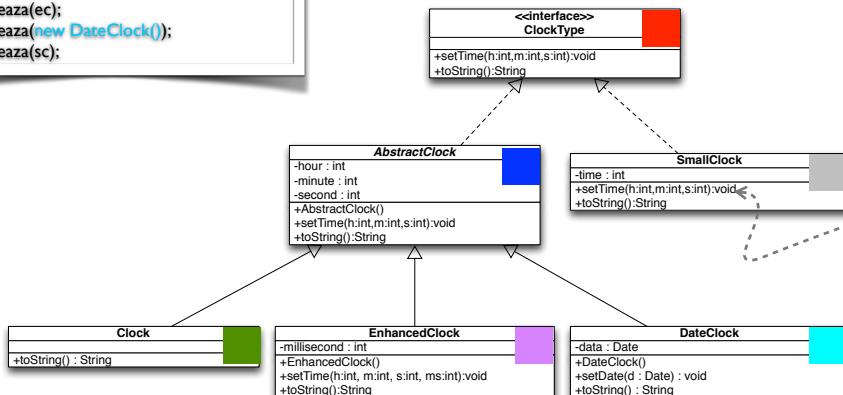


```
class Ceasornicar {
    public void regleaza(ClockType x) {
        x.setTime(12, 0, 0);
    }
}
```

```
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);
```

Ceasurile

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



extends între interfețe

```
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 EnhancedClockType extends ClockType {
    public void setTime(int h, int m, int s, int ms);
}
```

```
<<interface>>
EnhancedClockType
+setTime(h:int, m:int, s:int, ms:int):void
```

Ceva problemă?

```
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)

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();
}
```

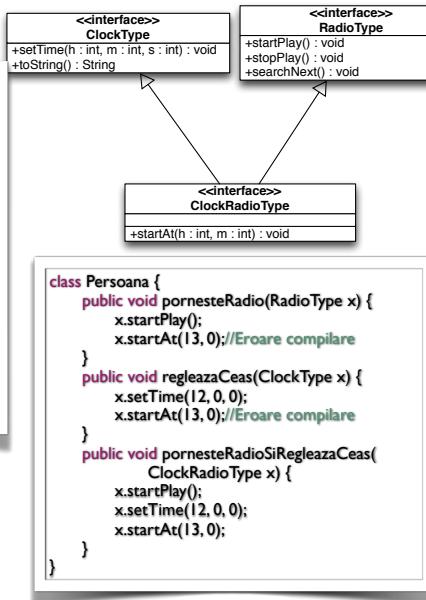
```
<<interface>>
ClockRadioType
+startAt(h : int, m : int) : void
```

```
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)

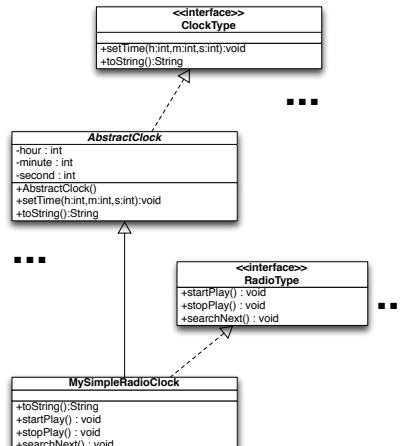
Atenție la ce putem apela

```
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(rct);
        om.regleazaCeaș(rt); //Eroare compilare
        om.pornesteRadioSiRegleazaCeaș(rt);
        om.pornesteRadioSiRegleazaCeaș(ct); //Eroare compilare
        om.pornesteRadioSiRegleazaCeaș(rt); //Eroare compilare
    }
}
```



implements again

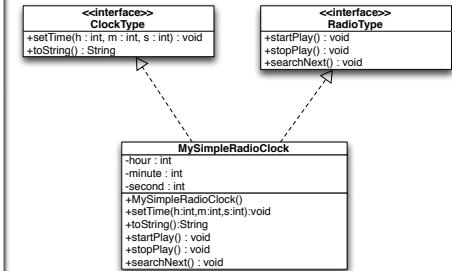
```
class MySimpleRadioClock extends AbstractClock
    implements RadioType {
    public String toString() {
        return "MySimpleRadioClock - " + super.toString();
    }
    public void startPlay() {
        System.out.println("Start");
    }
    public void stopPlay() {
        System.out.println("Stop");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



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

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");
    }
}
```



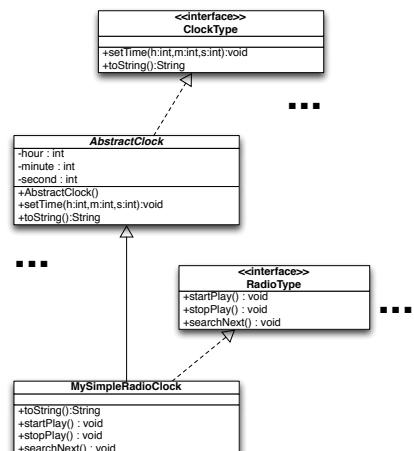
Cam multă duplicare ... dar există soluție :)

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);
```

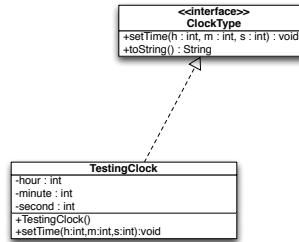
... și evident, oriunde e nevoie de un obiect **RadioType** putem folosi un **MySimpleRadioClock**



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;  
    }  
}
```

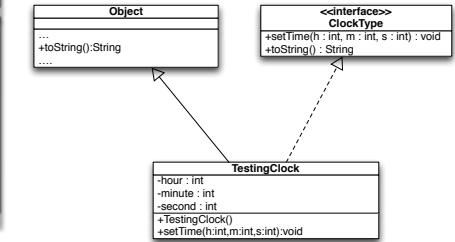


Nu există implementare în
TestingClock pt. `toString` și
totuși compilează

```
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;  
    }  
}
```

De ce o fi ok asta ?



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

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

Potentiale 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)

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

Interfață

- Nu putem factoriza codul comun diverselor implementări
 - ... dar am putea să-l factorizăm într-o clasă intermedieră 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 ?

Nici vorbă ! Interfața nu conține implementare

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 menționat explicit A.CONST sau B.CONST!