

# Curs 7 PP00

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

[cristian.ciurea@ie.ase.ro](mailto:cristian.ciurea@ie.ase.ro)

# Java fundamentals

- ▶ Singleton
- ▶ Callback

# Singleton

- ▶ **Singleton** este un model de proiectare (**design pattern**) care restricționează instanțierea unei clase la un singur obiect. Acest lucru este util atunci când este nevoie de exact un obiect pentru a coordona acțiunile în cadrul sistemului.
- ▶ Un design pattern este o soluție generală și reutilizabilă a unei probleme comune în proiectarea aplicațiilor software.
- ▶ Un design pattern nu este un design în forma finală, ceea ce înseamnă că nu poate fi transformat direct în cod. Acesta este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei.

# Singleton

- ▶ Design pattern-urile au fost inițial grupate în următoarele categorii: *Creational patterns*, *Structural patterns*, și *Behavioral patterns*.
- ▶ *Creational Patterns* sunt pattern-uri ce implementează mecanisme de creare a obiectelor. În această categorie se încadrează pattern-urile **Singleton** și **Factory**.
- ▶ *Structural Patterns* sunt pattern-uri ce simplifică design-ul aplicației prin găsirea unei metode de a defini relațiile dintre entități. În această categorie se încadrează pattern-ul **Decorator**.
- ▶ *Behavioral Patterns* sunt pattern-uri ce definesc modul în care obiectele comunică între ele. În această categorie se încadrează pattern-urile **Command**, **Visitor** și **Observer**.

# Singleton

- ▶ **Singleton** asigură faptul că o clasă are o singură instanță și oferă un punct global de acces la aceasta.
- ▶ Se dorește **crearea unei singure instanțe** pentru o clasă prin care să fie gestionată o resursă într-un mod **centralizat**.
- ▶ *Scenarii:*
  - ▶ Conexiune unică la baza de date
  - ▶ Gestiune unică fișiere
  - ▶ Gestiune unică conexiune la rețea
  - ▶ Gestiune unică preferințe pe platforma Android (SharedPreferences)
  - ▶ Gestiune centralizată a accesului la anumite resurse utilizate de soluție
  - ▶ Gestiune unică a unor obiecte costisitoare

# Singleton

- ▶ **Factory** definește o interfață pentru crearea unui obiect, dar lasă subclasele să decidă ce clasă să instanțieze.
- ▶ Implementarea unui mecanism centralizat prin care crearea obiectelor este **transparentă** pentru client.
- ▶ *Scenarii*: generator de documente, generator de caractere într-un joc (caracter negativ, caracter pozitiv).

# Singleton

- ▶ **Decorator** este un model de proiectare care permite adăugarea unui comportament unui singur obiect, într-o manieră statică sau dinamică, fără a afecta comportamentul celorlalte obiecte din aceeași clasă.
- ▶ Decorarea sau extinderea statică sau la run-time a funcționalității sau stării unor obiecte.
- ▶ *Scenariu*: copil în costum de Batman => devine Batman.

# Singleton

- ▶ **Command** este un model de proiectare comportamental în care un obiect este utilizat pentru a încapsula toate informațiile necesare pentru a efectua o acțiune sau a declanșa un eveniment ulterior.
- ▶ Aceste informații includ denumirea metodei, obiectul care deține metoda și valorile pentru parametrii metodei.



# Singleton

- ▶ **Visitor** este un model de proiectare care descrie o modalitate de a separa un algoritm de structura unui obiect pe care operează.
- ▶ Un rezultat practic al acestei separări este capacitatea de a adăuga noi operații la structurile obiectului existent fără modificarea structurilor.
- ▶ În esență, **Visitor** permite adăugarea de noi funcții virtuale unei familii de clase, fără a modifica clasele.

# Singleton

- **Observer** este un model de proiectare software în care un obiect denumit subiect păstrează o listă a dependenților săi, denumiți observatori, pe care îi notifică automat despre orice schimbări de stare, de obicei apelând una dintre metodele acestora.

# Singleton

- ▶ Modelul de Singleton trebuie să fie construit cu atenție în aplicațiile care utilizează mai multe fire de execuție (**multi-threading**).
- ▶ Soluția clasică a acestei probleme este de a utiliza excluderea reciprocă (**mutual exclusion**) pe clasa care indică faptul că obiectul este instanțiat.

# Singleton

- ▶ Scopul **Singleton** este de a controla crearea de obiecte din cadrul unei clase, limitând numărul la unu, dar în același timp să permită flexibilitatea de a crea mai multe obiecte în cazul în care situația se schimbă.
- ▶ Deoarece există doar o singură instanță **Singleton**, orice câmpuri ale unei instanțe de **Singleton** vor apărea o singură dată pentru fiecare clasă, similar cu attributele statice. **Singleton** deseori controlează accesul la resurse, cum ar fi conexiunile de baze de date sau socketi.

# Singleton

## Avantaj utilizare **Singleton**:

- ▶ este preferat variabilelor globale, deoarece nu încarcă namespace-ul global cu variabile care nu sunt necesare și permite inițializarea întârziată pentru a nu consuma inutil resursele sistemului.

## Dezavantaj utilizare **Singleton**:

- ▶ îngreunează testarea aplicației, deoarece introduce stări globale.

# Singleton

- ▶ La baza design pattern-ului **Singleton** stă o metodă ce permite crearea unei noi instanțe a clasei dacă aceasta nu există deja.
- ▶ Dacă instanța există deja, atunci întoarce o referință către acel obiect. Pentru a asigura o singură instanțiere a clasei, constructorul trebuie făcut **private** (un constructor privat împiedică reutilizarea sa sau accesul unei unități de testare).

# Singleton

```
public class Singleton {  
    private static final Singleton INSTANCE = new Singleton();  
  
    // private constructor prevents instantiation from other classes  
    private Singleton() { }  
  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

# Singleton

- ▶ Prin declararea unui constructor privat, utilizatorul clasei nu va putea să genereze o instanță nouă din această clasă, el fiind limitat la utilizarea singurei instanțe disponibile.
- ▶ Constructorul nu poate fi apelat decât din interiorul clasei, iar tentativa de a crea un obiect de tipul **Singleton** va genera eroare de compilator, deoarece se încearcă accesarea unui membru privat al clasei din afara ei.
- ▶ Metoda ce furnizează unica instanță a clasei se declară statică, deoarece dacă nu ar fi statică, ar fi necesară existența unei instanțe a clasei pentru ca metoda să poată fi apelată.



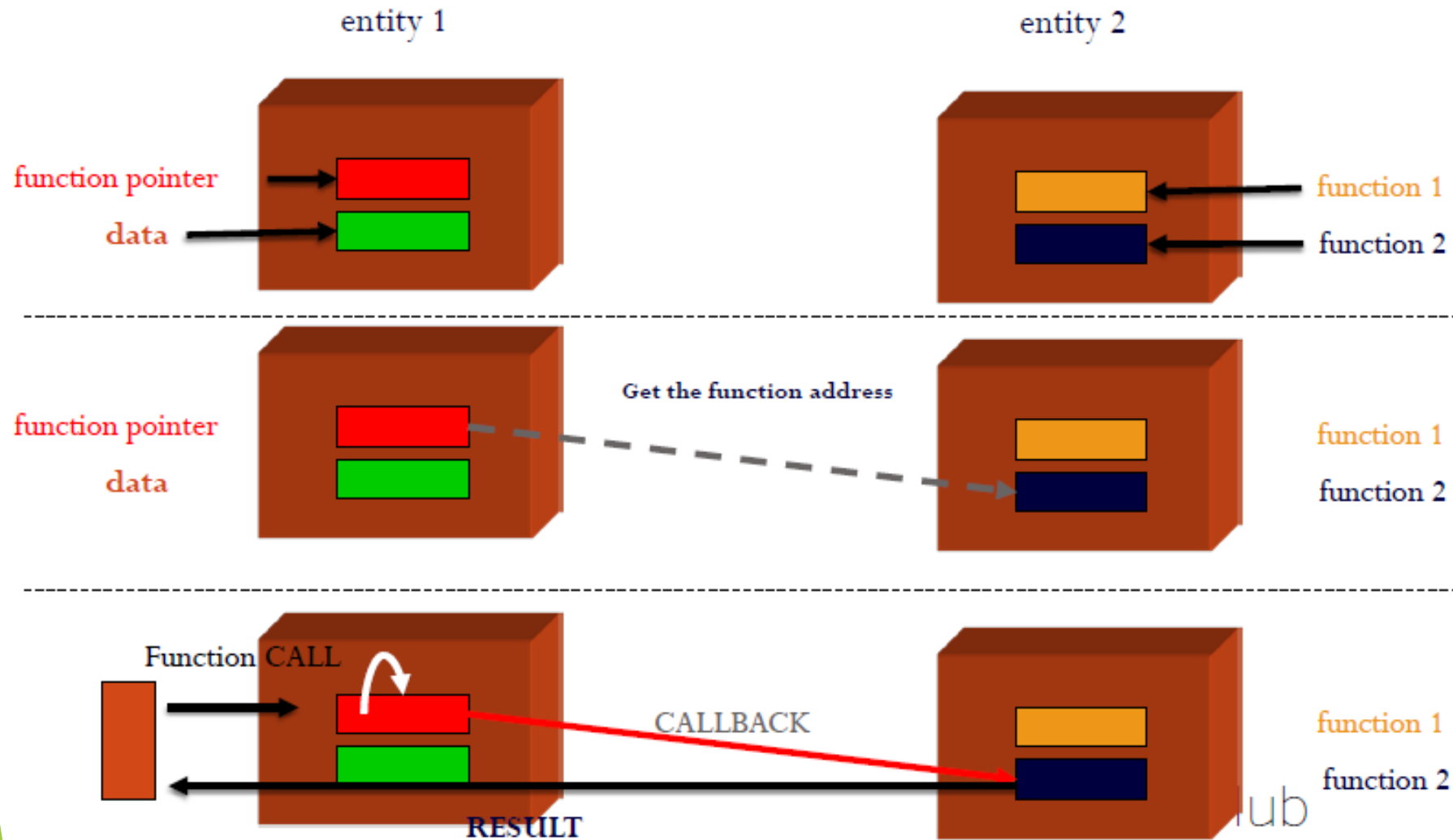
# Callback

- ▶ **Callback** reprezintă o bucată de cod executabil care este transmis ca argument unui alt cod, care se așteaptă să-l apeleze/execute înapoi la un moment dat.
- ▶ Invocarea poate fi imediată, ceea ce înseamnă un apel invers **sincron**, sau s-ar putea întâmpla mai târziu, adică un apel invers **asincron**.

# Callback

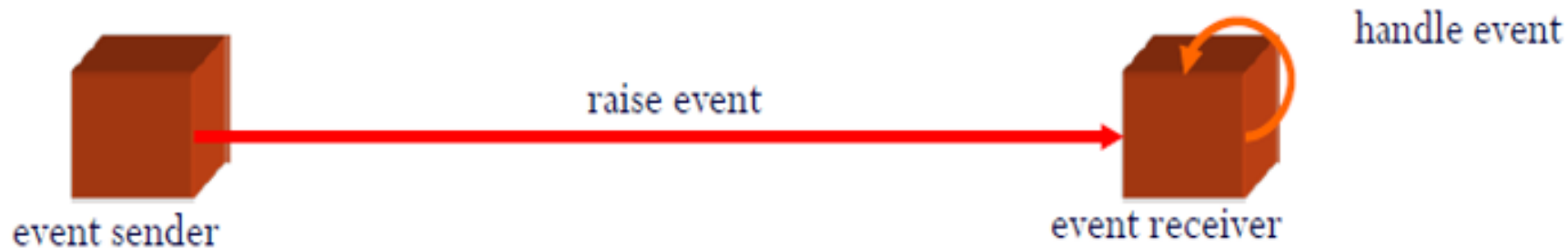
- ▶ Un callback (apel invers) este o secvență de cod care va fi transmisă unei anumite metode, astfel încât să poată fi apelată la un moment ulterior.
- ▶ Callback-ul este frecvent utilizat în programarea asincronă. În Java, funcțiile cu apel invers (callback) sunt utilizate în principal pentru a implementa un "Observer Pattern", care este strâns legat de programarea asincronă.
- ▶ În Java acest lucru poate fi în mod obișnuit realizat prin implementarea unei interfețe și transmiterea unui obiect care o implementează.

# Callback



# Callback

- ▶ Un **eveniment** este un mesaj transmis de un obiect pentru a anunța ceva (la nivelul interfeței grafice a unei aplicații - click de mouse, click de buton sau la nivel de logică a programului - rutine de aplicații).
- ▶ În modelul de gestionare a evenimentelor, obiectul care gestionează evenimentul nu cunoaște ce metodă va primi și va gestiona; de aceea este necesar ceva care să conecteze sursa și destinația.



# Callback

- ▶ În Java nu există pointeri la funcții (ca în C/C++);
- ▶ În Java nu există tipul delegate sau evenimente ca în .NET;
- ▶ Avem la dispoziție doar INTERFEȚE.

# Callback

- ▶ În C/C++ este posibil să declarăm pointeri la funcții ca argumente la alte funcții, însă în Java nu există pointeri.
- ▶ În Java, doar obiecte și tipuri de date primitive pot fi transmise ca parametri în cadrul metodelor unei clase.
- ▶ Suportul Java privind interfețele furnizează un mecanism prin care putem obține echivalentul callback-ului. Este necesar să se declare o interfață care declară la rândul ei funcția care se dorește a fi transmisă.

# Callback

- ▶ Un exemplu de implementare a mecanismului de apel invers în Java (callback) este dat de funcția **Collections.sort(List list, Comparator c)**
- ▶ În acest caz, **c** este o instanță a unei clase care implementează metoda **compare(e1, e2)** în interfața **Comparator**.
- ▶ Se sortează lista specificată în funcție de ordinea indusă de comparatorul specificat. Toate elementele din listă trebuie să fie reciproc comparabile folosind un comparator specificat.

# Callback

```
class CodedString implements Comparable<CodedString> {
    private int code;
    private String text;
    ...
    @Override
    public boolean compareTo(CodedString cs) {
        // Compare using "code" first, then
        // "text" if both codes are equal.
    }
}

...

public void sortCodedStringsByText(List<CodedString> codedStrings) {
    Comparator<CodedString> comparatorByText = new Comparator<CodedString>() {
        @Override
        public int compare(CodedString cs1, CodedString cs2) {
            // Compare cs1 and cs2 using just the "text" field
        }
    }
    Collections.sort(codedStrings, comparatorByText);
}
```



# Callback

- ▶ **Collections.sort(...)** va apela acest apel invers (**comparatorByText**) ori de câte ori trebuie să compare două elemente din lista care este sortată.
- ▶ Ca rezultat, vom obține lista sortată doar după câmpul "text". Dacă nu transmitem un apel invers, **Collections.sort()** va utiliza comparația implicită pentru clasă (mai întâi după câmpul "cod", apoi după câmpul "text").

# Callback

## Comparable vs. Comparator

- ▶ Interfața **Comparable** poate fi utilizată pentru a oferi un singur mod de sortare, în timp ce interfața **Comparator** este utilizată pentru a oferi diferite moduri de sortare.
- ▶ Pentru a utiliza **Comparable**, clasa trebuie să o implementeze, în timp ce pentru a utiliza **Comparator** nu este necesar să facem nicio schimbare în clasă.
- ▶ Interfața **Comparable** se afla în pachetul `java.lang`, în timp ce interfața **Comparator** este prezentă în pachetul `java.util`.
- ▶ Nu este necesar să facem modificări în cod pentru a utiliza **Comparable**, deoarece `Arrays.sort()` sau `Collection.sort()` utilizează automat metoda `compareTo()` a clasei. Pentru **Comparator**, este nevoie să se furnizeze clasa **Comparator** pentru a o utiliza în metoda `compare()`.

# Bibliografie

- ▶ [1] Jonathan Knudsen, Patrick Niemeyer - *Learning Java, 3<sup>rd</sup> Edition*, O'Reilly.
- ▶ [2] <http://www.itcsolutions.eu>
- ▶ [3] <http://www.acs.ase.ro>
- ▶ [4] [http://en.wikipedia.org/wiki/Singleton\\_pattern](http://en.wikipedia.org/wiki/Singleton_pattern)
- ▶ [5] [http://en.wikipedia.org/wiki/Callback\\_\(computer\\_programming\)](http://en.wikipedia.org/wiki/Callback_(computer_programming))
- ▶ [6] [http://cursuri.cs.pub.ro/~poo/wiki/index.php/Design\\_Patterns\\_Basics](http://cursuri.cs.pub.ro/~poo/wiki/index.php/Design_Patterns_Basics)