Incapsulare, imutabilitate si final, static vs instante, Singleton

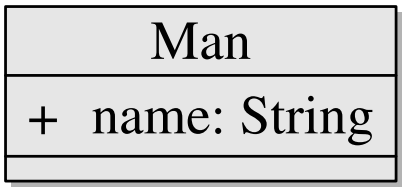
Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică si Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

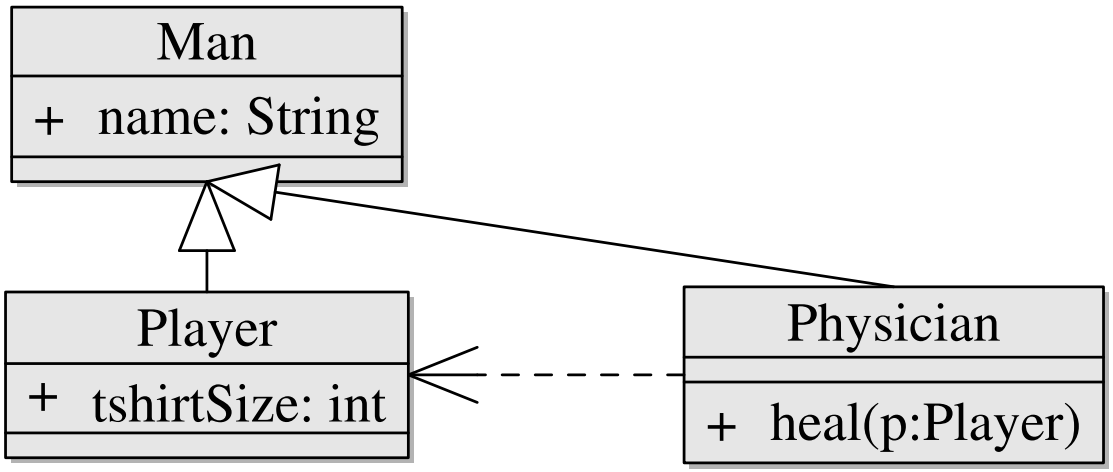
OOP, 2020

Universitatea
Politehnica
București

Curs 2: ce contine o clasa



Curs 3: relatia dintre clase



Static vs instante

static

static desemnează membrii și metode independente de instanțele clasei

```
public class BasketBall {
  public static int size = 12;
  private String color;

  Ball(String color) {
    this.color = color;
  }
}
```

static

static desemnează membrii și metode independente de instanțele clasei

```
public class BasketBall {
  public static int size = 12;
  private String color;

  Ball(String color) {
    this.color = color;
  }
}
```

static vs non-static

ceea ce se petrece într-o funcție statică ar trebui să poată funcționa chiar dacă obiectul a fost instanțiat sau nu

static vs non-static

ceea ce se petrece într-o funcție statică ar trebui să poată funcționa chiar dacă obiectul a fost instanțiat sau nu

```
public class TestApp {
        int count;
        public static void main(String[] args){
                count++;
        }
}
```

static vs non-static

ceea ce se petrece într-o funcție statică ar trebui să poată funcționa chiar dacă obiectul a fost instanțiat sau nu

```
public class TestApp {
        int count;
        public static void main(String[] args){
                count++;
        }
}
```

Primim eroare la compilare:
"Cannot make a static reference to the non-static field count"

Exercitiu: Cannot make a static reference to the non-static field count

Exercitiu: Cannot make a static reference to the non-static field count

static vs non-static

Dacă count chiar nu este static, o posibilă rezolvare ar fi:

```
public class TestApp {
        int count;
        public static void main(String[] args){
                TestApp app = new TestApp();
                app.incrementCount();
        }
        public void incrementCount() {
                count++;
        }
}
```

Imutabilitate si final

Date primitive constante

În Java, cuvântul cheie final

```
public class Album {

    public final int tracks = 15;
    public final int releaseYear;

    Album(int releaseYear) {
      this.releaseYear = releaseYear;
    }

}
```

Obiecte constante

```
public class Boxer {
    public final int birthYear;
    public String name;

    Boxer(String name, int birthYear) {
      this.name = name;
      this.birthYear = birthYear;
    }
}
...
final Boxer champion = new Boxer("Cassius Clay", 1942);
```

Obiecte constante vs campuri

Variabila referință final permite modificarea internă a obiectului către care indică:

```
public class Boxer {
    public final int birthYear;
    public String name;

    Boxer(String name, int birthYear) {
      this.name = name;
      this.birthYear = birthYear;
    }
}
...
final Boxer champion = new Boxer("Cassius Clay", 1942);
champion.name = "Muhammad Ali";    // ok
```

Obiecte constante vs campuri

O încercare nouă de a asigna o variabilă final rezultă în eroare de compilare

```
public class Boxer {
    public final int birthYear;
    public String name;

    Player(String name, int birthYear) {
      this.name = name;
      this.birthYear = birthYear;
    }
}
...
final Boxer champion = new Boxer("Cassius Clay", 1942);
champion = new Boxer("Joe Frazier", 1952);   // eroare
champion.birthYear = 2016;                   // eroare
```

Immutable objects

Immutable objects: toate atributele unui obiect admit o unică inițializare:

```
public class Ball {
  public final String color;
  public Ball(String color) {
    this.color = color;
  }
}
Ball basketball = new Ball("brown");
```

Immutable objects

Immutable objects: toate atributele unui obiect admit o unică inițializare:

```
public class Ball {
  public final String color;
  public Ball(String color) {
    this.color = color;
  }
}
Ball basketball = new Ball("brown");
```

Așa sunt, spre exemplu, obiectele de tip String sau Integer

Vectori constanti

- Nu folosiți `final` pentru array-uri
  - array-ul e final, însă obiectele din el pot fi modificate :)
  - folosiți colecții nemodificabile[1]

[1] exemplu Chapter 4, pg 70 in Joshua Bloch. 2008. Effective Java (2nd Edition) (The Java Series) (2 ed.). Prentice Hall

Clase care nu pot fi mostenite

Puteti folosi final pentru clase care doriti sa nu poata fi mostenite

```
public final class Animal {
    ...
}
public class Dog extends Animal {
    ...
}
```

error: cannot inherit from final Animal

Immutable vs composition

Puteti marca drept final obiectele folosite in relatia de Compunere

```
public class Punct2D {
    ...
}
public class Circle {
    public final Punct2D centru;
    ...
}
```

Singleton

Ce sunt Design Patterns?

Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
- *Christopher Alexander*

Ce sunt Design Patterns?

Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.
- *Christopher Alexander*

e.g. Courtyard

Clasificare Design Patterns

- *creational patterns*: definesc mecanisme de creare a obiectelor (GoF)
- *structural patterns*: definesc relații între entități (GoF)
- *behavioral patterns*: definesc comunicarea între entități (GoF)
- *concurrency patterns*: definesc mecanisme utile in programarea pe paralela si distribuita (sincronizare etc.)
- *architectural patterns*: descriu structura intregului sistem (e.g. MVC, multi-tier etc.)

GoF Design Patterns

E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)
"Design Patterns: Elements of Reusable Object-Oriented Software"

Creational	Structural	Behavioral
Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Visitor

Description

Singleton ensures a class only has one instance, and provides a global point of access to it.

Problem that the pattern solves

- uneori este nevoie sa avem o singura instanta a unei clase

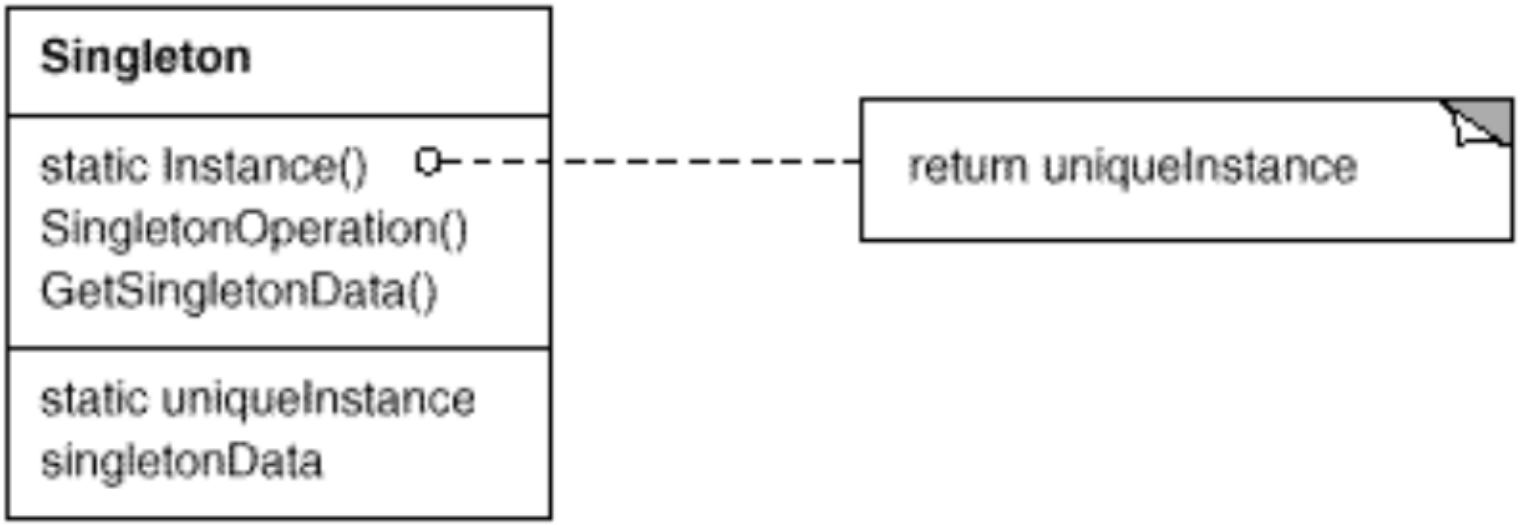
Problem that the pattern solves

- uneori este nevoie sa avem o singura instanta a unei clase
- ar trebui sa ne putem razgandi relativ usor

Problem that the pattern solves

- uneori este nevoie sa avem o singura instanta a unei clase
- ar trebui sa ne putem razgandi relativ usor
- trebuie sa fie clar pentru oricine foloseste codul nostru cum poate obtine acea instanta

Solution Structure



* E. Gamma, R. Helm, R. Johnson, J. Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software"

Exercitiu: Singleton

Exercitiu: Singleton

Implementation Examples

```
public class ClassicSingleton {

    private static ClassicSingleton instance = null;
    private ClassicSingleton() {
        // Exists only to defeat instantiation.
    }

    public static ClassicSingleton getInstance() {
        if(instance == null) {
            instance = new ClassicSingleton();
        }
        return instance;
    }
}

ClassicSingleton c = ClassicSingleton.getInstance();
```

Solution Details

- clasa incapsuleaza unica sa instanta si gestioneaza accesul la ea

Solution Details

- clasa incapsuleaza unica sa instanta si gestioneaza accesul la ea
- exista alternativa de a folosi metode statice, dar este mai rigida (in C++ nu pot fi extinse, in general mai greu sa ne razgandim)

Consequences

- acces controlat la unica instanta

Consequences

- acces controlat la unica instanta
- evita poluarea cu variabile globale

Consequences

- acces controlat la unica instanta
- evita poluarea cu variabile globale
- clasa poate fi rafinata simplu prin extindere

Consequences

- acces controlat la unica instanta
- evita poluarea cu variabile globale
- clasa poate fi rafinata simplu prin extindere
- e usor sa ne razgandim si sa permitem mai multe instante, chiar putem pune o limita a numarului de instante

Consequences

- acces controlat la unica instanta
- evita poluarea cu variabile globale
- clasa poate fi rafinata simplu prin extindere
- e usor sa ne razgandim si sa permitem mai multe instante, chiar putem pune o limita a numarului de instante
- mai flexibila decat alternativa de a folosi metode statice

Încapsulare

Încapsulare

Definition

Încapsularea este proprietatea claselor de obiecte de:

- a grupa datele și metodele aplicabile asupra datelor
- a proteja accesul la acestea față de utilizarea eronată

Încapsulare

Cum contribuie urmatoarele elemente la incapsulare?

- ce poate contine o clasa
- specificatori de acces
- setter si getter

Încapsulare

```
class GeneratorNume {
  private String[] vocabular = {new String("Ana"), new
      String("Mihai"), new String("Robert")};
  private Random randomGenerator = new Random();

  public String genNume() {
    return vocabular[randomGenerator.nextInt(3)];
  }
}
...
GeneratorNume gn = new GeneratorNume();
System.out.println(gn.genNume());
```

Încapsulare vs Mostenire

Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation". Inheritance and composition each have their advantages and disadvantages.
- E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four), "Design Patterns" (Introduction)

Exercitiu: Inheritance vs Composition

Exercitiu: Inheritance vs Composition

Reading material

- Lab 4: Static. Final. Singleton Design Pattern
- Inheritance vs Compositions, Introduction, "Design Patterns", E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)
- Delegation, Introduction, "Design Patterns", E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)