Agregare vs moștenire, upcasting, downcasting, overriding, overloading, super

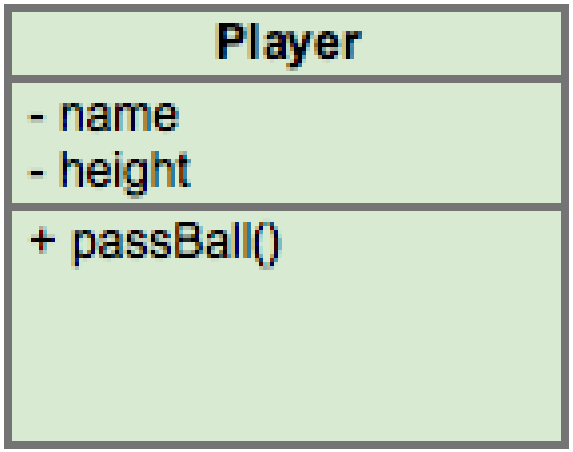
Alexandru Olteanu

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



Precizări diagrame

- UML (Unified Modeling Language) este un standard pentru modelarea sistemelor software
- "+" e pentru membrii publici, "#" protected, "-" private, "$\sim$" default (package)
- Asocierea, agregarea, compunerea și moștenirea se deduc din tipul săgeții folosite între componente
- Nu este indicată includerea getterilor și setterilor în diagrame (pot ajunge foarte mari)
- Metodele moștenite apar în diagramele claselor derivate doar în cazul în care sunt suprascrise

Relații între clase

- Asociere (*association*)
- Agregare (*aggregation*)
- Compunere (*composition*)
- Moștenire (*inheritance*)

Asocierea

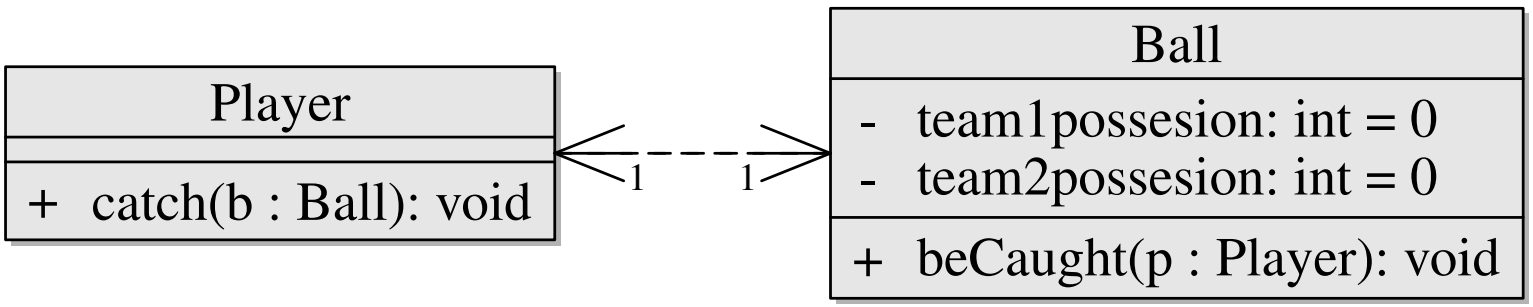
Asocierea

- Reprezintă o relație între clase privite ca entități independente (...is using...)
- Fiecare entitate are propriul ciclu de viață
- Relația poate fi unu-la-unu, unu-la-mulți, mulți-la-unu, mulți-la-mulți

Asocierea: exemplu

```
public class Player {
  public void catch(Ball b) {
    b.beCaught(this);
  }
}

public class Ball {
  private team1Possesion = 0, team2Possesion = 0;
  public void beCaught(Player p) {
    if (p.getTeam()==team1) {
      team1Possesion++;
    } else {
      team2Possesion++;
    }
  }
}
```

Asocierea: UML



Agregarea

Agregare

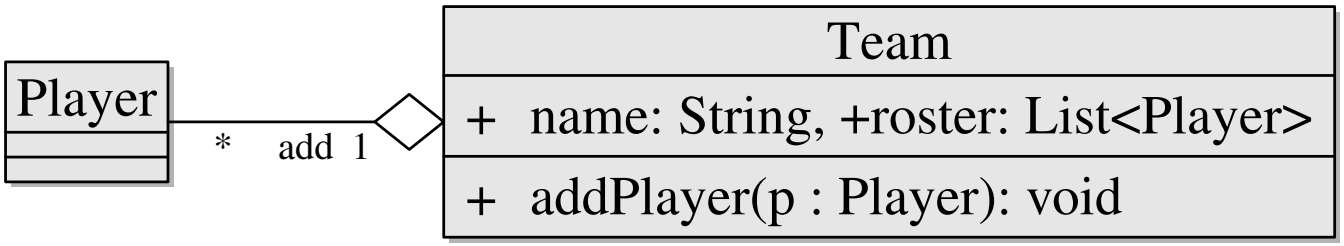
Agregarea

- o clasă conține o referință către o altă clasă (...has a...)
- relația HAS—A
- este unidirecțională (*one way association*)
- *weak association* - obiectul container poate exista și fără obiectele agregate

Agregare: exemplu

```
public class Player {
  ...
}

public class Team {
  public String name;
  public List<Player> roster;
  ...
  public addPlayer(Player p) {
    roster.add(p);
  }
}
```

Agregare: UML



Compunerea

Compunere

Compunerea

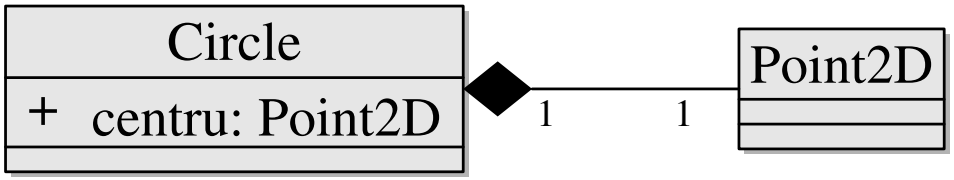
- *strong association* - clasele sunt dependente una de alta și nu pot exista una fără cealaltă
- numită și "Death relationship"[1]

[1] tutorial bun despre tipurile de asociere: http://www.codeproject.com/Articles/330447/Understanding-Association-Aggregation-and-Composit

Compunere: exemplu

```
public class Point2D {
  ...
}

public class Circle {
  public Point2D centru;
  public Float raza;
}
```

Compunere: UML



Moștenirea

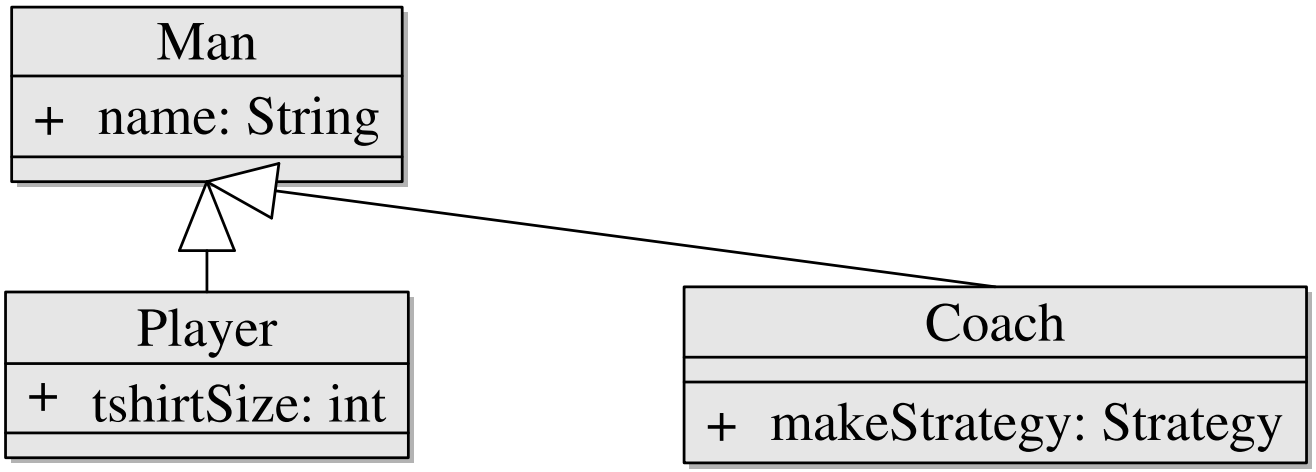
Moștenirea

- Permite unei clase să fie subclasa unei alte clase, superclasa (...is a...)
- În Java o clasă poate extinde doar o singură clasă (revenim cand o sa vorbim de Mostenire multipla data viitoare)
- O subclasă moștenește toate câmpurile și metodele public și protected ale superclasei
- Constructorii sunt apelați ierarhic: constructorul subclasei apelează constructorul superclasei
  - nu se apelează explicit dacă constructorul superclasei nu are parametri sau este 'default'
  - se apelează cu 'super' dacă clasa are doar constructori cu parametri

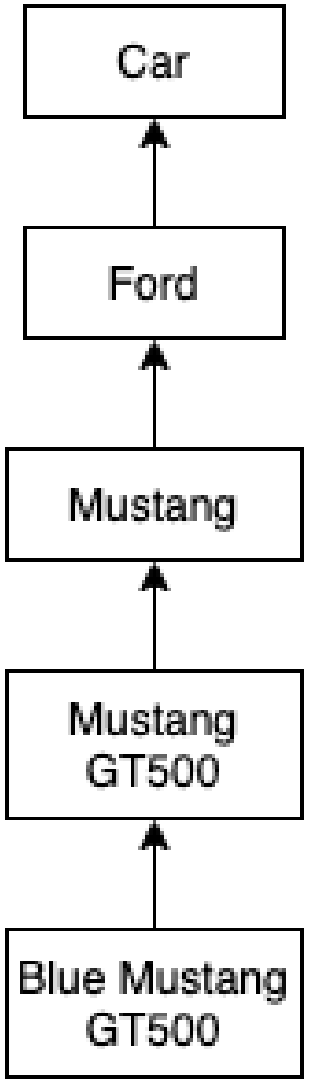
Moștenire: exemplu

```
public class Man {
  public String name;
}

public class Player extends Man {
  public int tshirtNo;
}

public class Coach extends Man {
  public Strategy makeStrategy() {
    ...
  }
}
```

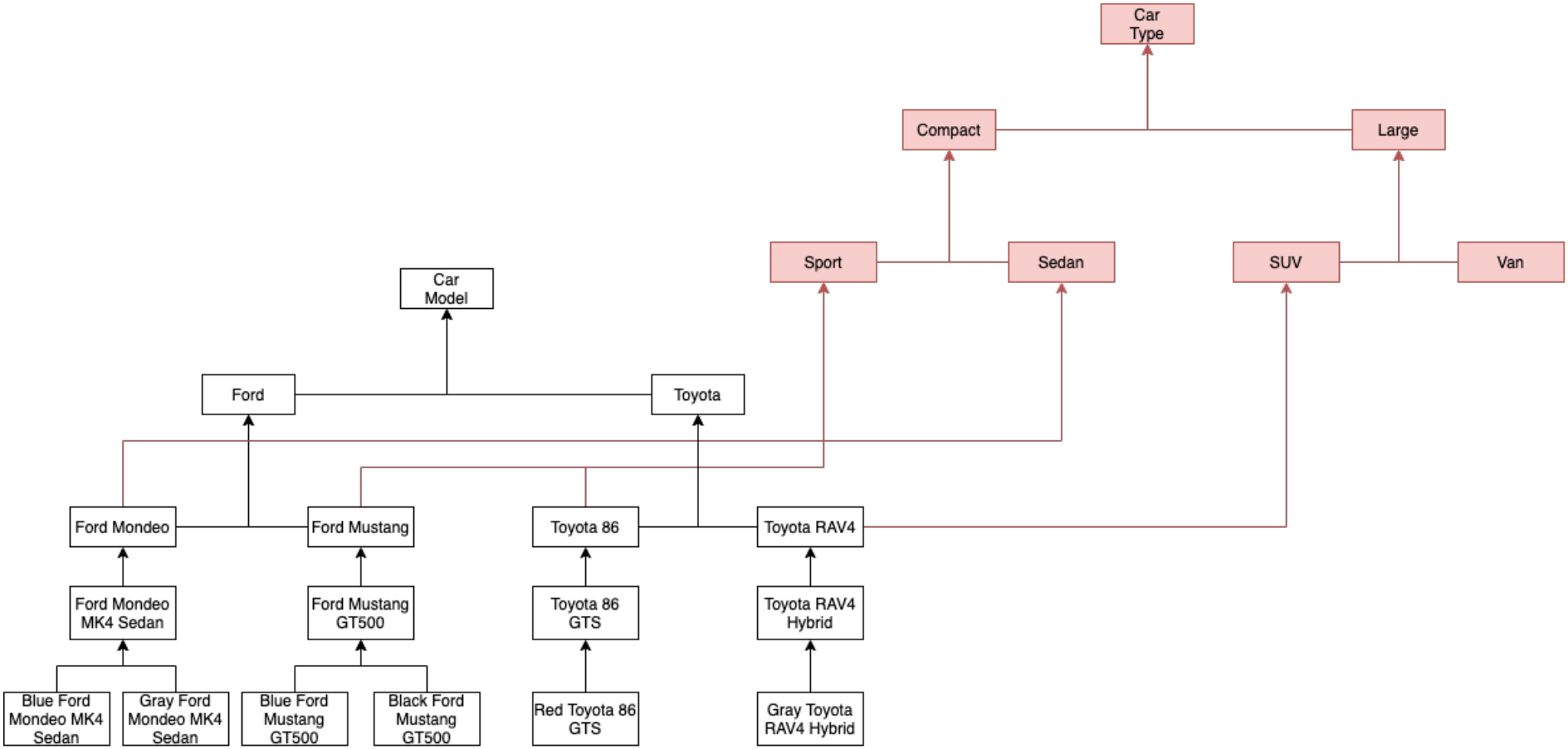
Moștenire: UML



Moștenire: UML



Moștenire: UML



Overriding (suprascrierea)

O clasă poate suprascrie orice metodă dintr-o superclasă a sa prin definirea unei metode cu aceeași semnătură (inclusiv constructori).

Overriding (suprascrierea)

O clasă poate suprascrie orice metodă dintr-o superclasă a sa prin definirea unei metode cu aceeași semnătură (inclusiv constructori).
În metoda din subclasă se poate folosi cuvantul cheie super pentru a apela metoda suprascrisă din superclasă:

```
public class Man {
  public void train(Gym gym) {
    gym.liftWeights();
  }
}

public class Player extends Man {
  public void train(Gym gym) {
    super.train(gym);
    gym.cardio();
  }
}
```

Overriding (suprascrierea)

Dar atentie la folosirea acestui mecanism pentru a nu ajunge in situatii nedorite:

```
public class Man {
  public void train(Gym gym) {
    gym.enter();
    gym.liftWeights();
    gym.exit();
  }
}

public class Player extends Man {
  public void train(Gym gym) {
    super(gym);
    gym.cardio();
  }
}
```

Overriding și prezența constructorilor

Următorul cod compilează sau nu?

```
public class Man {
  protected String name;
  public Man(String name){
        this.name = name;
  }
}

public class Coach extends Man {
}
```

Overriding și prezența constructorilor

Soluția 1: constructor default în superclasă

```
public class Man {
  protected String name;
  public Man() {
    this.name = "anonim";
  }
  public Man(String name){
        this.name = name;
  }
}

public class Coach extends Man {
}
```

Overriding și prezența constructorilor

Solutia 2: suprascriem constructorul cu parametrii în subclasă

```
public class Man {
  protected String name;
  public Man(String name){
        this.name = name;
  }
}

public class Coach extends Man {
  public Coach(String name){
    super(name);
  }
}
```

Overriding: exemplu toString

Implicit, orice clasă moștenește **java.lang.Object** deci are metodele: clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait

Overriding: exemplu toString

Implicit, orice clasă moștenește **java.lang.Object** deci are metodele: clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait
Dar ar trebui suprascrise pentru a face ceva relevant pentru clasa respectivă.

Overriding: exemplu toString

Implicit, orice clasă moștenește **java.lang.Object** deci are metodele: clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait
Dar ar trebui suprascrise pentru a face ceva relevant pentru clasa respectivă.

```
public class Man {
  public String name;
}

public class Player extends Man {
  public int tshirtNo;
  public String toString() {
    return "("+tshirtNo+") "+name;
  }
}
```

Overloading (supraîncarcarea): nu confundați cu Overriding

O clasă poate avea oricâte metode cu același nume, dacă listele de parametrii sunt diferite.

Overloading (supraîncarcarea): nu confundați cu Overriding

O clasă poate avea oricâte metode cu același nume, dacă listele de parametrii sunt diferite.

```
public class Adder {
  public Integer sum(Integer a, Integer b) {
    return a+b;
  }

  public Integer sum(List<Integer> numbers) {
    Integer result = 0;
    for (Integer number : numbers) {
      result+=number;
    }
    return result;
  }
}
```

Suprascriere și supraîncărcare operatori

Operator overloading and overriding are not supported in Java.

```
class Point{
  public double x;
  public double y;

  public Point(int x, int y){
      this.x = x;
      this.y = y;
  }

  public Point add(Point other){
      this.x += other.x;
      this.y += other.y;
      return this;
  }
}
...
Point a = new Point(1,1);
Point b = new Point(2,2);
a.add(b);
```

Upcast

Casting to a supertype. Se întâmplă automat.

```
public class Man {
  protected String name;
  public String getName() {
      return name;
  }
  public void greet(Man otherGuy) {
      System.out.println("Hi "+otherGuy.getName());
  }
}

public class Coach extends Man {
  public String getName() {
    return "coach "+name;
  }
}

Man mike = new Man("Mike");
Man charlie = new Coach("Charlie"); // automat
mike.greet(charlie);
```

Tip declarat vs instantiat

```
public class Man {
  protected String name;
  public String getName() {
      return name;
  }
  public void greet(Man otherGuy) {
      System.out.println("Hi "+otherGuy.getName());
  }
}

public class Coach extends Man {
  public String getName() {
    return "coach "+name;
  }
}

Man mike = new Man("Mike");
Man charlie = new Coach("Charlie"); // runtime coach
mike.greet(charlie);
```

Downcast

Casting to a subtype. Trebuie făcut manual.

```
public class Man {
  protected String name;
  public String getName() {
      return name;
  }
  public void greet(Man otherGuy) {
      System.out.println("Hi "+otherGuy.getName());
      System.out.println(((Coach)otherGuy).giveSignature());
  }
}

public class Coach extends Man {
  public String getName() { return "coach "+name; }
  public String giveSignature() { return "signature"; }
}

Man mike = new Man("Mike");
Man charlie = new Coach("Charlie");
mike.greet(charlie);
```

Downcast

ClassCastException la rulare

```
public class Man {
  protected String name;
  public String getName() {
      return name;
  }
  public void greet(Man otherGuy) {
      System.out.println("Hi "+otherGuy.getName());
      System.out.println(((Coach)otherGuy).giveSignature());
  }
}

public class Coach extends Man {
  public String getName() { return "coach "+name; }
  public String giveSignature() { return "signature"; }
}

Man mike = new Man("Mike");
Man charlie = new Man("Charlie");
mike.greet(charlie);
```

Downcast

Ca să nu riscăm ClassCastException (eroare la rulare) putem folosi instanceof:

```
public class Man {
   ...
   public void greet(Man otherGuy) {
      System.out.println("Hi "+otherGuy.getName());
      if (otherGuy instanceof Coach)
         System.out.println(((Coach)otherGuy).giveSignature());
   }
}

...

Man mike = new Man("Mike");
Man charlie = new Man("Charlie");
mike.greet(charlie);
```

Downcast

Ca să nu riscăm ClassCastException (eroare la rulare) putem folosi supraîncărcarea:

```
public class Man {
   ...
   public void greet(Man otherGuy) {
        System.out.println("Hi "+otherGuy.getName());
   }
   public void greet(Coach otherGuy) {
        System.out.println("Hi "+otherGuy.getName());
        System.out.println(((Coach)otherGuy).giveSignature());
   }
}

...

Man mike = new Man("Mike");
Coach charlie = new Coach("Charlie");
mike.greet(charlie);
```

Revenim la ideea asta cand discutam despre Visitor.

Tipul declarat vs instanțiat

Ca să nu riscăm ClassCastException (eroare la rulare) putem folosi supraîncărcarea:

```
public class Man {
   ...
   public void greet(Man otherGuy) {
        System.out.println("Hi "+otherGuy.getName());
   }
   public void greet(Coach otherGuy) {
        System.out.println("Hi "+otherGuy.getName());
        System.out.println(((Coach)otherGuy).giveSignature());
   }
}

...

Man mike = new Man("Mike");
Man charlie = new Coach("Charlie"); // atentie la tip!
mike.greet(charlie);
```

Reading Assignments

- Lab 03: agregare și moștenire
- Understanding Association, Aggregation, and Composition
- UML Association vs Aggregation vs Composition
- Generate UML class diagrams from IntelliJ IDEA
- Generate UML Class Diagram from Java Project