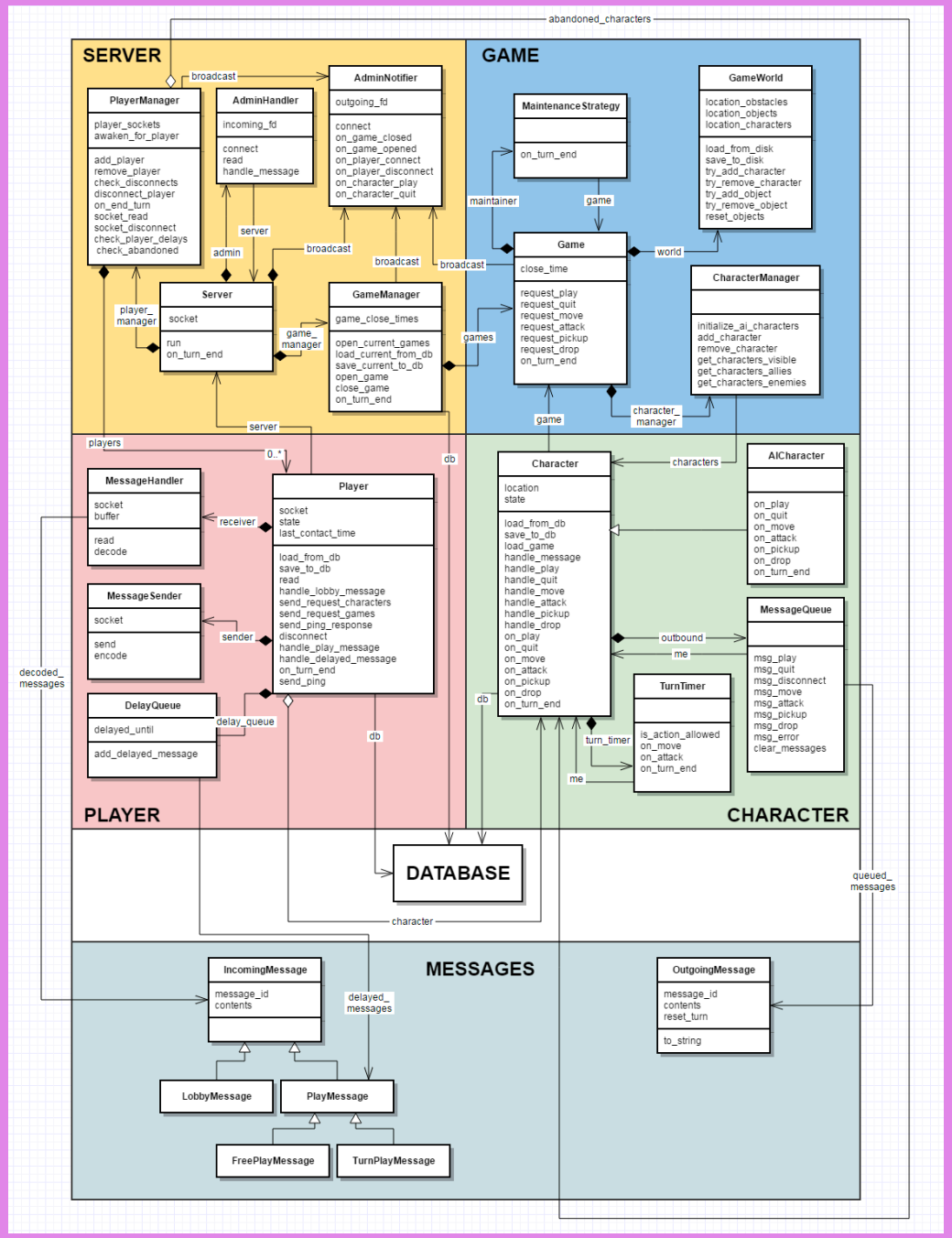


Search . . .

02

OOP – Objektově Orientovaný Přístup

- Objekty, třídy, rozhraní, hierarchie
- Overloading, overriding
- Class diagramy a modelování statické struktury
- Kompozice versus dědičnost
- Správná reprezentace

Objektové Modelování - podzim 2022

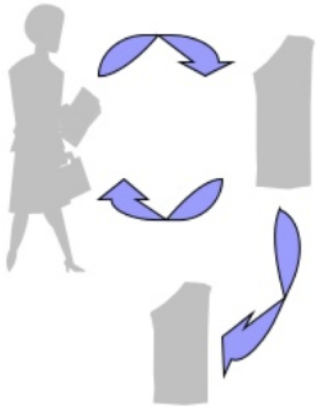
david.kadlecek@cvut.cz, www.czm.fel.cvut.cz

02 OBJEKTOVÝ PŘÍSTUP

Problém se snažíme řešit tak, že ho kompletně strukturujeme do objektů, které mezi sebou komunikují. Objekty intuitivně volíme tak, aby co nejvíce odpovídaly objektům z reálného světa.

Procedurální nebo funkcionální přístup

Objektový přístup



- výběr peněz
- vklad peněz
- převod peněz mezi účty

- zákazník
- účet
- peníze

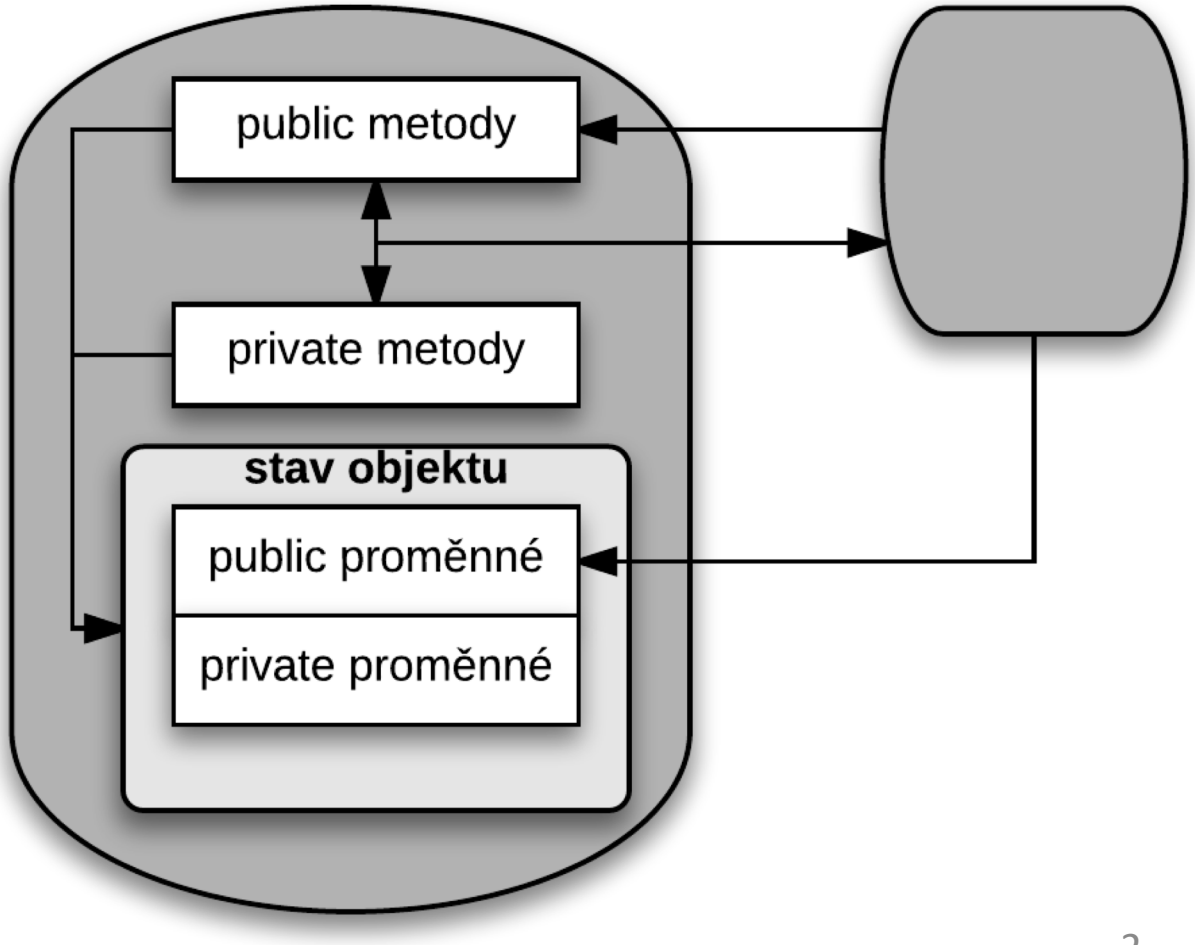
Jaký přístup je lepší?

Záleží na problému, který řeším

02 OBJEKT

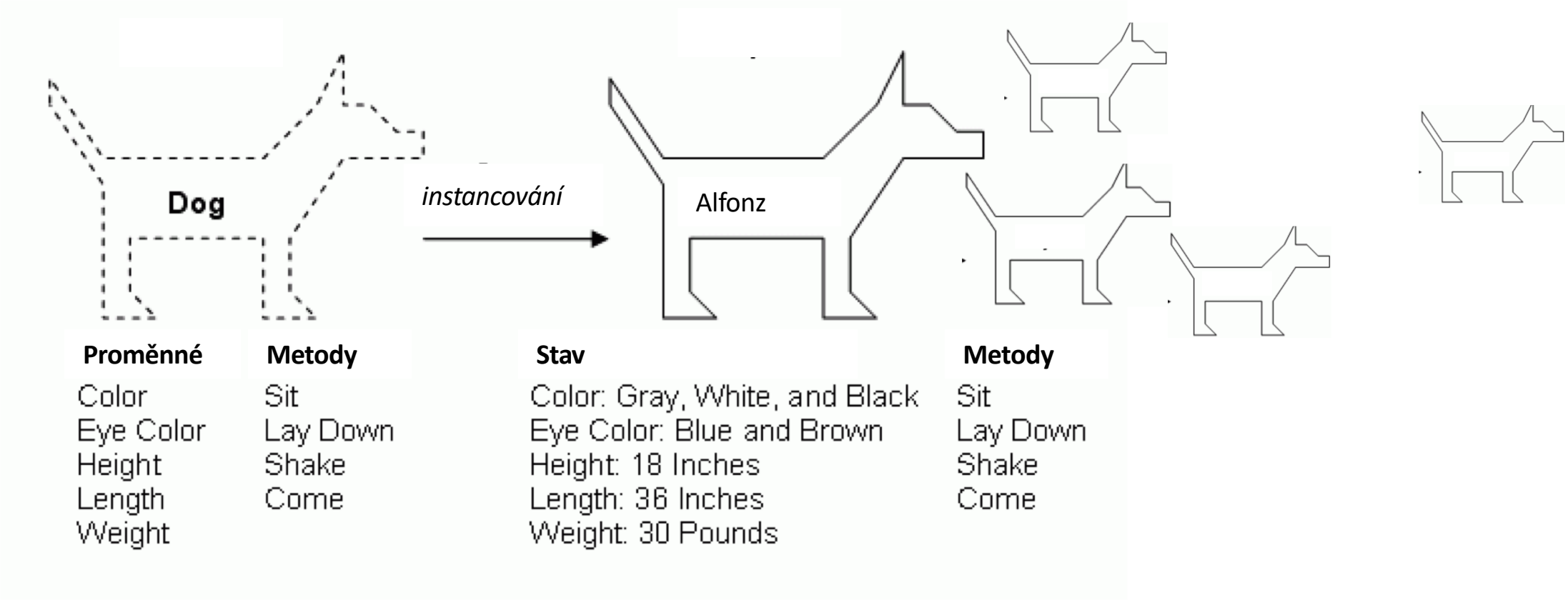
Objekt do sebe zapouzdřuje ***proměnné***, které reprezentují jeho stav a **metody**, které s tímto stavem pracují (vracet hodnotu, upravit hodnotu …)

Uvnitř objektu má všechno přístup ke všemu. Z vnějšího světa lze volat pouze veřejné metody (***public***) a napřímo pracovat pouze s veřejnými proměnnými (narozdíl od ***private***).



02 TŘÍDA

Třída je “**předpis” pro tvorbu a chování objektů** - instancí této třídy. Z každé třídy můžeme vytvořit *N* instancí (objektů). Ty se od sebe liší pouze stavem.



02 TŘÍDA

Třída je “**předpis” pro tvorbu a chování objektů** - instancí této třídy. Z každé třídy můžeme vytvořit *N* instancí (objektů). Ty se od sebe liší pouze stavem.

```java
public class Karel {
    private int position;
    public boolean isSmiling;
    public int getPosition() {
        return position;
    }
    public Karel(int position){
        this.position = position;
    }
    private void updatePosition(int step){
        position = position + step;
    }
    public void stepForward() {
        updatePosition(1);
    };
    public void stepBackward() ...
}
```

```java
public class BabyProgrammer {
    public static void main(String[] args){
        Karel myKarel = new Karel(10);
        myKarel.isSmiling = false;
        myKarel.stepForward();
        myKarel.stepForward();
        myKarel.stepBackward();
        myKarel.isSmiling = true;
        System.out.println("Position is " + myKarel.getPosition());
    }
}
```

02 RECORDS

```java
public class Person {
    private final String name;
    private final String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public int hashCode() {
        return Objects.hash(name, address);
    }

    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        } else if (!(obj instanceof Person)) {
            return false;
        } else {
            Person other = (Person) obj;
            return Objects.equals(name, other.name)
                    && Objects.equals(address, other.address);
        }
    }

    public String toString() {
        return "Person [name=" + name + ", address=" + address + "]";
    }
}
```

Java 14

Imutable verze třídy primárně pro účely držení stavu objektu

```java
public record Person (String name, String address) {}
```

Pozn. V případě nižší verze Javy se podívej na Lombok

5

02 INTERFACES - DEFAULT METHOD

```java
public interface Vehicle {
    void cleanVehicle();
}
```

Pro zajištění zpětné kompatibility, když chceme do interface přidat novou metodu a nechceme sahat do všech implementací tohoto interface.

```java
public class Car implements Vehicle{
    public void cleanVehicle() {
        System.out.println("Cleaning the Car");
    }
}
```

```java
public class Bus implements Vehicle {
    public void cleanVehicle() {
        System.out.println("Cleaning the Bus");
    }
}
```

```java
public interface Vehicle {
    void cleanVehicle();
    default void startVehicle() {
        System.out.println("Vehicle is starting");
    }
}
```

```java
public class Bus implements Vehicle {
    public void cleanVehicle() {
        System.out.println("Cleaning the Bus");
    }
    public void startVehicle() {
        System.out.println("New starting of the Bus");
    }
}
```

02 DELEGACE Kotlin

Pro zajištění zpětné kompatibility, když chceme do interface přidat novou metodu a nechceme sahat do všech implementací tohoto interface.

```kotlin
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print() //Prints 10
}
```

```kotlin
interface SoundBehavior {
    fun makeSound()
}

class ScreamBehavior(val n:String): SoundBehavior {
    override fun makeSound() = println("${n.toUpperCase()}")
}

class RockAndRollBehavior(val n:String): SoundBehavior {
    override fun makeSound() = println("I'm The King of Rock 'N' Roll: $n")
}

class KarelGott(n:String): SoundBehavior by ScreamBehavior(n)

class ElvisPresley(n:String): SoundBehavior by RockAndRollBehavior(n)

fun main() {
    val karel = KarelGott("Kavu si osladim...!")
    karel.makeSound() //KAVU SI OSLADIM...!
    val elvis = ElvisPresley("Dancin' to the Jailhouse Rock")
    elvis.makeSound() //I'm The King of Rock 'N' Roll: Dancin' to the ...
}
```

02 SEALED (ZAPEČETĚNÉ) TŘÍDY a ROZHRANÍ Java 15

```java
public sealed interface Service permits Car, Truck {

    int getMaxServiceIntervalInMonths();

    default int getMaxDistanceBetweenServicesInKilometers() {
        return 100000;
    }

}
```

```java
public abstract sealed class Vehicle permits Car, Truck {

    protected final String registrationNumber;

    public Vehicle(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }

    public String getRegistrationNumber() {
        return registrationNumber;
    }

}
```

… sealed … A permits B, C = pouze B a C mohou extendovat A

Extendovaná třída musí obsahovat přepínač *final, sealed, non-sealed*

```java
public non-sealed class Car extends Vehicle implements Service {

    private final int numberOfSeats;

    public Car(int numberOfSeats, String registrationNumber) {
        super(registrationNumber);
        this.numberOfSeats = numberOfSeats;
    }

    public int getNumberOfSeats() {
        return numberOfSeats;
    }

    @Override
    public int getMaxServiceIntervalInMonths() {
        return 12;
    }

}
```

02 INTERFACE - ROZHRANÍ

Interface (neboli rozhraní) je “předpis” pro to, jaké metody musí třída implementovat (realizovat).

Třída musí implementovat všechny metody - ne jen některé => interface tedy definuje kompaktní skupinu spolu souvisejících funkcionalit a předepisuje rozhraní kterým budeme k objektu přistupovat.

```
Interface Bicycle {
  void changeCadence(int newValue);
  void changeGear(int newValue);
  void speedUp(int increment);
  void applyBrakes(int decrement);
}
```

Java 9

Privátní metody uvnitř interface

```
public class SpecializedVengeBicycle implements Bicycle{
   int cadence = 0;
   int speed = 0;
   int gear = 1;
 public void changeCadence(int newValue) {
       cadence = newValue;
   }
   public void changeGear(int newValue) {
       gear = newValue;
   }
   public void speedUp(int increment) {
       speed = speed + increment;
   }
   public void applyBrakes(int decrement) {
       speed = speed - decrement;
   }
}
```

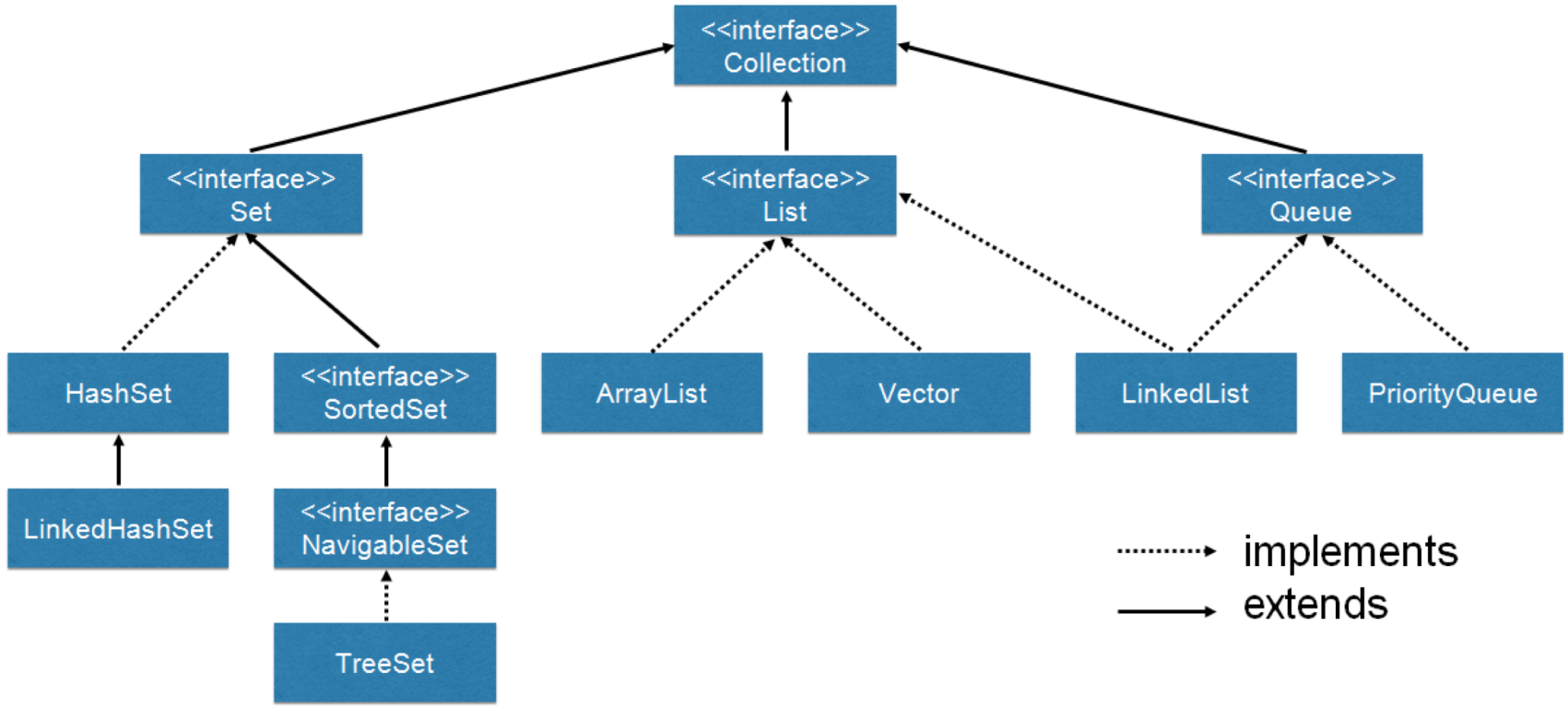
02 INTERFACE - ROZHRANÍ

Implements - třída implementuje interface (realizuje jeho metody)

Extends - interface extenduje (dědí) vlastnosti rodičovského interface

Třída může implementovat více interface najednou

Příklad rozhraní v Java Collection API



02 ABSTRACT CLASS – ABSTRAKTNÍ TŘÍDA

- Abstraktní třída je taková, kterou **nelze instanciovat (nelze od ní vytvořit objekt)**
- Uvnitř abstraktní třídy jsou implementovány metody, které pak přebírají potomci této abstraktní třídy.
- Uvnitř abstraktní třídy mohu definovat abstraktní metody, které nemají implementaci. Tedy jako bych definoval rozhraní, které pak musí implementovat potomci této abstraktní třídy

```
public abstract class AbstractEmployee {
   private String name;
   private String address;
   private int number;

   public AbstractEmployee(String name, String address, int number) {
       this.name = name;
       this.address = address;
   }

   public abstract double computePay();
   public abstract int getNumber();

   public void mailCheck() {
       System.out.println("Mailing check to"+this.name+" " + this.address);
   }
   public String getName() {
       return name;
   }
   public String getAddress() {
       return address;
   }
   public void setAddress(String newAddress) {
       address = newAddress;
   }
}
```

02 ROZDÍLY ABSTRAKTÍ TŘÍDY a INTERFACE

Interface - všechny proměnné jsou automaticky *public static final* a metody *public*.

Java 9

Privátní metody uvnitř interface

Abstraktní třída - lze definovat také proměnné, které nejsou *static* a *final* a metody mohou být *public, protected, private*

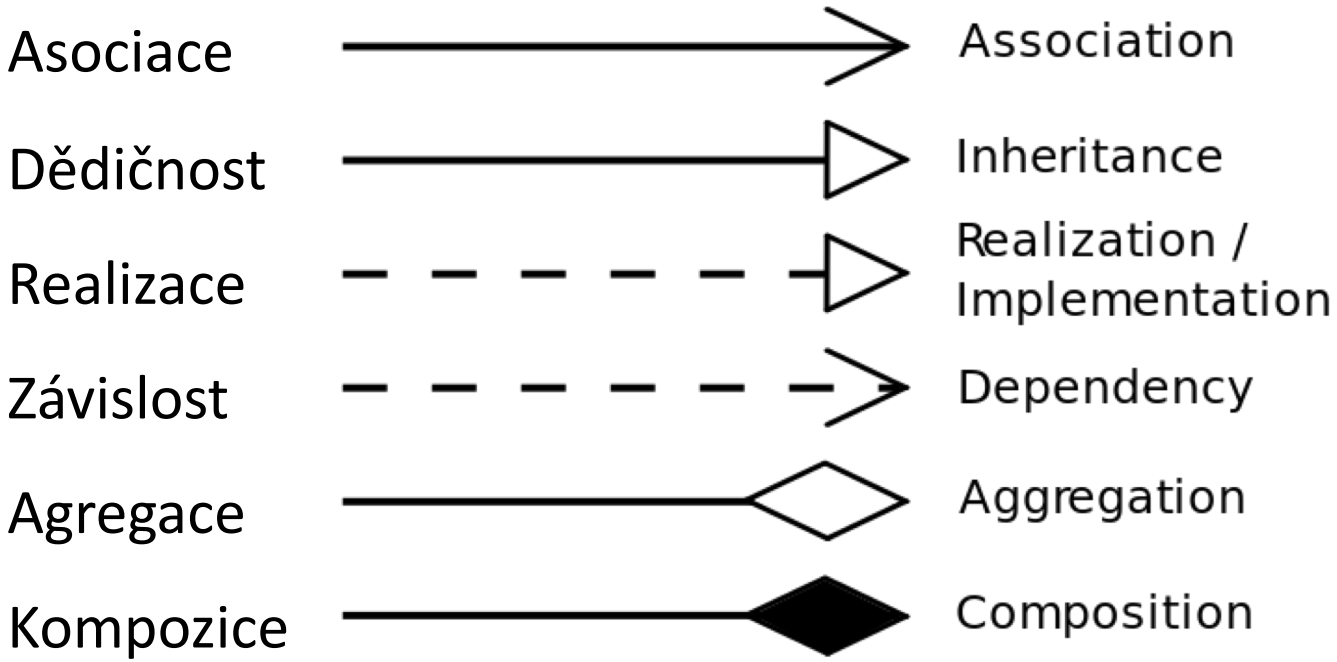
***Abstraktní třídu použijeme, jestliže*:**

1. Chci sdílet kód mezi více třídami a neexistuje “neabstraktní” předek těchto tříd
2. Třídy jsou mezi sebou úzce spjaté - sdílí mezi sebou mnoho proměnných a metod
3. Potřebujeme předepsat i metody, které neslouží k vnější komunikaci s objektem, tedy *private* a *protected*.
4. Chci předefinovat proměnné, které nejsou *static* a *final*.

***Interface použijeme, jestliže*:**

1. Vytvářím vnější rozhraní k objektům - vytvářím veřejné API ke své komponentě
2. Chci, aby i objekty z jiné class hierarchie implementovaly stejné rozhraní
3. Předepisuju pouze metody a nikoliv jejich implementaci - mým cílem není, aby třídy sdílely implementaci
4. Chci předepsat třídě, aby implementovala metody z více rozhraní

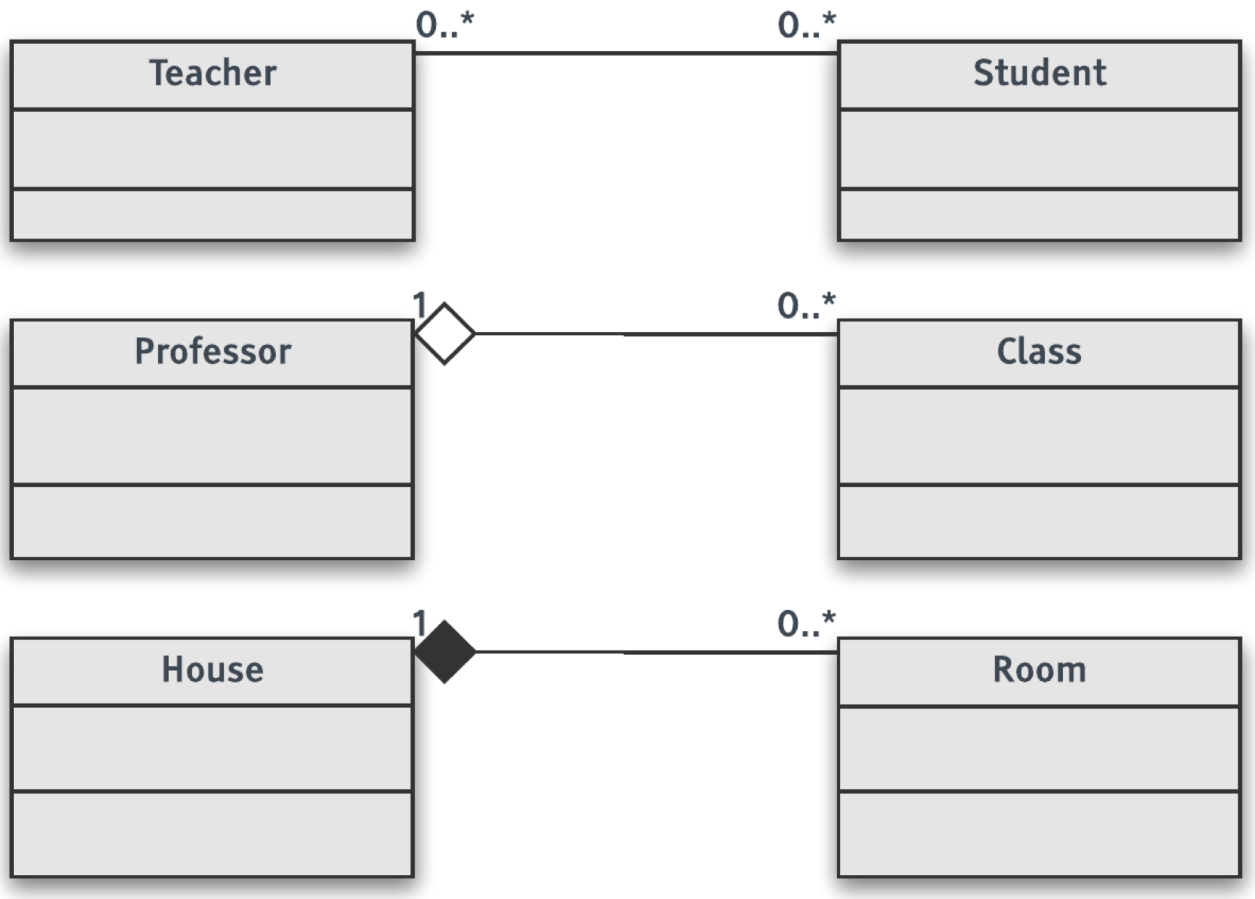
02 VZTAHY MEZI OBJEKTY



02 VZTAHY MEZI OBJEKTY

Liší se od sebe primárně sílou vazby

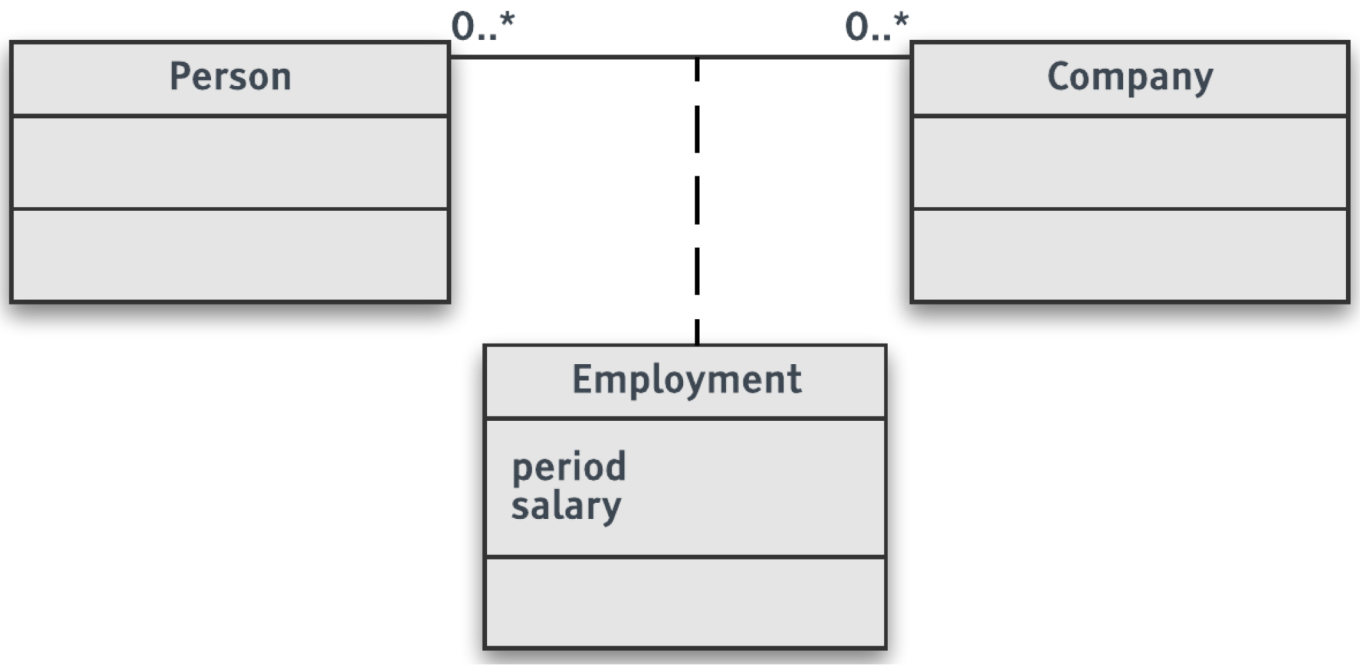
- ***Asociace*** - objekty mají zcela nezávislý životní cyklus, realizována jako proměnná držící referenci na instanci nebo proměnná na vstupu metody, jakákoliv multiplicita.
- ***Agregace*** - objekty mají zcela nezávislý životní cyklus, vlastněný objekt nemůže mít dalšího vlastníka, realizována jako proměnná držící referenci na instanci. Nemůže vytvářet cykly, multiplicita 1:1 nebo 0:N.
- ***Kompozice*** - objekty mají svázaný životní cyklus, jeden objekt vlastní druhý a s jeho zánikem i ten druhý zaniká, realizována jako proměnná držící referenci na instanci nebo zanořená (inner) třída. Nemůže vytvářet cykly, multiplicita 1:1 nebo 0:N.



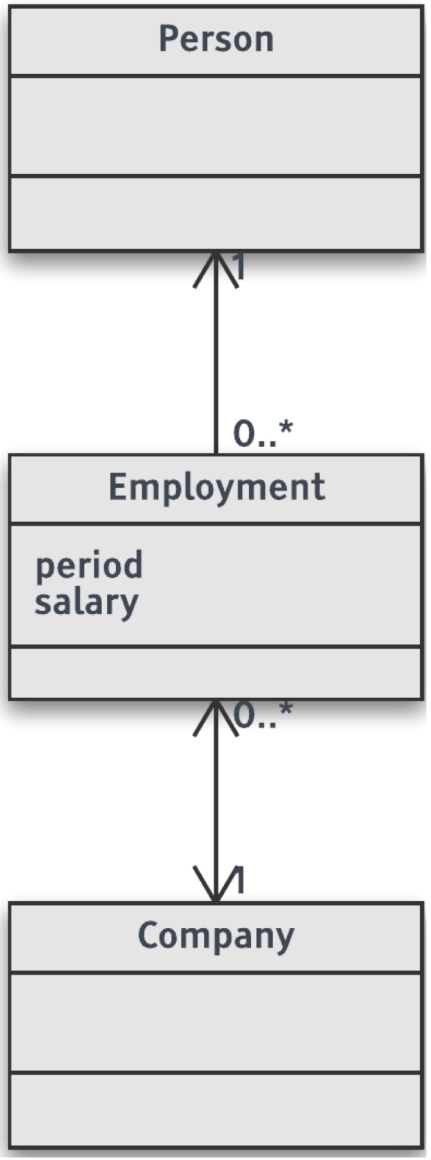
Pozn: Závislost *– nejslabší typ vazby, říkáme, že změna v jedné třídě může ovlivnit druhou. Většinou se na class diagram ani nemodelují, pokud nechceme čtenáře explicitně upozornit na něco důležitého*

02 ASOCIAČNÍ TŘÍDA

Asociační třídu potřebuji v případě, že vazby mezi objekty za dvou různých tříd mají různý stav.



02 ASOCIAČNÍ TŘÍDA



Co navigabilita? Je lepší obousměrná vazba nebo jednosměrná?

- Obousměrná vazba znamená, že při každé změně vazby z jedné strany musím upravit i druhý směr
- Jednosměrná vazba je jednodušší na správu, ale neumožňuje efektivně hledat z obou stran - jak zjistím v jakých všech objektech *Company* je *Person* zaměstnán?

```java
public class Person {
}

public class Employment {
   long salary;
   Interval period;
   Company company;
   Person person;
   public Person getPerson( ) {
       return person;
   }
   ...
}

public class Company {
   private Set<Employment> employments;
   Set <Person> getEmployees( ){
       Set<Person> employees = new HashSet<>( );
       for(Employment employment: employments){
           employees.add(employment.getPerson( ));
       }
       return employees;
   }
  ...
```

02 POLYMORFISMUS

Overloading

Vytvoření více metod se stejným jménem metody, ale různými parametry. Různými parametry je zde míněno jiný počet parametrů nebo jiné typy parametrů

- metody se stejnými parametry nemůžou mít různé návratové typy
- děje se mezi metodami v rámci té samé třídy

Overriding

Předefinování metody předka metodou potomka s identickými parametry a návratovým typem

- je možné upravit access level, aby byl méně restriktivní
- návratový typ může být potomkem původního návratového typu
- potomek nemůže vyhazovat novou výjimku
- nelze dělat override statické nebo final metody

02 POLYMORFISMUS – OVERLOADING

Různé počty parametrů

```
class Adder{
   static int add(int a,int b){return a+b;}
   static int add(int a,int b,int c){return a+b+c}
}
class TestOverloading1{
   public static void main(String[] args){
       System.out.println(Adder.add(11,11));
       System.out.println(Adder.add(11,11,11));
   }
}
```

Různé typy parametrů

```
class Adder{
   static int add(int a, int b){return a+b;}
   static double add(double a, double b){
return a+b;}
}
class TestOverloading2{
   public static void main(String[] args){
       System.out.println(Adder.add(11,11));
       System.out.println(Adder.add(12.3,12.6));
   }}
```

Různé návratové typy

```
class Adder{
   static int add(int a, int b){return a+b;}
   static double add(int a, int b){return a+b;}
}
class TestOverloading2{
   public static void main(String[] args){
       System.out.println(Adder.add(11,11));
       System.out.println(Adder.add(12,12));
   }
}
```

=> Compile Time Error: method add(int,int) is already defined in class Adder

02 POLYMORFISMUS - OVERRIDING

Bez overridingu

```
class Vehicle{
  void run(){
        System.out.println("Vehicle is running");
    }
}

class Bike extends Vehicle{

   public static void main(String args[]){
       Bike obj = new Bike();
       obj.run(); //=> Output:Vehicle is running
   }
}
```

S overridingem

```
class Vehicle{
   void run(){
        System.out.println("Vehicle is running");
   }
}

class Bike2 extends Vehicle {
   void run() {
       System.out.println("Bike is running");
   }
   public static void main(String args[]) {
       Bike2 obj = new Bike2();
       obj.run(); //=> Output:Bike is running
   }
}
```

02 POLYMORFISMUS - OVERRIDING

Při overloadingu je možné změnit i návratový typ z předka na potomka - tzv. *kovariantní typ*

```
class A{
   A get(){return this;}
}

class B1 extends A{
   B1 get(){return this;}
   void message(){
System.out.println("Covariant return type");
   }

   public static void main(String args[]){
       new B1().get().message();
   }
}
```

02 POLYMORFISMUS - OVERRIDING - PROMĚNNÉ

Overriding proměnných není podporován, vazba se vytváří v compile time na rozdíl od metod

```java
public class Parent {
    String name = "P";
    public String identify(){
        return "I am Parent with name " + name;
    }
}
public class Child extends Parent{
    String name = "C";
    public String identify(){
        return "I am Child with name " + name;
    }
}
public class Client {
    public static void main(String[] args){
        Parent p = new Child();
        System.out.println(p.name); //Prints P
        System.out.println(p.identify()); //Prints C
    }
}
```

Overriding proměnných je podporován

```kotlin
open class Parent {
    open val name: String = "P"
}

open class Child: Parent() {
    override val name: String = "C"
}

fun main() {
    val p: Parent = Child()
    println(p.name) //Prints C
}
```


21

02 KOMUNIKACE MEZI OBJEKTY

Objektové orientovaný přístup je definovaný jako zapouzdření proměnných a metod do objektů, které si mezi sebou posílají zprávy. Tzv. **message passing** je předávání zpráv mezi dvěma objekty. Příkladem je komunikace mezi objekty v Smalltalk.

Jaká je analogie z reálného života?

- *Malý Karlík se rozhodne, že napíše dopis své babičce.*
- *Vezme papír, vezme tužku, napíše dopis a odnese ho na poštu. Pak se vrátí domu a jde spát*
- *Dopis přijde babičce do její schránky*
- *Babička když každé ráno pouští psa Alíka, tak kontroluje svou schránku*
- *Jednoho rána je celá radostí bez sebe, ve schránce je dopis od jejího jediného vnoučka, vnouček ji prosí o jeho oblíbené sušenky, tak se babička sebere a ihned začne péct sušenky*

Jaká je realizace v Java pomocí standardního message passingu?

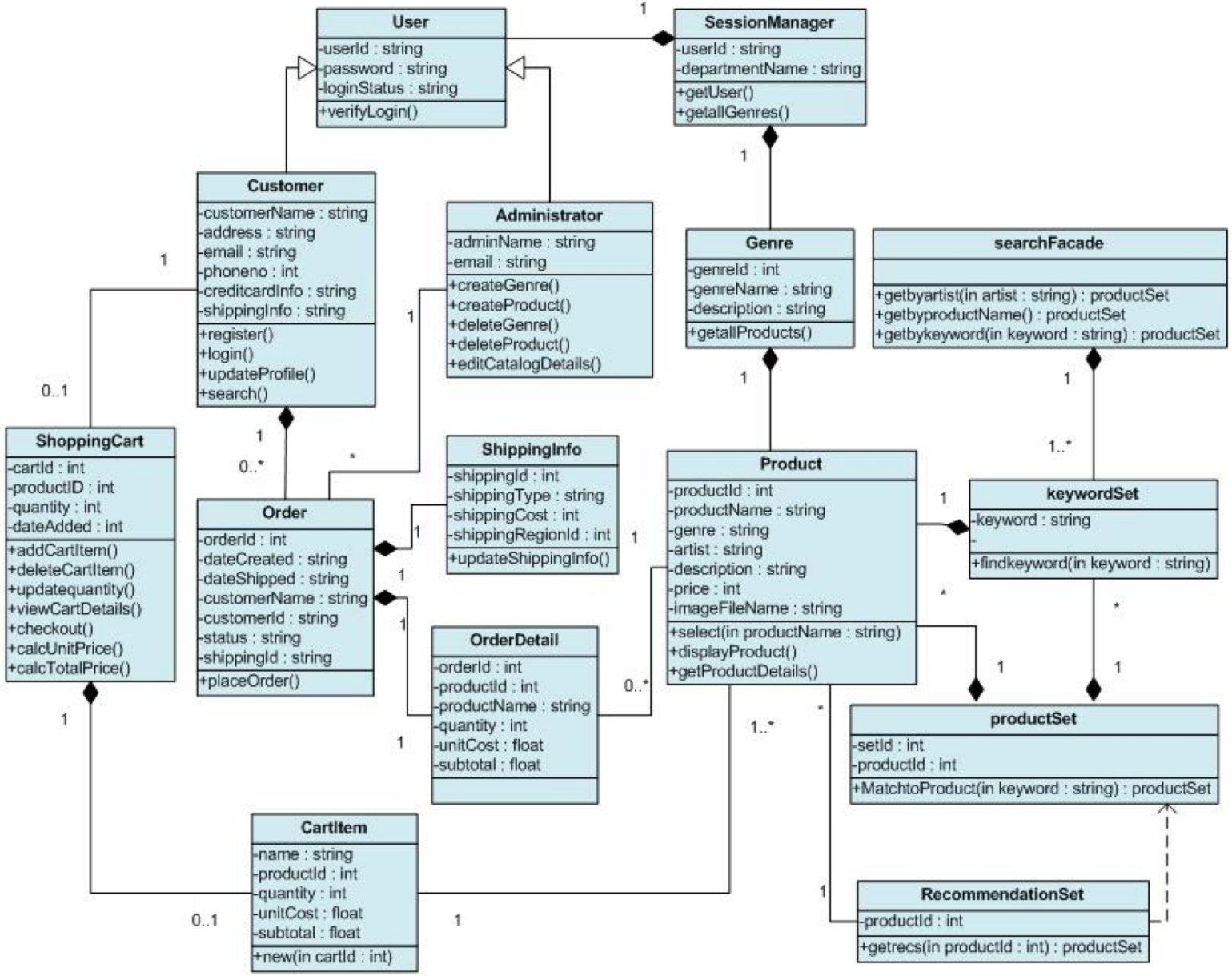
- *Hlavní program zavolá na objektu Karlik metodu sendMessage() a předá jí referenci na objekt Babicka*
- *Metoda sendMessage() v objektu Karlik zavola synchronně metodu receiveMessage() na objektu babicka, z této metody se vola další metoda bakeCookie()*
- *Objekt Karlik čeká než babička dopeče buchty, hlavní program čeká na Karlika*

Java ve skutečnosti neimplementuje opravdový message passing mezi objekty =>

- Synchronní komunikace mezi objekty se děje jednoduše přes provolání metody druhého objektu, thread je blokován
- Asynchronní posílání zpráv si musíte doprogramovat nebo využít nějakou existující

02 PŘÍKLADY SYSTÉMŮ MODELOVANÉ OOP PŘÍSTUPEM

Tento class model reprezentuje systém pro správu objednávek



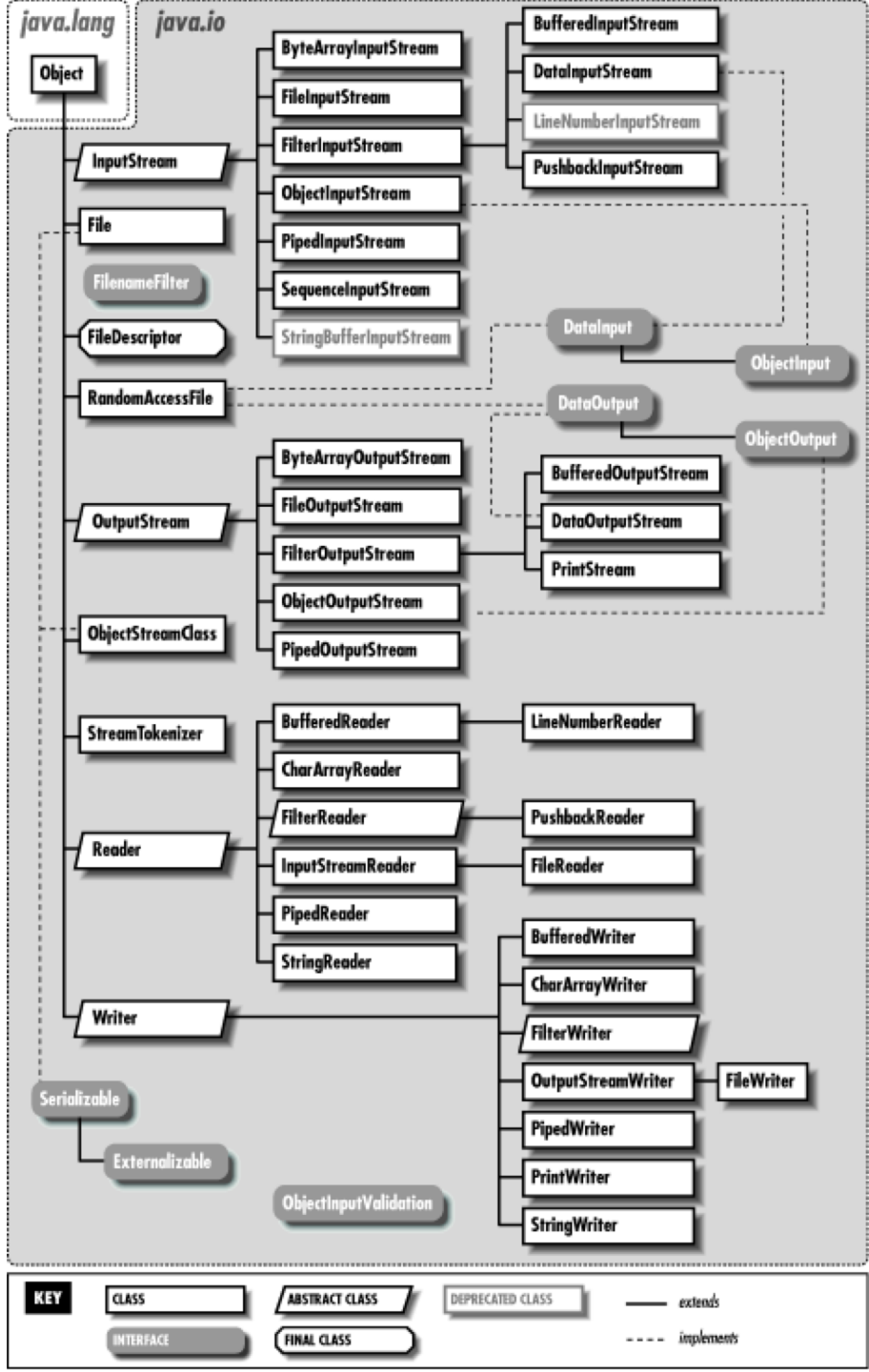
Systém vhodný pro realizaci pomocí OOP

Tento class diagram ukazuje třídy a rozhraní v package *java.io*. Ukazuje, že i Java knihovny jsou realizovány pomocí OOP.

Při debuggování Javy je zřejmá hierarchie tříd a modularizace, je jednoduše pochopitelné trasovat co se právě děje

Negativem je, že neustále procházíte desítkami vrstev tříd a jejich předků, kde v každé metodě je pár řádků kódu (stack trace se nevejde ani na jednu stránku)

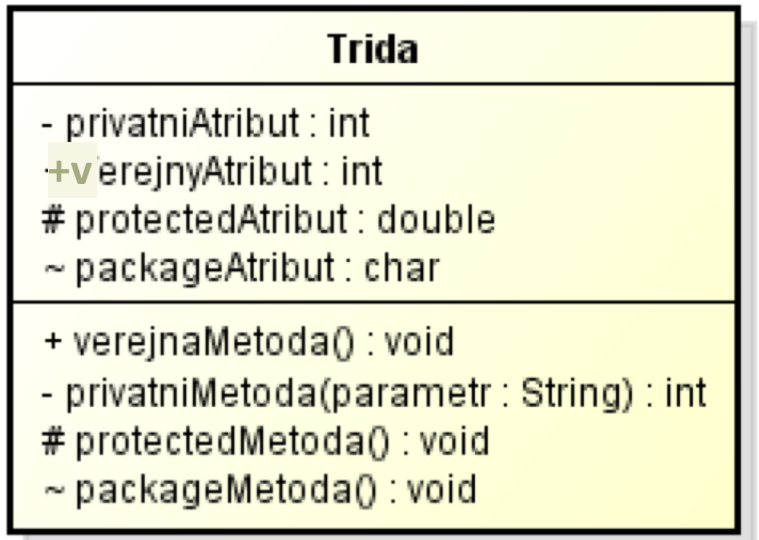
Java je realizovaná pomocí OOP

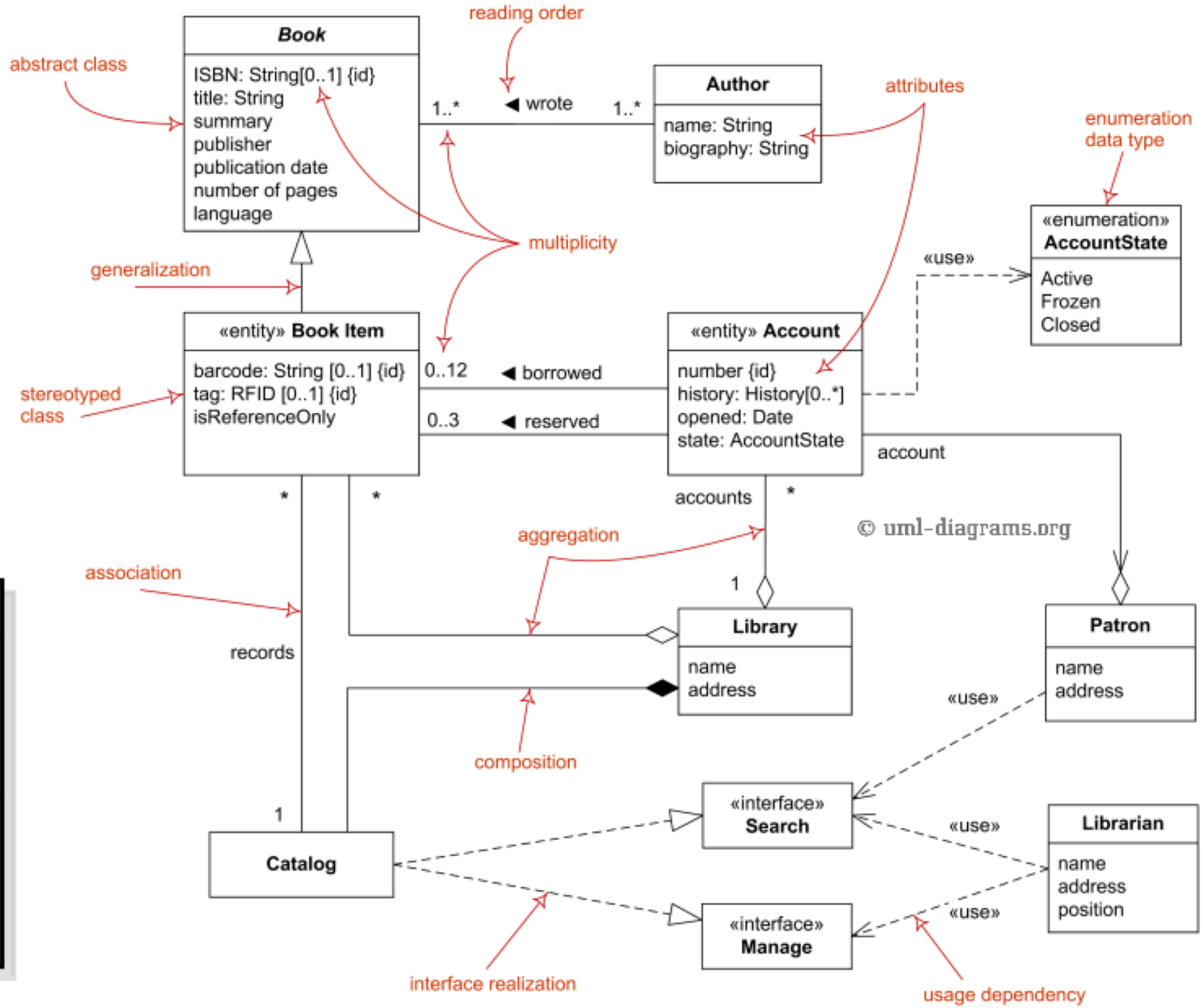


Class diagram

Class diagram je UML diagram pro grafické zobrazení tříd, jejich vlastností a vztahů mezi nimi

Je možné jít do většího detailu a zobrazit i metody a úroveň přístupu

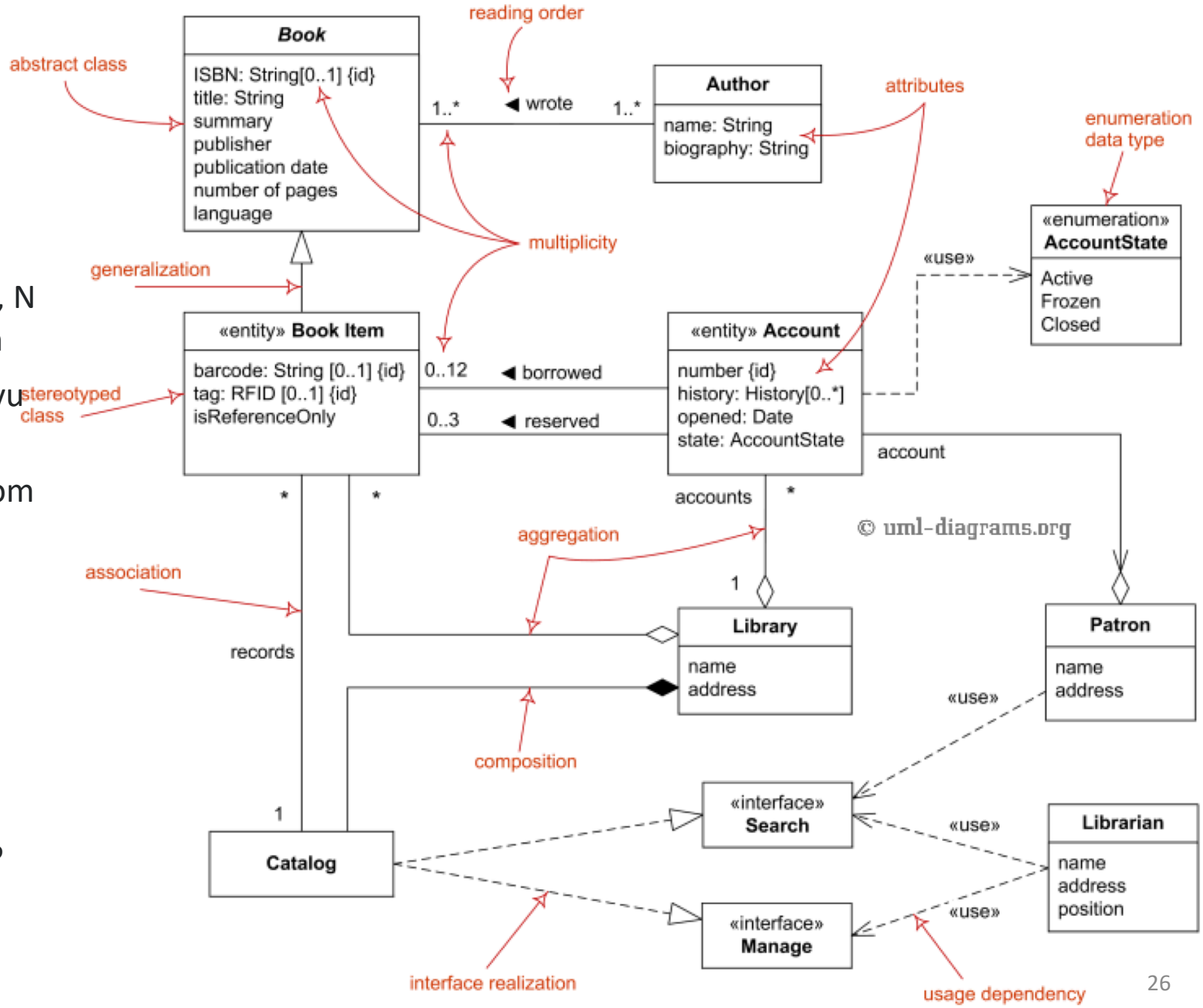




Tento systém reprezentuje knihovnu:

- Mám knihovnu, ta spravuje N účtů, N knih a disponuje jedním katalogem
- U účtů evidujeme v jakém jsou stavu a pro jakého člena jsou otevřeny
- V knihovně se půjčují knihy, abychom je mohli elektronicky evidovat a vyhledávat, tak jsou opatřeny čarovým kódem a RFID
- Ke katalogu můžeme přistupovat pomocí rozhraní pro vyhledávání a správu

Systém vhodný pro realizaci pomocí OOP



02 JE LEPŠÍ KOMPOZICE NEBO DĚDIČNOST?

- Dědičnost má výhodu v tom, že zavádí pravidla a minimalizuje duplicity v kódu
- Dědičnost je nicméně extrémně silná vazba, kterou zavádím do svého systému. Strukturální zásahy do hierarchie tříd ve chvíli, kdy už mám implementován komplexní systém, jsou extrémně pracné
- Pro nějaké problémy nelze rigidní hierarchii tříd ani sestavit
- Při návrhu je dobré si dopředu dobře rozmyslet, které struktury a pravidla jsou natolik pevné, že je můžu fixovat pomocí dědičnosti.
- Tam, kde vím, že budu v budoucnosti potřebovat flexibilitu, tak radši volím kompozici
- Kompozice má nevýhodu, že někdy končí duplicitami v kódu a je mnohem více benevolentnější - tedy “hloupý programátor” může udělat značné škody

02 SOLID

Single Responsibility Principle

Třída má pouze jeden účel a jednu zodpovědnost a všechny její metody by měly sloužit k plnění tohoto účelu. Proč? Čím méně bude třída plnit účelů, tím bude méně důvodů do této třídy zasahovat. Např. je vhodné rozdělit formátování a generování reportů do různých tříd.

Open/Closed Principle

Třídy by měly být otevřené pro rozšiřování a uzavřené pro modifikaci. V existujících třídách by měl probíhat pouze bug fixing, ale nová funkcionalita by měla přicházet do potomků třídy. Důvodem je minimalizace zásahů do hotového kódu

Liskov Substitution Principle

Objekt je vždy možné nahradit objektem z třídy potomka. Kód pak nemusí kontrolovat s jakým konkrétním podtypem pracuje, Důvodem je, abychom nemuseli v kódu provádět nepříjemné kontroly typu a řešit různé side efekty.

Interface Segregation Principle

Více klient specifických rozhraní je lepší než jedno víceúčelové rozhraní. Důvodem je, že to více specifických rozhraní vede k menšímu couplingu. Např. PersistenceManager implementuje rozhraní *DBReader* a *DBWriter*

Dependency Injection Principle

Závislosti mezi objekty mají být na abstrakcích (abstraktní třídy, parent třídy, rozhraní), nikoliv konkrétních implementacích

Je lepší dědičnost nebo kompozice?

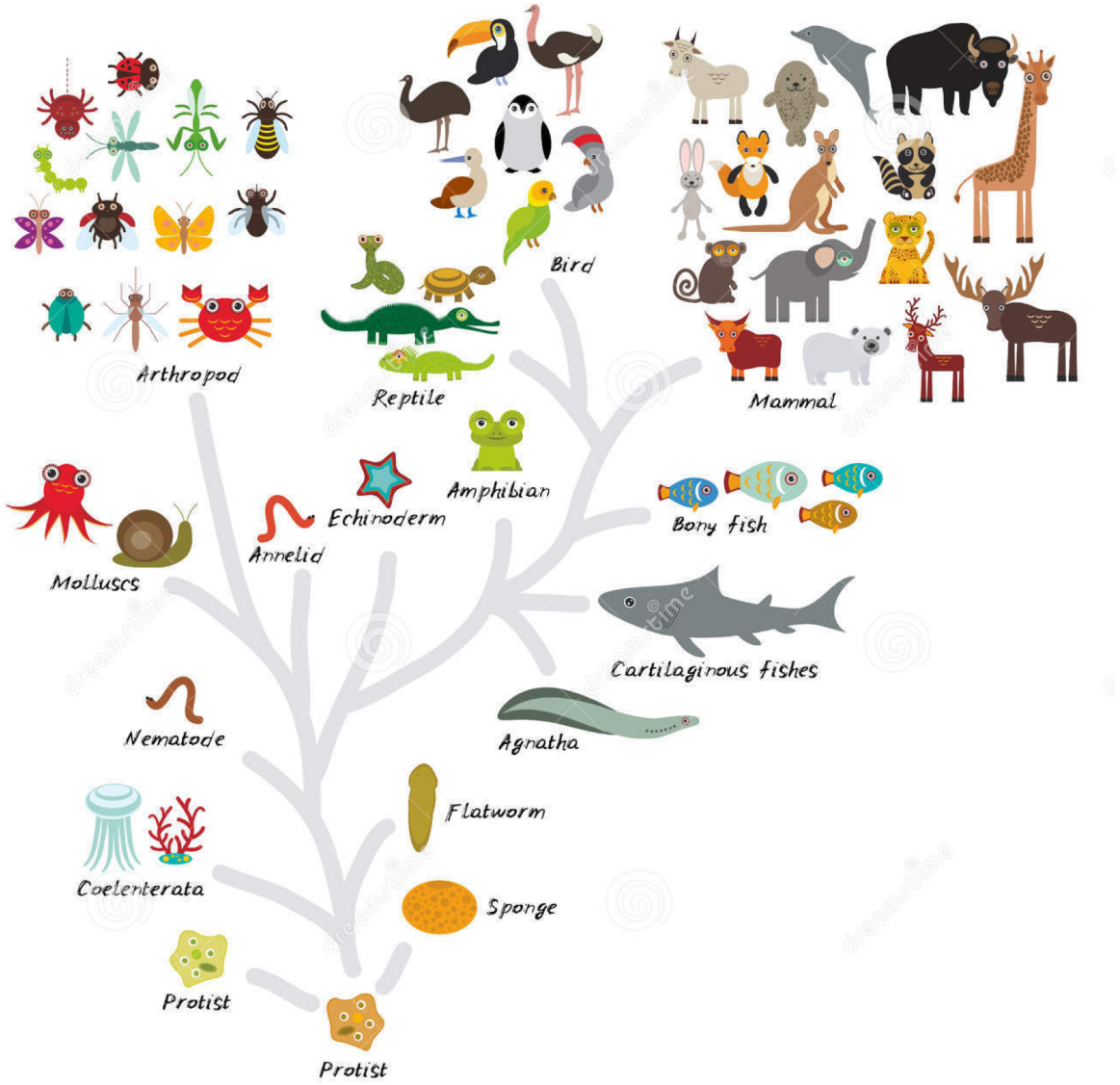
<= ... a chci implementovat toto

… co se stane, když budu chtít přenést vlastnosti z jednoho místa hierarchie tříd do jiné části u hodně komplexního systému?

02 JE LEPŠÍ KOMPOZICE NEBO DĚDIČNOST?

Mám následující class hierarchii =>

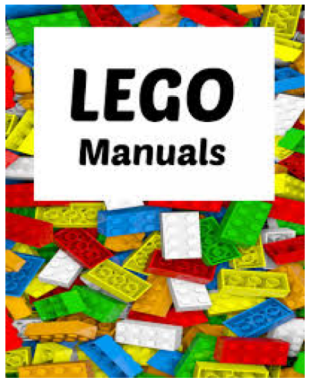
… ne příliš podobná zvířata jsou od sebe v class hierarchii vzdálena



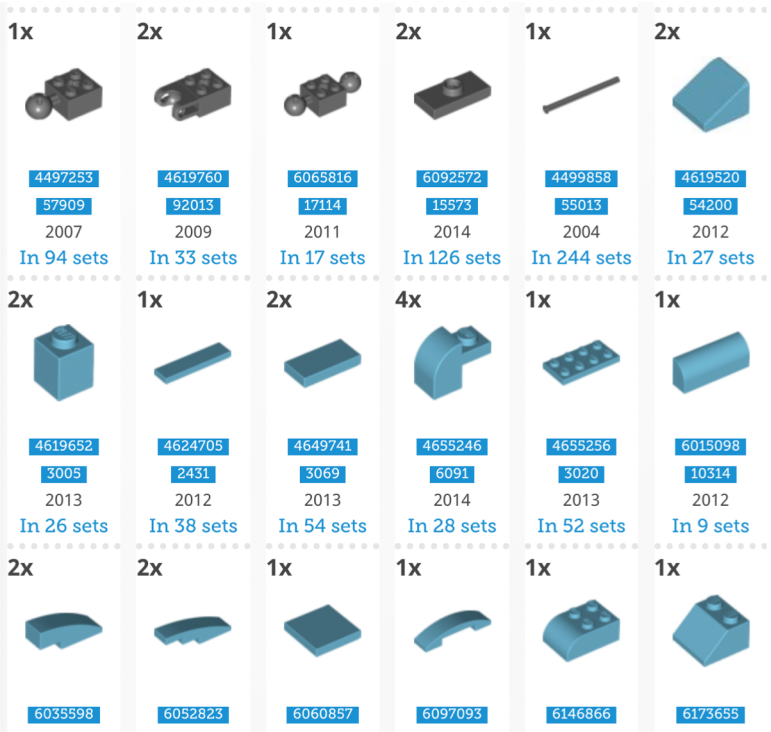
02 JE LEPŠÍ KOMPOZICE NEBO DĚDIČNOST?

Co je Lego?

Specifikace podle které dělam kompozici



Kostky, ze kterých dělám kompozici



02 SPRÁVNÁ REPREZENTACE

- Absence struktury (zde objektové reprezentace) je menší zlo než špatná struktura
  - Pozor zejména na nesprávné relace, jejich směr a multiplicitu
- Správná objektová reprezentace je co nejbližší reálnému světu ve kterém žijeme
- Pokud neimplementujeme reálný systém, tak by reprezentace měla být co nejbližší systému, který realizujeme.

Když programujete tuto počítačovou hru, tak po přečtení knihy na OPP budete intuitivně vytvářet složitou class hierarchii, která reprezentuje typologii "potvor" a typologii překážek. Pak začnete vyhodnocovat pozici SuperMaria podle absolutní pixel pozice přečtené z obrazovky. Obojí je špatně. Místo složité typologie vám stačí obálka (šířka a výška) předmětu, id ikony/textury pro zobrazení a flagy definující chování. Místo odečítání pozice z pixelů na obrazovce realizujete uvnitř programu fyzikální model, který počítá aktuální pozice a kolize s předměty; a zobrazení je jen projekce do aktuálního zobrazovacího zařízení v aktuálním rozlišení.