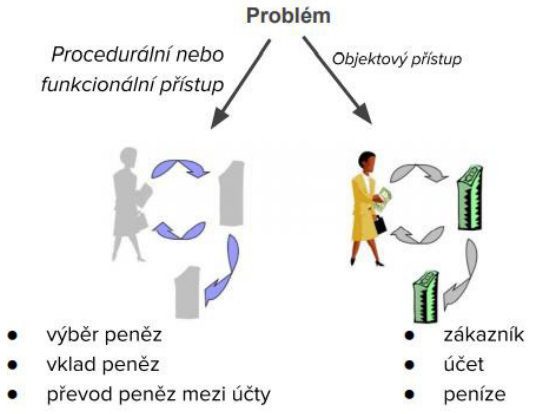
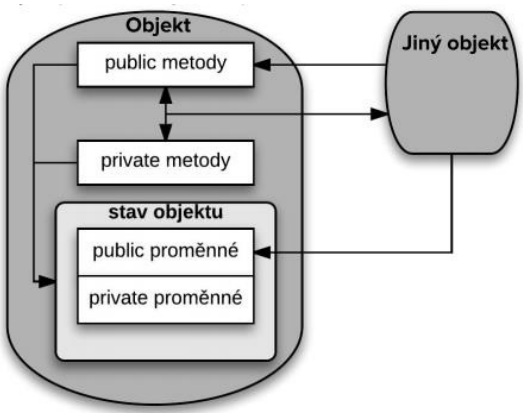
Objektově orientovaný přístup I

Objekty, třídy

- **objektový přístup** = problém se snažím řešit tak, že jej kompletně strukturuji do objektů, které mezi sebou komunikují
- objekty volím tak, aby co nejvíce odpovídali objektům z reálného světa

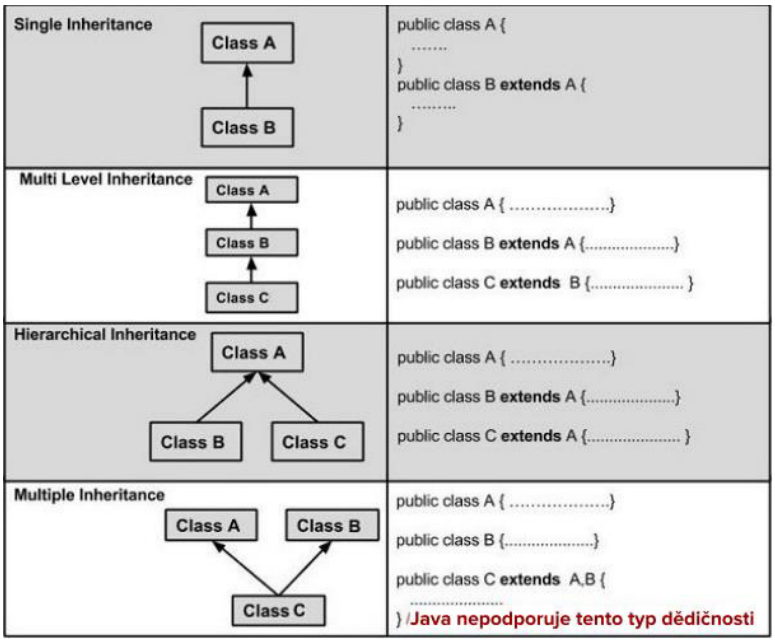




- objekt zapouzdřuje **proměnné** (reprezentují jeho stav) a **metody** (pracují se stavem)
- uvnitř objektu má vše přístup ke všemu
- zvně objektu lze volat pouze veřejné metody a přímo pracovat jen s veřejnými proměnnými
- hráči na mapě např. nenastavuji pozici přímo, ale posouvám pomocí pokynů

- **třída** = předpis pro tvorbu a chování objektů (instancí)
  - z každé třídy mohu vytvořit N instancí, ty se liší pouze stavem

- **dědičnost** – třída potomka přebírá vlastnosti předka
  - potomek může přepoužívat proměnné a metody předka, pokud je nepředefinuje



Interface a abstraktní třídy

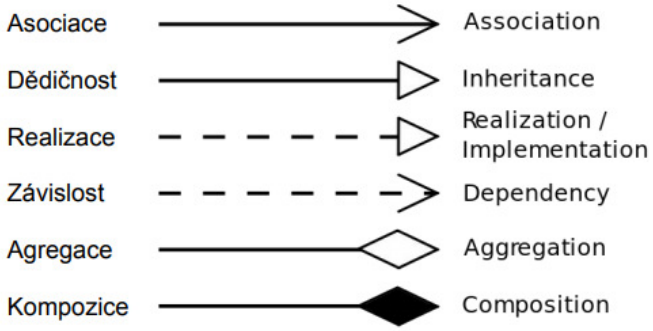
- **interface (rozhraní)** = předpis, jaké metody musí třída implementovat
  - třída musí implementovat VŠECHNY metody
  - definuje kompaktní skupinu spolu souvisejících funkcionalit
  - předdefinuje rozhraní, kterým budu k objektu přistupovat
  - třída může implementovat více interface najednou

- **abstraktní třída** = třída, kterou nelze instanciovat
  - uvnitř jsou implementované metody, které potomci přebírají
  - lze definovat abstraktní metody (nemají implementaci)
  - podobné definici rozhraní, které musí potomci implementovat

- **rozdíly**:
  - **interface** – všechny proměnné jsou automaticky public static final, metody public
    - **použití**:
      - vytvářím vnější rozhraní k objektům (veřejné API ke komponentě)
      - chci, aby objekty z jiné class hierarchy implementovaly totéž rozhraní
      - předepisuji jen metody, nikoliv jejich implementaci
      - chci, aby třída implementovala metody z více rozhraní

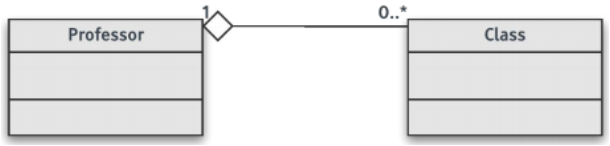
  - **abstraktní třída** – lze definovat proměnné, které nejsou static a final, metody mohou být public/protected/private
    - **použití**:
      - chci sdílet kód mezi třídami, neexistuje neabstraktní předek
      - úzce spjaté třídy – sdílení mnoha proměnných a metod
      - předepisuji i metody, které neslouží k vnější komunikaci (private,…)
      - chci předefinovat proměnné, které nejsou static a final

Rozdíl mezi asociací, agregací a kompozicí



- liší se primárně silou vazby
- **asociace**
  - objekty mají zcela nezávislý životní cyklus
  - proměnná drží referenci na instanci
  - popř. proměnná na vstupu metody
  - jakákoliv multiplicita

- **agregace**
  - objekty mají zcela nezávislý životní cyklus
  - HAS-A vztah
  - vlastněný objekt nemůže mít dalšího vlastníka
  - proměná drží referenci na instanci
  - nemůže vytvářet cykly
  - multiplicita 1:1 nebo 0:N



- **kompozice**
  - objekty mají svázaný životní cyklus
  - OWNS vztah (objekt vlastní objekt)
  - se zánikem zaniká i vlastněný
  - proměnná drží referenci na instanci
  - popř. inner class
  - nemůže vytvářet cykly
  - multiplicita 1:1 nebo 0:N

- **asociační třída**
  - potřebuji v případěm že vazby mezi objekty ze dvou různých tříd mají různý stav
  - obousměrná vazba – při změně vazby z jedné strany musím upravit i druhý směr
  - jednosměrná vazba – jednodušší na správu, ale neumožňuje efektivně hledat z obou stran

Message passing

- = předávání zpráv mezi dvěma objekty
- příkladem komunikace mezi objekty ve Smalltalk

- **analogie**:
  - Karlík chce napsat dopis babičce.
  - Vezme papír a tužku, napíše dopis. Odnese jej na poštu.
  - Dopis přijde babičce do schránky.
  - Babička každé ráno venčí psa a zkontroluje schránku.
  - Jednou najde ve schránce dopis. Přečte si jej a začne péct sušenky.

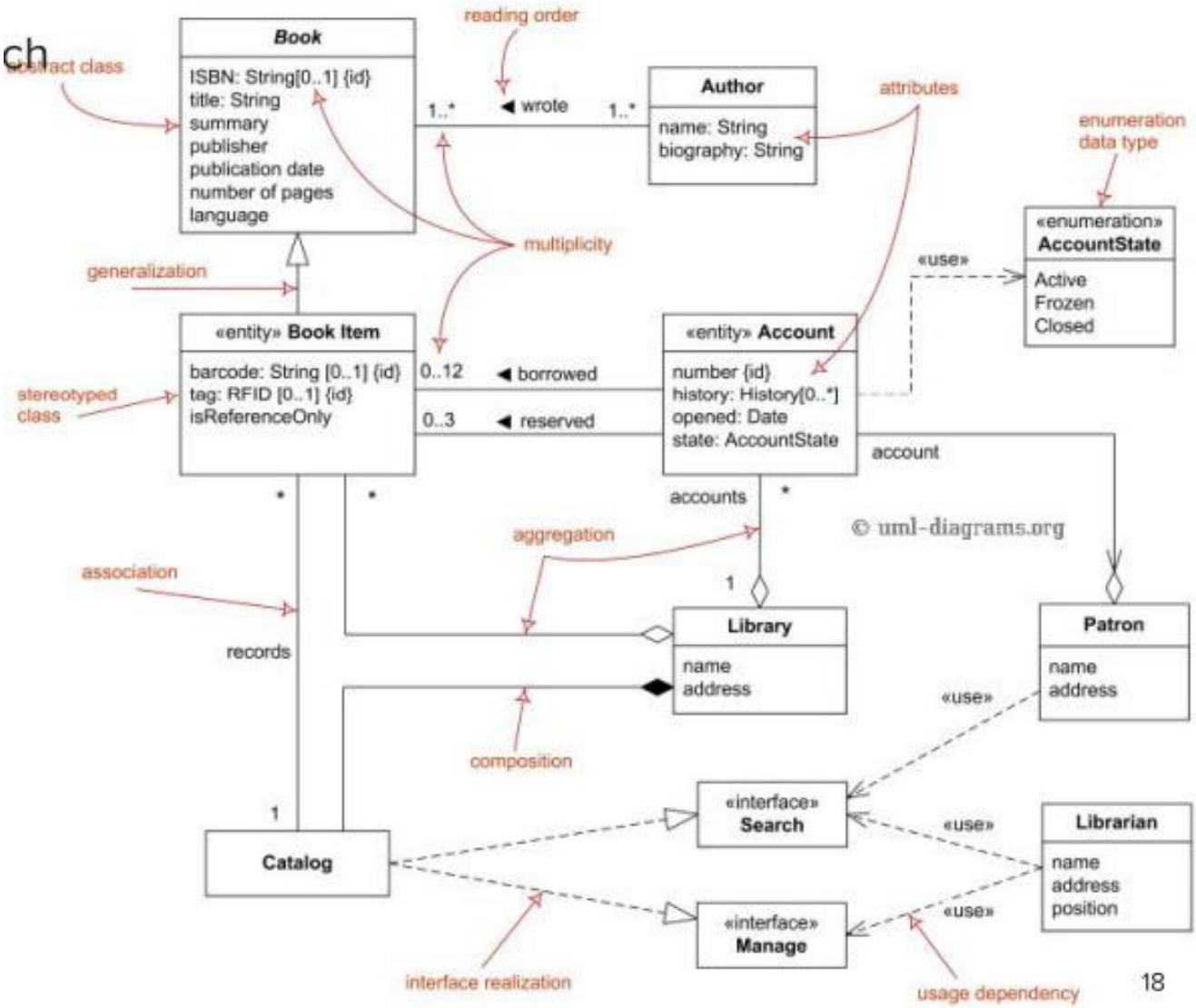
- **realizace v Javě:**
  - Hlavní program zavolá Karlik.sendMessage(), předá metodě referenci na babičku.
  - Karlik.sendMessage() zavolá synchronně metodu Babicka.receiveMessage. Z té metody se volá další metoda – bakeCookies().
  - Objekt Karlik čeká, než babička dopeče buchty, hlavní program čeká na Karlíka.

- Java ve skutečnosti neimplementuje opravdový message passing mezi objekty
  - **synchronní komunikace** – jednoduše přes provolání metody druhého objektu, thread je blokován
  - **asynchronní komunikace** – nutné doprogramnovat, popř. využít existující nadstavbu

Class diagramy a příklady systémů modelovaných pomocí OOP

- **class diagram** = UML diagram pro grafické zobrazení tříd, jejich vlastností a vztahů mezi nimi
- lze i větší detail – metody a úroveň přístupu
- nevhodné pro modelování pomocí OOP je např. realizace neuronové sítě
- vhodné jsou knihovní systémy, systém pro správu objednávek,…

- **příklad**:
  - 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 vyhledávat, tak jsou opatřeby čárovým kódem a RFID.
  - Ke katalogu můžeme přistupovat pomocí rozhraní pro vyhledávání a správu.



Dědičnost versus kompozice

- **dědičnost** má výhodu v tom, že zavádí pravidla a minimalizuje duplicity v kódu
- je to nicméně extrémně silná vazba
    - strukturální zásahy do hierarchie implementovaného komplexního systému jsou extrémně pracné
    - pro některé problémy nelze ani rigidní hierarchii tříd sestavit
- při návrhu dobré promyslet, které struktury/pravidla jsou tak pevné, že je mohu takto fixovat

- tam, kde pravděpodobně budu potřebovat flexibilitu → **kompozice**
    - má ale nevýhodu, že někdy končí duplicitami v kódu
    - mnohem více benevolentnější – „hloupý programátor“ může nadělat škody

SOLID

= návrhové principy, jak psát dobře udržovatelný OOP kód

Single Responsibility Principle (= jedna odpovědnost)

- třída má jen jeden účel a jednu zodpovědnost
- všechny její metody by měli sloužit k tomuto účelu
- důvod - méně účelů → méně důvodů do ní zasahovat
- např. vhodné rozdělit formátování a generování reportů do různých tříd

Open/Closed Principle (= otevřenost/uzavřenost)

- třídy by měly být otevřené pro rozšiřování, uzavřené pro modifikaci
- jen přidání kódu, ne zasahování do existujícího
- v existujících třídách jen bug fixing, nová funkcionalita v potomcích
- důvod - minimalizace zásahů do hotového kódu

Liskov Substitution Principle (= princip zaměnitelnosti)

- objekt vždy možno nahradit objektem z třídy potomka
- kód pak nemusí kontrolovat, s jakým konkrétním podtypem pracuje
- důvod – absence nutnosti provádět typové kontroly a řešit side-efekty

Interface Segregation Principle (= oddělení rozhraní)

- více specifických rozhraní je lepší než jedno víceúčelové
- důvod – menší coupling
- např. PersistenceManager implementuje DBReader a DBWriter

Depencency Inversion Principle (= obrácení závislostí)

- abstrakce nezávisí na detailech, ale detaily na abstrakci
- high-level moduly by neměly záviset na low-level modulech
    - obojí by mělo záviset na abstrakcích
- důvod – HL moduly se lépe přepoužívají, ignorují-li implementační detaily LL modulů
- např. dependency injection; rozdělení HL modulů a LL modulů do různých packages

Polymorfismus

- v rámci třídy mám metody, které mají stejné jméno (jiné parametry), ale jiné chování
- ve třídě potomka a předka mám metody mající stejné jméno (a parametry), ale jiné chování

Overloading (přetěžování)

- = vytvoření více metod se stejným názvem, ale různými parametry
- různé parametry = různé typy či počet
- metody se stejnými parametry nemohou mít různé návratové typy
- jen mezi metodami v rámci téže třídy

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));
   }}
```

Overriding (přepisování)

- = předefinování metody předka metodou potomka (identické parametry i návratový typ)
- je možné upravit access level (méně restriktivní)
- návratový typ může být potomkem původního návratového typu
- potomek nemůže vyhazovat novou výjimku
- nelze přepsat statické či final metody

Co se děje bez overloadingu

```
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
   }
}
```

Po úpravě

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

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

- lze změnit i návratový typ z předka na potomka (= kovariantní typ)