Pokročilé informační systémy

Objektový model dat

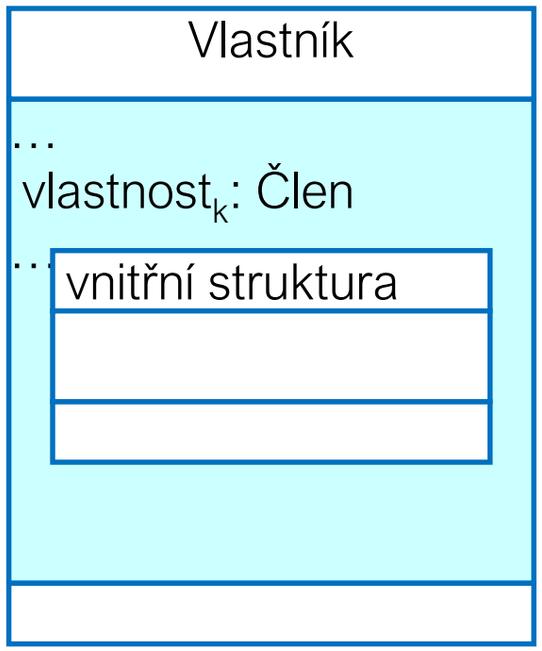
prof. Ing. Tomáš Hruška, CSc.
Ing. Radek Burget, Ph.D.
burgetr@fit.vutbr.cz

Objektový datový model

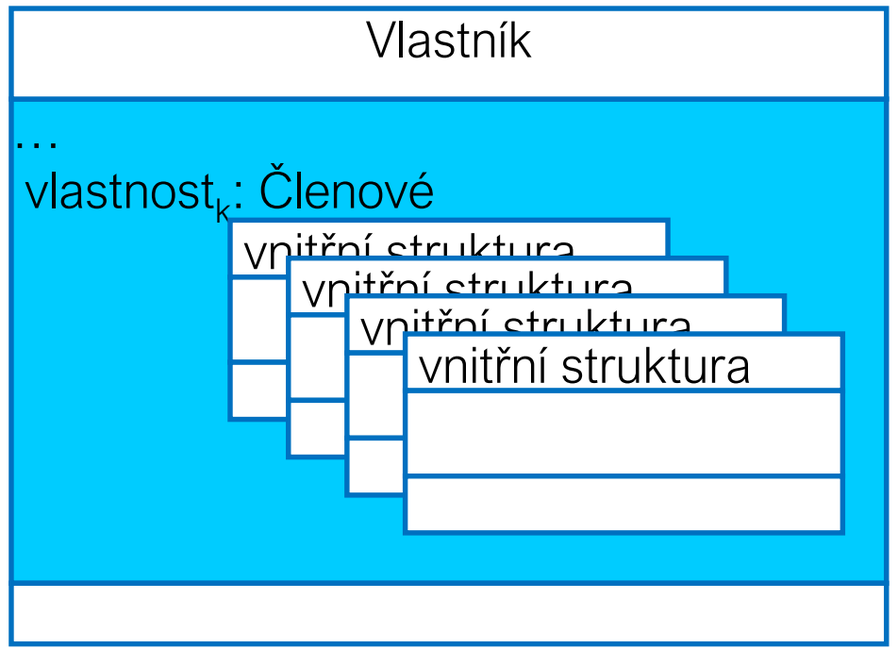
- Motivace: v aplikacích obvykle modelujeme data objektově
  - Objektově-orientované modelování v návrhu IS, UML
- Data reprezentovaná pomocí konceptů objektově orientovaného modelování
  - Třídy, objekty (instance)
- Vztahy (reference)
  - Na rozdíl od relačních databází (nemluvě o NoSQL)

Struktura objektů a vztahy

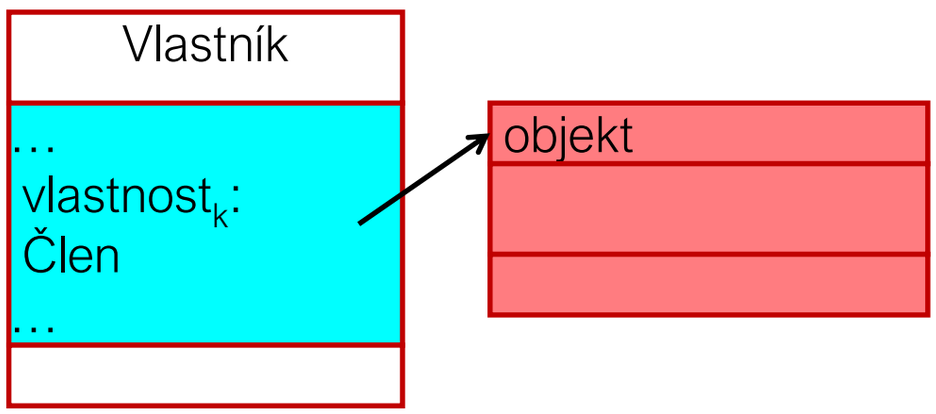
Zanoření a vztahy



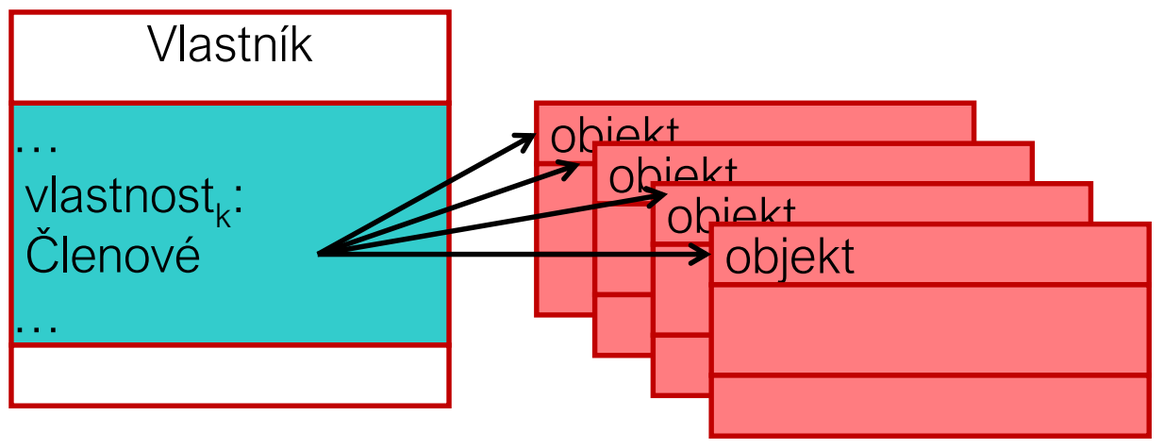
1. člen je jedinou prostou strukturou = vnoření hierarchie struktur



2. členové jsou v kolekci prostých struktur = vytváření hierarchie struktur s vnořenými kolekcemi



3. člen je jediným objektem = vytváření vztahu typu 1:1



4. členové jsou v kolekci objektů = vytváření vztahu typu 1:N

Vztahy kontra strukturalizace

	struktura	kolekce
vnoření v rámci obálky (strukturovaná data uvnitř)	jmenný prostor nižší úrovně (přístup přes tečku)	prostor přístupný operacemi kolekce
Vztah (k objektu vně)	1:1	1:N

Jmenný prostor nižší úrovně

concept TYPD [Data=Value]

…

end concept

concept TYPB

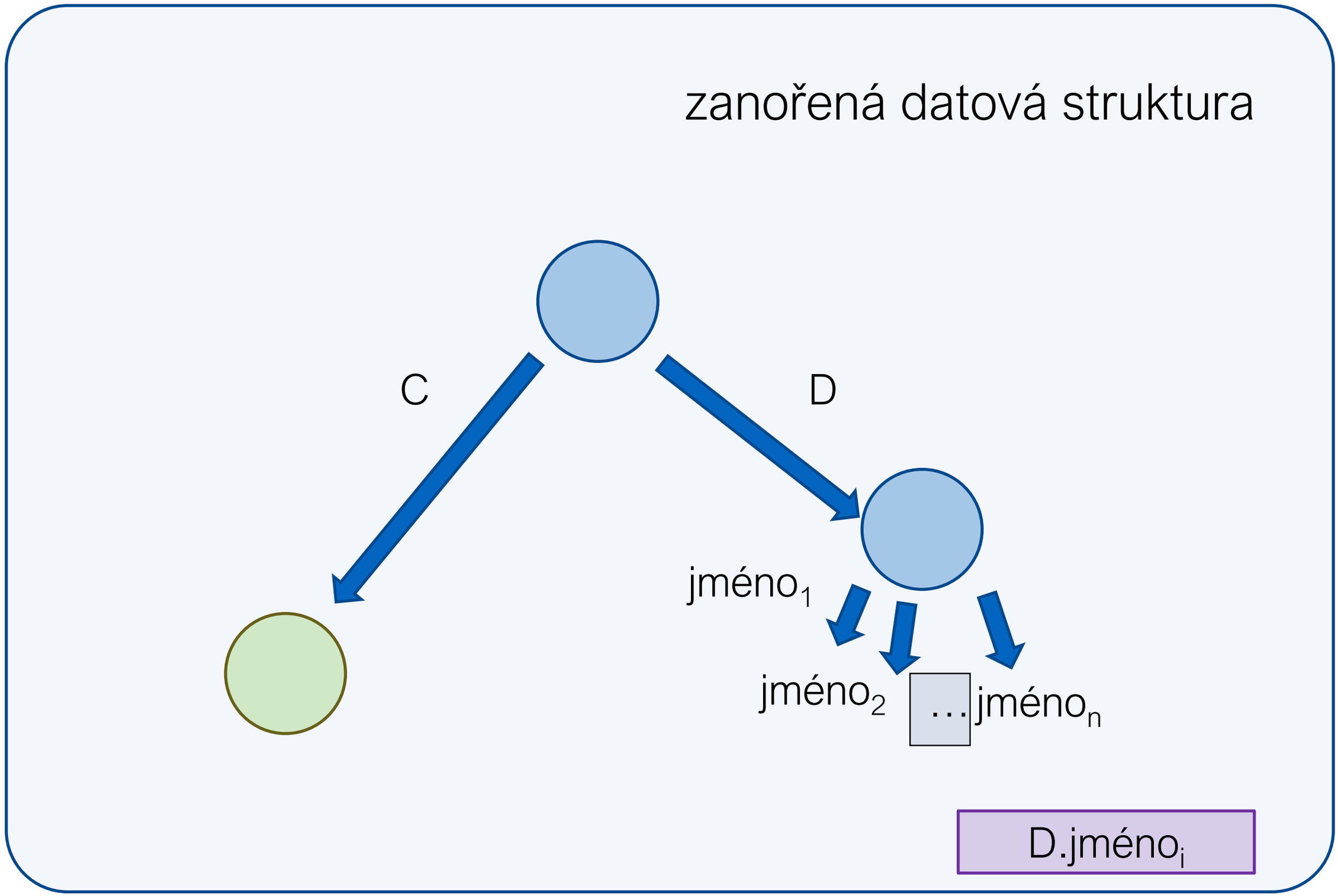
properties

C: **integer**

D: TYPD

end concept

Jmenný prostor nižší úrovně



Prostor přístupný operacemi kolekce

concept TYPB/TYPYB [Data=Value]

properties

C: **integer**

D: TYPD

end concept

concept ZANORENA

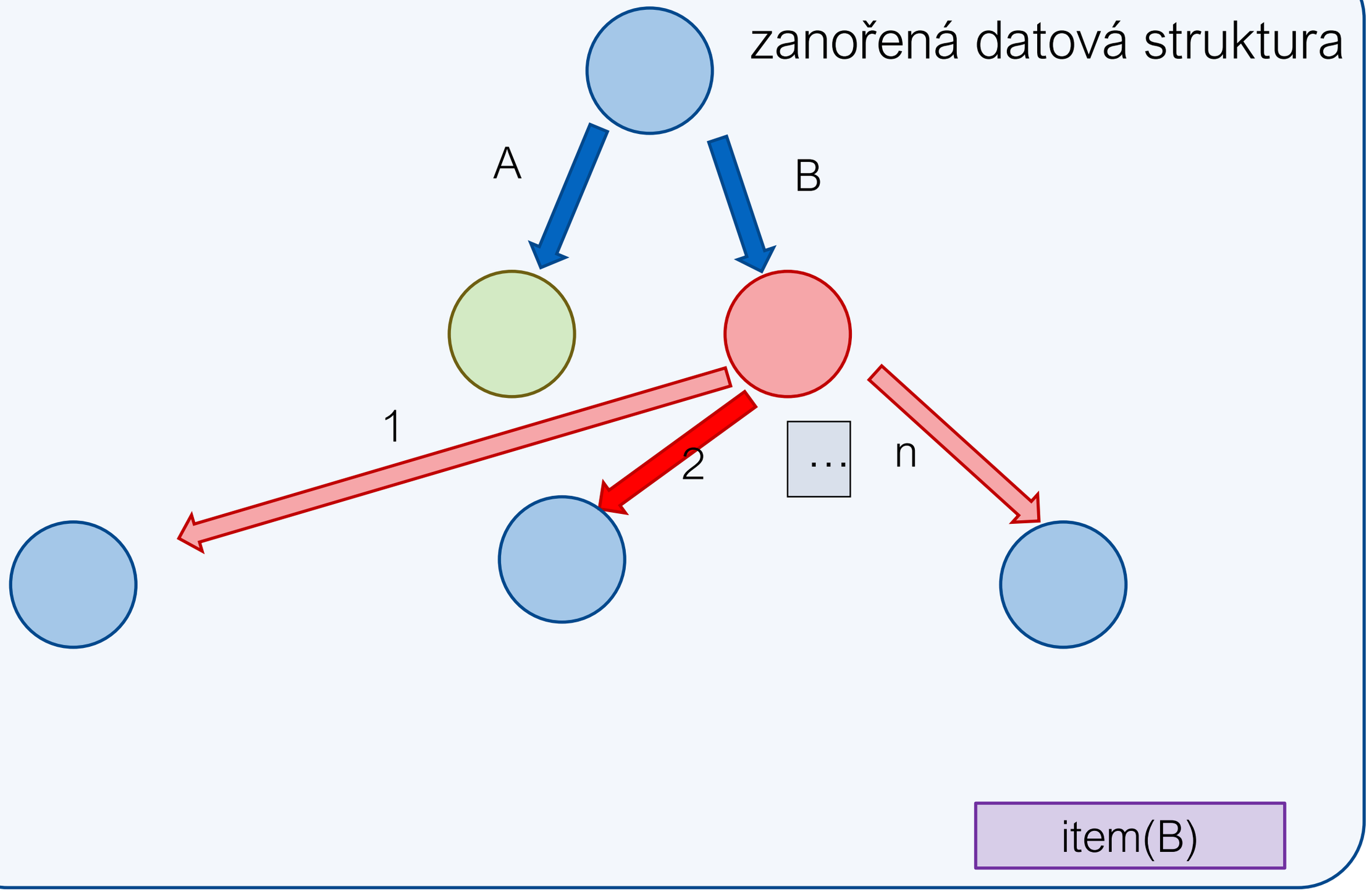
properties

A: **integer**

B: TYPYB

end concept

Prostor přístupný operacemi kolekce



Vztah 1:1

concept CLEN [Data=Ref]

properties

C: **integer**

D: **…**

…

end concept

concept VLASTNIK

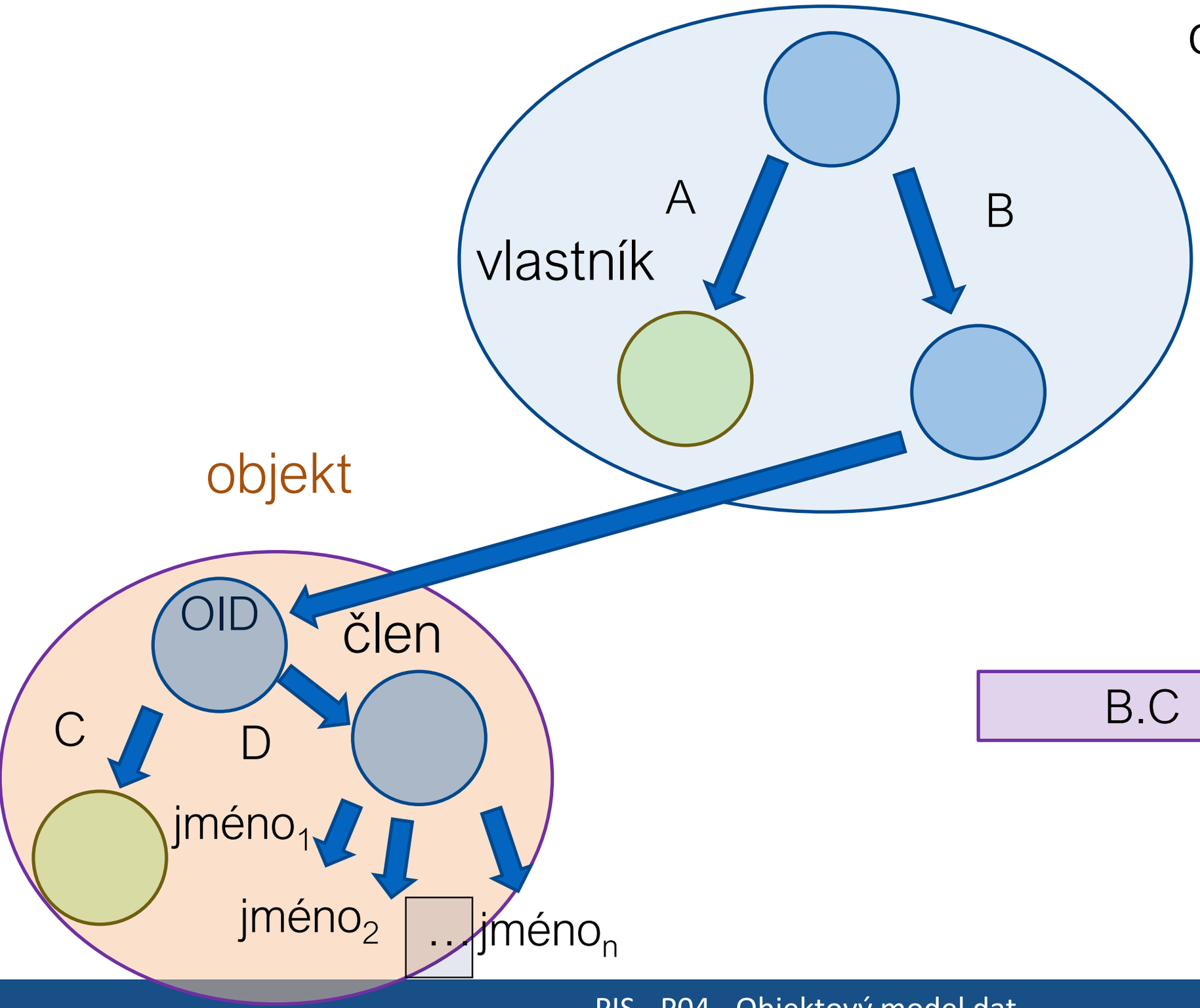
properties

A: **integer**

B: CLEN

end concept

Vztah 1:1


odkazované objekty
A
B
vlastník
odkaz
objekt
OID
člen
C
D
B.C
jméno1
jméno2
…
jménon

Vztah 1:N

concept VLASTNIK

properties

A: **integer**

B: CLENOVE

end concept

concept CLEN/CLENOVE [Data=Ref]

properties

C: **integer**

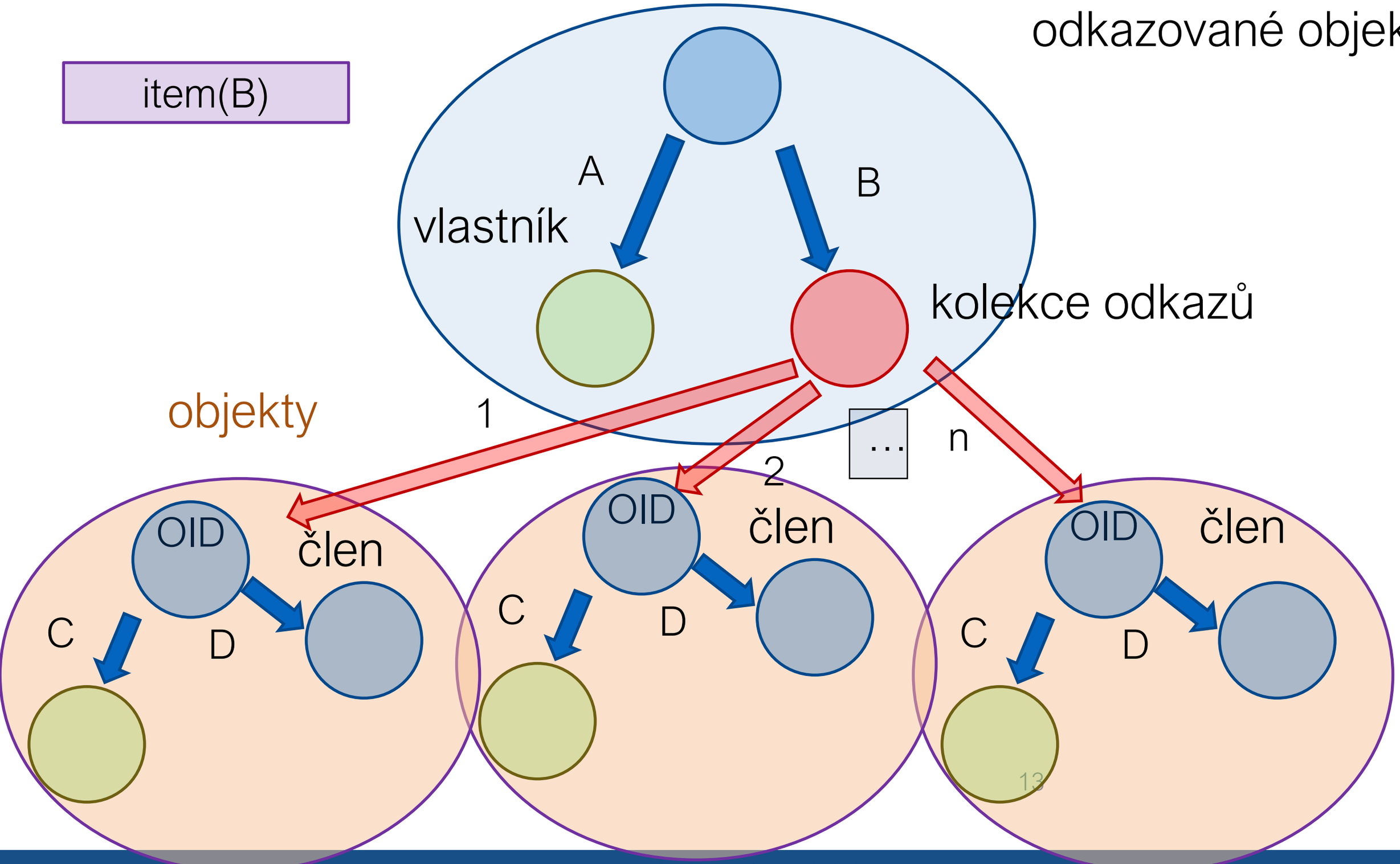
D: **…**

…

end concept

Vztah 1:N

item(B)



Příklad vnořené prosté struktury

concept Adresa/Adresy [Data=Value]

  properties

  vlastnosti adresy

end concept

concept PrvekSAdr/PrvkySAdr

  properties

    Adresat: **string**

    Adresy: Adresy

    Adresa: Adresa

end concept

Inverzní vztahy

- Častým modelovaným případem je situace, kdy je požadováno, aby ***vytvoření vztahu V z objektu A na objekt B vyvolalo rovněž vytvoření vztahu W z objektu B na objekt A***.
- Podobně při zrušení vztahu V z objektu A na objekt B musí dojít i ke zrušení vztahu W z objektu B na objekt A.

Inverzní vztahy

- Tuto situaci vyjádříme zápisem atributu **Inverse** ke vztahu V. Datový typ vztahu V určuje, na který objekt vztah povede. Hodnota atributu Inverse udává jméno vztahu W, který má být v objektu B udržován jako inverzní.

Příklad inverzních vztahů

concept A
  properties
    …
    V: B [Inverse=W]
    …
end concept

concept B
  properties
    …
    W: A
  …
end concept

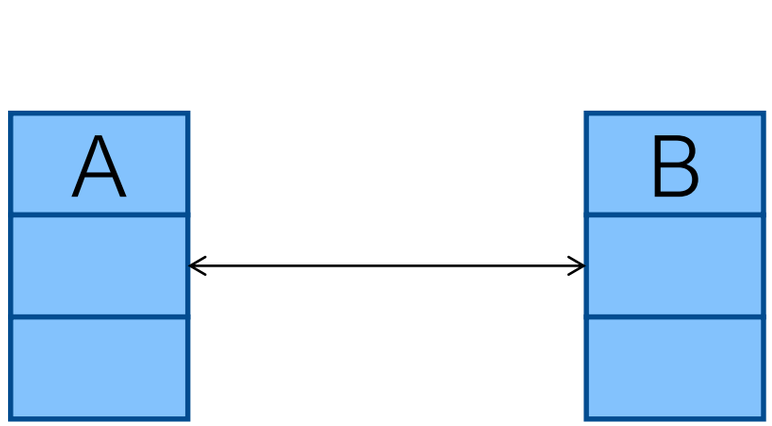
Typy inverzních vztahů

- Inverzní vztahy mohou být jak typu 1:1, tak typu 1:N.
- Je však potřeba si uvědomit poněkud rozdílné použití jména kolekce zde, v atributu Inverse, nežli při definici vztahu typu 1:N.
- Je-li použit atribut Inverse u vztahu 1:N, znamená to, že ***inverzní vztah bude vytvářen s každým prvkem příslušné kolekce***, nikoliv se vztahem, který by byl vlastností kolekce samotné.

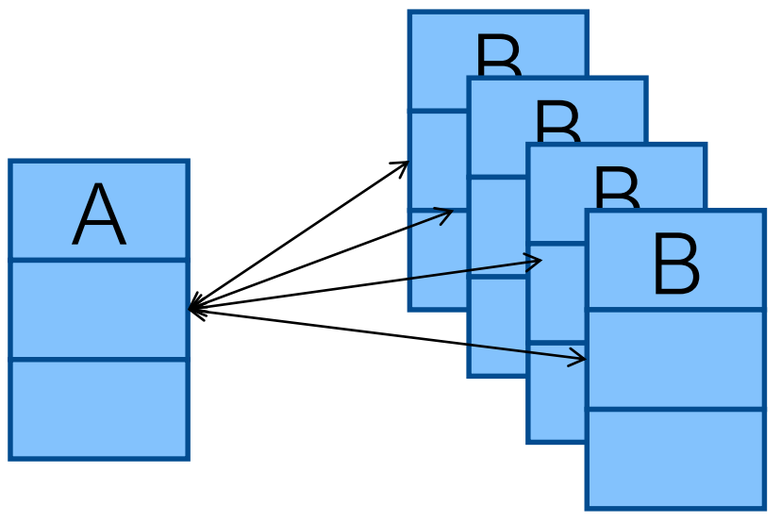
Oboustranné inverzní vztahy

- Inverzní vztahy ***nemusejí být vždy oboustranné***.
- Nicméně nejčastějším případem bývají oboustranné inverzní vztahy, kdy popisované vlastnosti inverze vztahu platí jedním i druhým směrem. Tři možné situace oboustranných inverzních vztahů jsou na následujícím slajdu

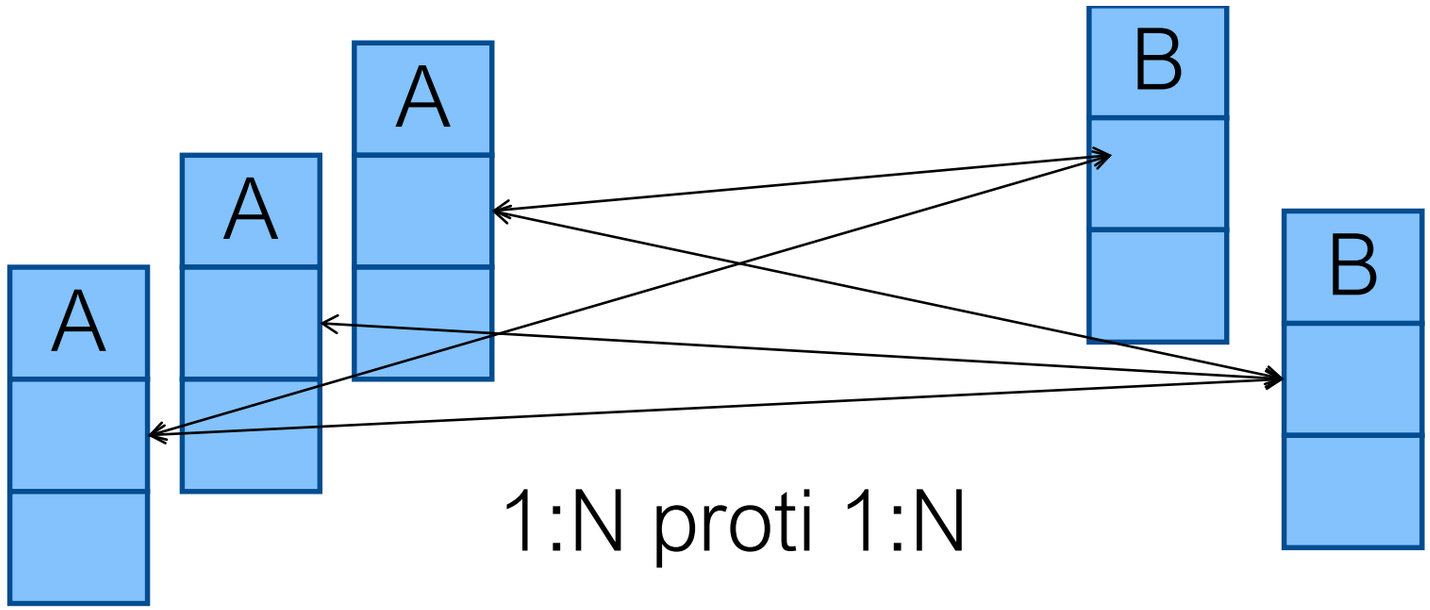
Inverzní vztahy



1:1 proti 1:1



1:N proti 1:1 a naopak



1:N proti 1:N

Příklad

concept PrvekSAdr/PrvkySAdr
properties
Adresat: **string**
Adresy: Adresy [Inverse=CiAdresa]
Adresa: Adresa [Inverse = Adresy]
end concept

concept Adresa/Adresy
properties
Ucel: DruhAdr
CiAdresa: PrvekSAdr
Adresat: **string**
ObsahAdr: ObsahAdr
end concept

Příklad

- Dva typy objektů.
  - Prvním z nich je objekt modelující prvek s adresou, v němž se vyskytuje vlastnost Adresy typu vztah 1:N na prvek kolekce Adresy.
  - Zde je definována inverze tohoto vztahu vzhledem ke vztahu CiAdresa objektu Adresa.
- Inverze vztahu je ***oboustranná***. Znamená to, že v objektu Adresa a jeho vlastnosti CiAdresa je definována inverze typu 1:1 na vztah Adresy objektu PrvekSAdr.

Generalizace a specializace (dědičnost)

Dědičnost – vazby mezi typy objektů

- Vazby ***mezi typy*** struktur. Všechny možné vazby diskutované zde se vyskytují pouze separátně mezi stejnými typy struktur, tedy mezi:
  - ***objekty*** a
  - ***prostými strukturami***.
- Nebudeme uvažovat vazbu mezi typem prosté struktury a typem objektu. Rovněž vazby mezi výčtovými typy a kolekcemi neexistují.

Dědění

- Uvažujme dva obecně různé typy struktur A a B.

concept A/AA
  properties
    $vlastnost_{A1}$
    $vlastnost_{A2}$
    $vlastnost_{A3}$
    …
end concept

concept B/BB
  properties
    $vlastnost_{B1}$
    $vlastnost_{B2}$
    $vlastnost_{B3}$
    …
end concept

Dědění

- Vlastnosti struktury A a struktury B jsou ***obecně různé***. To znamená, že jednou krajní situací je, že
  - ***obě struktury jsou typově zcela stejné*** a druhou, že
  - ***jsou zcela různé***.
- Mezi tím je možno nalézt mnoho situací, kdy se struktury částečně shodují co do některých vlastností, jmen vlastností apod.

Diference

- Pokud ***mají struktury společné rysy***, bývá často výhodné vyjádřit typ struktury B pomocí typu struktury A. K tomu můžeme použít tří druhů popisu ***rozdílu typů*** (***diferencí***):
  - ***přidávání*** nové vlastnosti ke stávajícím vlastnostem typu A,
  - ***modifikaci*** (***upřesňování***) stávající vlastnosti typu A a
  - ***zrušení*** (***vypouštění***) vlastnosti typu A.

Definice typu B z A

- Definici typu struktury B z A můžeme potom provést výrokem:
- Typ B ***obsahuje všechny vlastnosti*** typu A, avšak ***jsou do něj přidány nové vlastnosti*** D, E, F…, ***jsou upraveny vlastnosti*** G,H,I,… následujícím způsobem a vlastnosti J,K,L … ***byly zrušeny***.
- Toto nazveme ***děděním z A do B***.

Předek a následník

- Pokud definujeme typ určením diferencí, pak tento způsob definice nazýváme ***děděním***. Typ A nazýváme ***předkem*** a typ B ***následníkem***.
- Pokud vzájemné odvozování typů má více kroků, pak typ A, z něhož byl typ B přímo odvozen se nazývá ***přímý předek*** a typ B ***přímý následník***. Pokud odvození proběhlo v několika krocích označujeme typ A pouze slovem předek a typ B následník.
- Jde o binární relace na množině typů struktur a relace předek a následník jsou ***tranzitivními uzávěry relace*** přímého předka a následníka.

Více přímých předků

- Přímých následníků je obecně více vždy
- Podobně můžeme postupovat pokud je přímých předků více. Definice potom zní následujícím způsobem:
- Typ B obsahuje všechny vlastnosti ***typů X, Y, Z***, …, avšak jsou do něj přidány nové vlastnosti D, E, F…, jsou upraveny vlastnosti G,H,I,… následujícím způsobem a vlastnosti J,K,L … byly zrušeny.

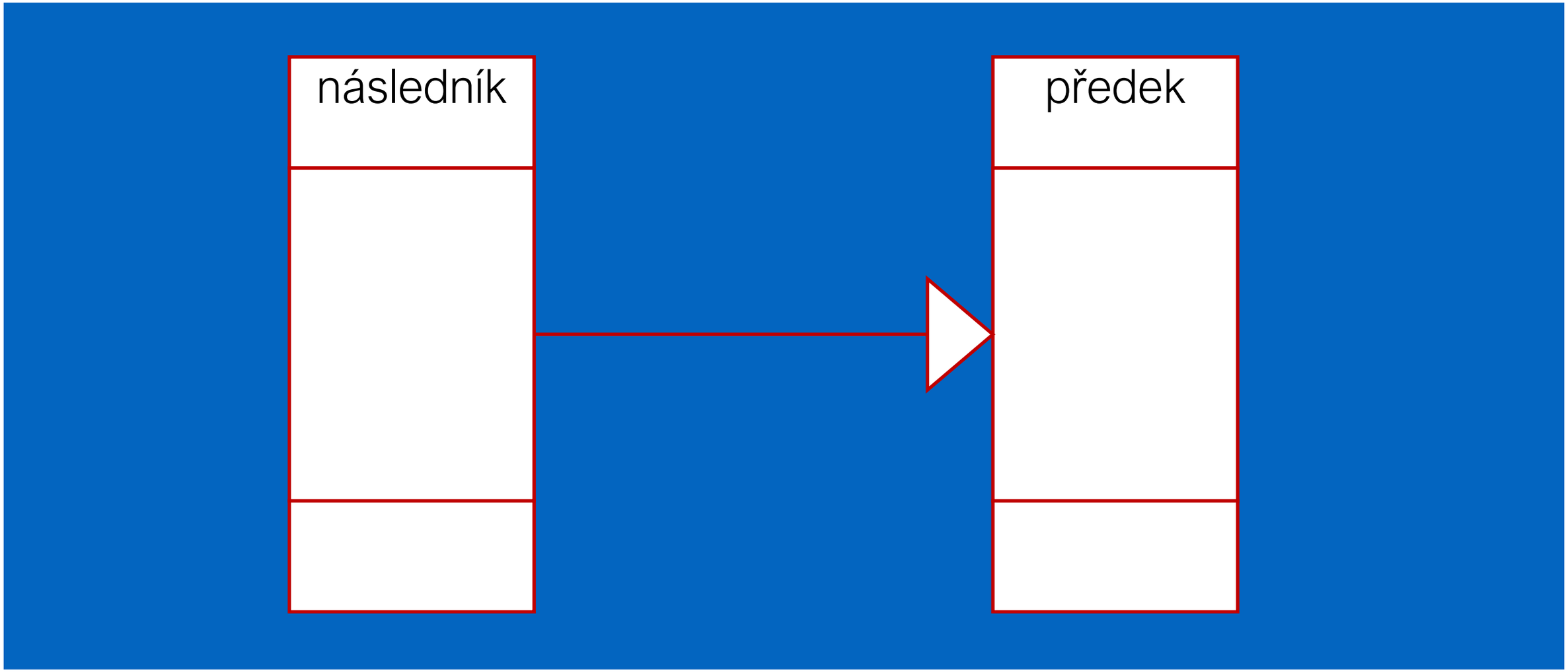
Technika odvozování

- Používáme-li definici typů odvozováním z jiných typů, vzniká hierarchická uspořádaná struktura typů.
- Při přidávání nového typu dochází pak k jeho zařazování do hierarchie typů. Hledá se vždy ***přímý předek***. Existují různé možnosti zařazení nového typu, např.:
  - průchodem hierarchie nalézt vhodného předka,
  - vytvořit typ zcela nezávislý na jiných již existujících typech,
  - restrukturalizovat stávající hierarchii typů, čímž získáme vhodného předka,

Generalizace a specializace

- Při budování hierarchického uspořádání typů lze použít při návrhu modelu dvojí postup:
  - Výběrem a sdílením společných charakteristik do nadřízených typů dochází ke ***generalizaci***.
  - Přidáváním nových tříd a doplňováním unikátních vlastností dochází ke ***specializaci***.

Zobrazení dědičnosti



- Ve schématu podle UML jde šipka ve směru generalizace.

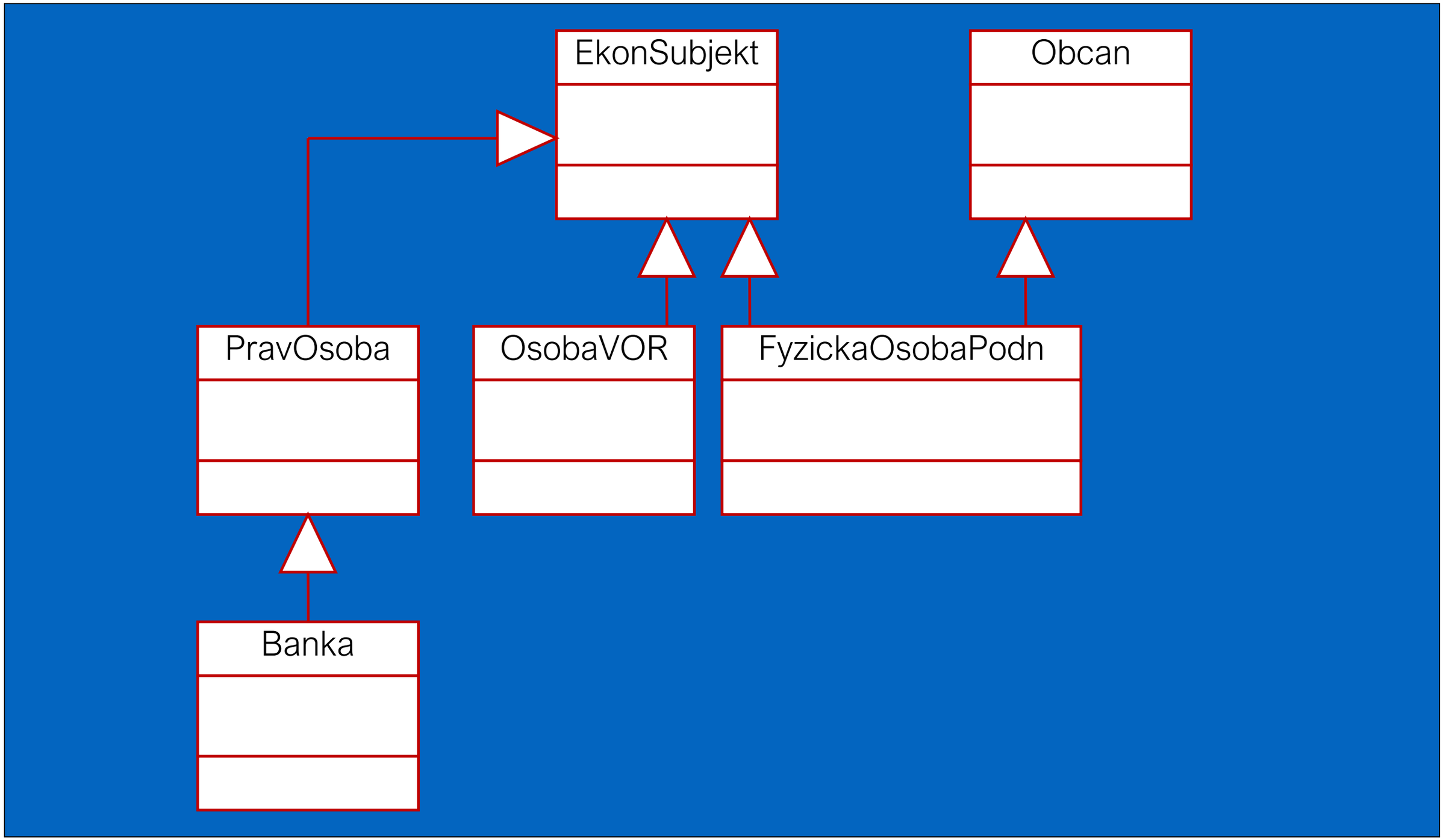
Definice dědičnosti v CDL

- Dědičnost budeme zapisovat do definice typů struktur za klíčové slovo **Inherits** (podobně jako vlastnosti zapisujeme za klíčové slovo **Properties**). Za toto klíčové slovo uvedeme postupně na řádcích jména všech ***přímých předků***.
- Typ ve skutečnosti obsahuje vlastnosti všech předků (nejen přímých). Tranzitivní uzávěr počítá překladač.
- Je vhodné se občas přesvědčit, co všechno jsme ***nadědili***.

Přidávání vlastností

- Nejčastějším případem diference při dědění je přidávání nových vlastností. Ty budeme definovat tak, že je uvedeme do seznamu vlastností nového typu. Ten pak bude obsahovat nejen všechny vlastnosti ***všech předků*** (až na další diference), ale i všechny ***nově definované vlastnosti***.

Příklad z ekonomické oblasti



Příklad

- V našem příkladu je uvedená část ekonomického modelu osob a adres. Vyskytují se zde běžně známé pojmy z ekonomického života jako
- ***ekonomický subjekt*** (EkonSubjekt),
- ***občan*** (Obcan),
- ***právnická osoba*** (PravOsoba),
- ***banka*** (Banka),
- ***osoba v obchodním rejstříku*** (OsobaVOR),
- ***fyzická osoba podnikatel*** (FyzickaOsobaPodn).

Příklad

- Kromě toho, že některé z uvedených objektů jsou současně partnerem (Partner), což přesahuje náš výřez modelu a popisu vlastností (z mnemotechniky identifikátorů), platí:
  - každá právnická osoba je ekonomickým subjektem,
  - každá fyzická osoba podnikatel je ekonomickým subjektem,
  - každá osoba v obchodním rejstříku je ekonomickým subjektem,
  - každá fyzická osoba podnikatel je současně občanem a
  - každá banka je právnickou osobou.

Příklad

concept Obcan/Obcane
inherits
FyzOsoba
Partner
properties
Oznaceni: **string**
Adresat: **string**
RodneCislo: **string** [Key]
RodnePrijmeni: **string**
Prezdivka: **string**
Pozice: **string**

end concept

Příklad

```
concept EkonSubjekt/EkonSubjekty
    inherits
     Partner
    properties
     Adresat:        string
     ObchJmeno:      string
     ICO:            long      [Key]
     DIC:            string  [Key]
     PlatceDPH:      boolean
     PravniForma:    PravFormaES

end concept
```

Příklad

concept FyzOsobaPodn/FyzOsobyPodn
inherits
 Obcan
 EkonSubjekt
properties
 Oznaceni:**string**
 Adresat: **string**
 ObchJmeno: **string**
 DIC: **string** [Key]
end concept

Příklad

concept PravOsoba/PravOsoby

inherits

EkonSubjekt

properties

Oznaceni:**string**

ObchJmeno: **string**

DIC: **string** [Key]

end concept

Příklad

concept OsobaVOR/OsobyVOR
inherits
EkonSubjekt
properties
VypisZOR: VypisZOR
end concept

Příklad

concept Banka/Banky
inherits
PravOsoba
properties
SmerovyKod: **integer** [Key]
SwiftCode: **string**

end concept

Příspěvek

- Připomeňme ještě jednou, že každý podtyp obsahuje všechny vlastnosti svých předků a navíc svoje vlastní nové vlastnosti nazvané ***příspěvek***.
- Obecně je možno vlastnosti i **měnit** (např. typ) a **vypouštět** (exclude), v praxi se to však nevyužívá.

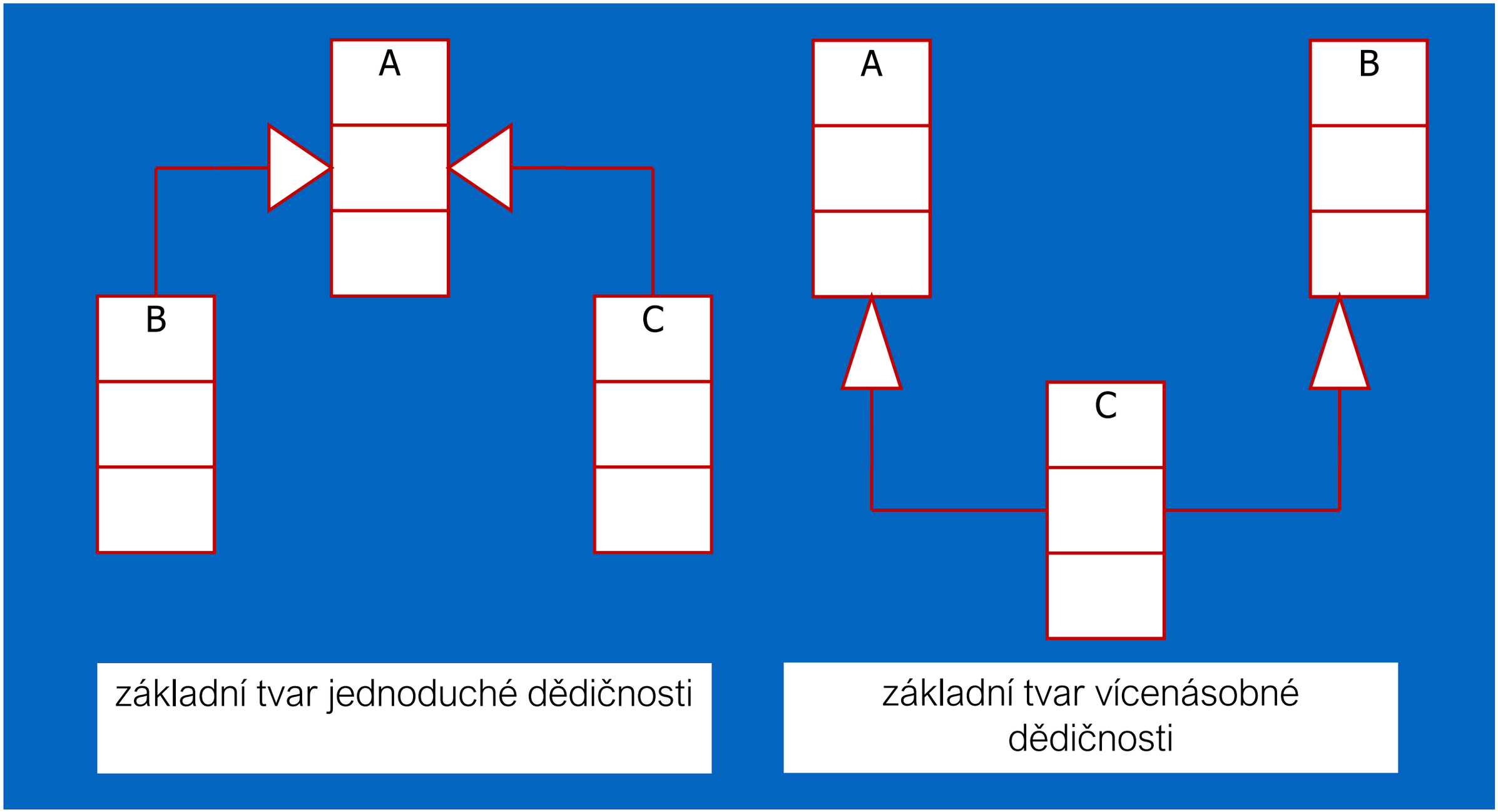
Jednoduchá a vícenásobná dědičnost

- U jednoduché dědičnosti každý následník smí mít ***pouze jediného*** předka. V grafické podobě dědičnosti to znamená, že ze žádného typu nesmí vycházet více, nežli jedna šipka. Takto zakreslený graf je potom ***stromem***.
- U vícenásobné dědičnosti není počet předků omezen. V grafické podobě dědičnosti to znamená, že z každého typu smí vycházet libovolný počet šipek. Takto zakreslený graf je ***obecný acyklický graf***.

Jednoduchá a vícenásobná dědičnost

- V žádném případě se v grafu dědičnosti nesmí vyskytovat cyklus, tj. žádný typ nesmí být svým vlastním předchůdcem nebo následníkem. Na obrázku vidíme dva základní grafické nákresy jednoduché a vícenásobné dědičnosti.

Jednoduchá a vícenásobná dědičnost



základní tvar jednoduché dědičnosti

základní tvar vícenásobné dědičnosti

Typová kompatibilita struktur

- Zavedením hierarchie dědičnosti nad typy jsme stanovili pro každý typ jeho předka (kromě nejvyššího typu v hierarchii). Žádný z typů nemůže být svým vlastním předkem, ani následníkem.

Typová kompatibilita struktur

- Předkové v hierarchii modelují vždy obecnější (generálnější) pojmy a následníci pojmy speciálnější.
- Proto, je-li následníkem osoby např. student, je logické, že každý student je osobou. Nikoliv ovšem naopak. Každá osoba není studentem.
- Podobně, je-li banka následníkem ekonomického subjektu, je každá banka ekonomickým subjektem, ale každý ekonomický subjekt není bankou.

Typová kompatibilita struktur

- Význačnou vlastností je, abychom se na specializovanějšího následníka mohli vždy dívat očima obecnějšího předka. Přeloženo do terminologie typů to znamená, že požadujeme, aby ***každá struktura jistého typu byla zároveň typu všech svých předků***. Struktura není tedy jediného typu, ale je současně více typů, a to svého typu a jeho všech přímých i nepřímých předchůdců.

Typová kompatibilita struktur

- Tedy, pokud máme k dispozici strukturu typu B, která je instancí následníka typu A, pak se může ***B vyskytovat všude tam, kde může být A***. Tj. v deklaracích proměnných, hodnotách vlastností, kolekcích, extentech apod.

Typová kompatibilita struktur

- Říkáme, že typ ***B je kompatibilní s typem A***, nikoliv naopak.
- Je třeba si být vždy vědom, že naopak toto tvrzení neplatí. Každá osoba není studentem, každé zvíře není psem a každý ekonomický subjekt není bankou.

Typová kompatibilita struktur

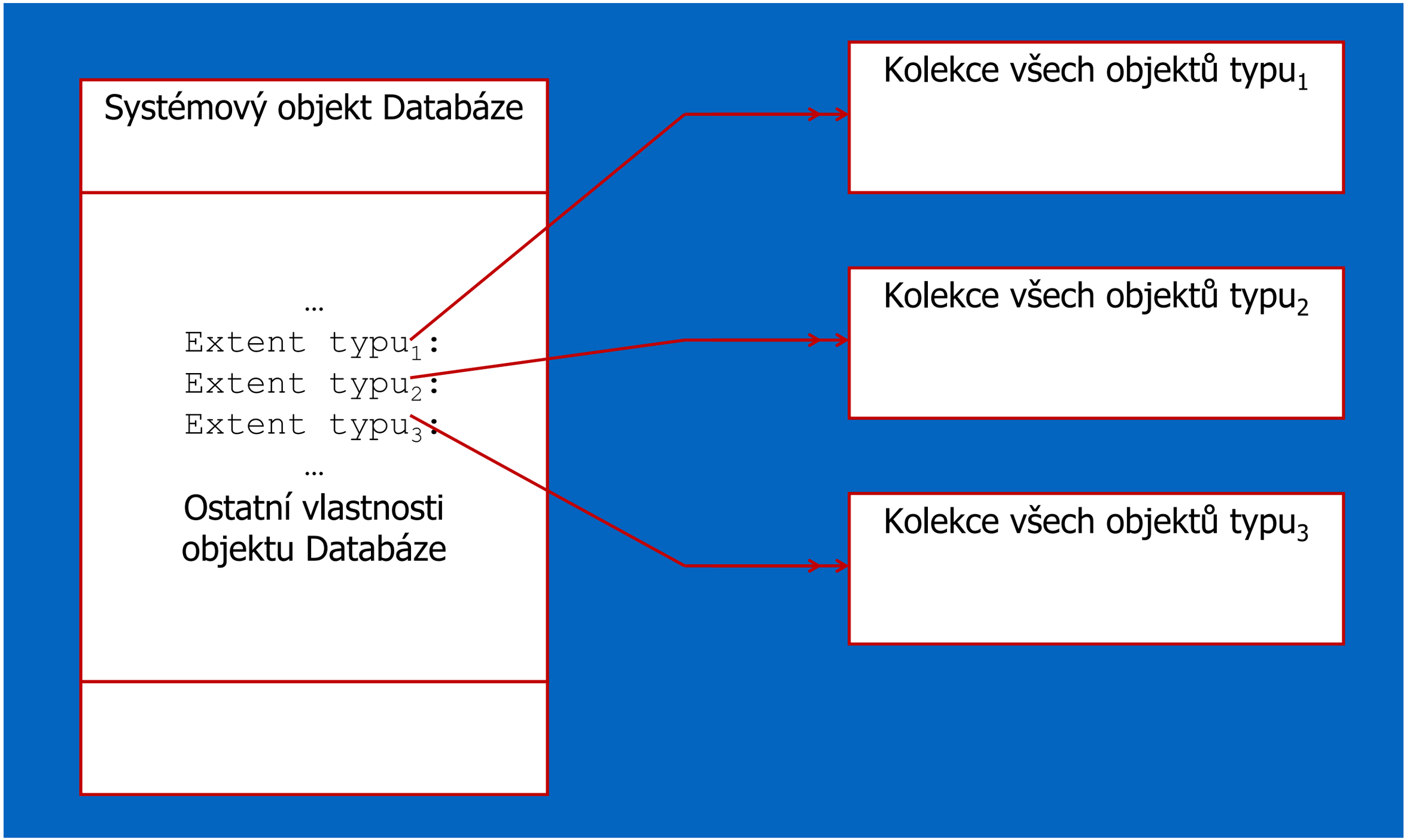
- Proto deklarujeme-li vlastnosti typu A, resp. kolekce prvků typu A, je třeba vidět, že se v ní budou vyskytovat nejen struktury typu A, ale rovněž všech možných následníků jeho typu. Podobně v kolekci budou nejen prvky typu A, ale i prvky všech následníků typu A.
- Například v kolekci ekonomických subjektů se mohou vyskytovat i právnické osoby, banky, podnikající fyzické osoby i osoby v obchodním rejstříku.

Příklad

- Struktura fyzické osoby má povinnou vlastnost UplneJmeno. Tato vlastnost může být uložena efektivněji.

concept FyzOsoba/FyzOsoby
properties
UplneJmeno: **string**
Jmeno: **string**
Prijmeni: **string**
Pohlavi: Pohlavi
DatumNaroz: **date**
Charakt: **string**
Oznaceni: **string**
end concept

Obor hodnot, extent



Obor hodnot, extent

- Podobně jako základní typy, jejichž možné hodnoty jsou předem známy, je potom možné znát ***obor všech možných hodnot*** i pro libovolný objekt v databázi. Oborem hodnot pro objekty jistého typu je hodnota kolekce nazývané ***extent***. Představme si, že systém udržuje pro celou databázi jeden ***systémovou strukturu – databázi***, ve které si ukládá hodnoty význačné pro tuto databázi.

Příklad

- Pokud předpokládáme, že pro fyzickou osobu bude systém udržovat extent, napíšeme:

concept FyzOsoba/FyzOsoby **[Extent]**
properties
UplneJmeno:**string**
Jmeno: **string**
Prijmeni: **string**
Pohlavi: Pohlavi
DatumNaroz: **date**
Charakt: **string**
Oznaceni:**string**
end concept

Příklad

- Atribut Extent je typu Boolean a má implicitní hodnotu False. Je atributem konceptu typu objekt.

Extenty a navigace

- Na rozdíl od relačních databází, kde je nejčastějším prostředkem pro přístup k databázím ***dotaz***, je nejčastějším ***vstupním bodem*** do objektové databáze extent.
- Od něho potom pokračuje navigace po vztazích v databázi. Vytváření extentů je u objektových databází základním prostředkem pro vytváření uživatelské nabídky objektového prohlížeče.

Abstraktní a konkrétní typy

- Při vytváření hierarchie dědičnosti se posléze definované typy struktur rozčlení na dvě kategorie:
  - ty, které slouží jen jako ***stavební kameny*** (vzory) pro vyváření následníků; této kategorie použijeme rovněž, chceme-li ***jednotným způsobem zacházet*** s množinou následníků, které mají množinu společných vlastností (viz kap. 4.4)a
  - ty, které skutečně ***budou mít své instance*** a budou skutečnými strukturami.

Abstraktní a konkrétní typy

- Těm prvním v seznamu říkáme ***abstraktní*** typy. V systému nemůže existovat žádná struktura, která by měla všechny své typy abstraktní. Systém zabrání tomu, aby takovou strukturu bylo možné vytvořit.
- Těm druhým v seznamu říkáme ***konkrétní*** typy.

Příklad

- Výše zmíněný typ modelující ***ekonomický subjekt*** je výhodné deklarovat jako abstraktní. Jeho následníci modelující banku, fyzickou osobu podnikatele apod., jsou již konkrétní.

- Ekonomický subjekt jako takový je pouze stavebním kamenem a nemůže nikdy existovat samostatně. Na všechny konkrétní následníky ekonomického subjektu však můžeme pohlížet jednotným způsobem jako na ekonomický subjekt.

Příklad

- Nicméně ***abstraktní typ může mít extent***. Víme, že v extentu jsou i výskyty všech následníků daného datového typu (zde ekonomického subjektu), které již mohou být konkrétní.
- Abstraktnost typu struktury vyjádříme booleovským atributem Abstract konceptu.

Příklad

concept EkonSubjekt/EkonSubjekty **[Abstract][Extent]**

inherits

Partner

properties

Adresat: **string**

ObchJmeno: **string**

ICO: **long**

DIC: **string** [Key]

PlatceDPH: **boolean**

PravniForma: PravFormaES

end concept

Extenty u děděných objektů

- Zdůrazněme, že ***v extentu typu A se budou vyskytovat všechny objekty typu A, ale i objekty všech následníků typu A***.
- Naopak, při vzniku není objekt zařazen pouze do extentu typu, který byl pro něj zvolen jako typ vzniku, ale rovněž do všech extentů jeho všech předchůdců.

Extenty u děděných objektů

- Musí platit, že pro každý objekt musí být deklarován alespoň jeden extent. Na druhé straně deklarace extentů pro všechny úrovně dědění může značně zneefektivnit operace vytváření a rušení objektu.

Objektový DB model v Javě

Objektově-relační mapování, Java Persistence API

Java EE – Objektově-relační mapování

- Java Persistence API (JPA)
  - Více implementací (EclipseLink, Hibernate, DataNucleus, …)
  - Existují alternativy (JDO)
- Primárně počítá s mapováním do relačních tabulek
- Využívá JDBC
- Mnoho ovladačů pro různé databáze

Java Bean

```
public class Person
{
    private long id;
    private String name;
    private String surname;
    private Date born;

    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
  ….
}
```

Persistence

- Pomocí anotací vytvoříme z třídy **entitu** persistence

```
@Entity
@Table(name = "person")
public class Person
{
    @Id
    private long id;
    private String name;
    private String surname;
    private Date born;
```

Generované ID

```
@Entity
@Table(name = "person")
public class Person
{
    @Id
    @GeneratedValue(strategy = IDENTITY)
    private long id;
    private String name;
    private String surname;
    private Date born;
...
```

Vztahy mezi entitami

- Anotace @OneToMany a @ManyToOne
- Nastavení mapování
  - V Eclipse pohled JPA
- Kolekce v Javě:
  - Collection<?>
    - List<?> (Vector, ArrayList, …)
    - Set<?> (HashSet, …)
    - Map<?, ?> (HashMap, …)s

Konfigurace presistence

- Soubor `persistence.xml`
  - Jméno jednotky persistence
  - Odkaz na data source na serveru
  - Případně další parametry pro mapování
    - Např. řízení automatického generování schématu

Implementace business operací

- Enterprise Java Beans (EJB)
  - Zapouzdřují business logiku aplikace
  - Poskytují business operace – definované rozhraní (metody)
  - EJB kontejner zajišťuje další služby
    - Dependency injection
    - Transakční zpracování
      - Metoda obvykle tvoří transakci, není-li nastaveno jinak

Vytvoření EJB

- Instance vytváří a spravuje EJB kontejner
- Vytvoření pomocí anotace třídy
  - `@Stateless` – bezstavový bean
    - Efektivnější správa – pool objektů přidělovaných klientům
  - `@Stateful` – udržuje se stav
    - Jedna instance na klienta
  - `@Singleton`
    - Jedna instance na celou aplikaci

Použití EJB

- Lokální
  - Anotace `@EJB` – kontejner dodá instanci EJB
- Vzdálené volání – dané rozhraní
  - Rozhraní definované pomocí @Remote

Propojení s JPA

- EJB implementují business operace
  - Často stačí stateless bean
- JPA rozhraní je reprezentováno objektem EntityManager
  - Dodá kontejner pomocí DI

Uložení objektu

```
@PersistenceContext
EntityManager em;


Person person = new Person();
person.setName(„karel“);
em.persist(person);
```

Změna objektu

```
@PersistenceContext
   EntityManager em;


   person.setName(“Karel”);
   em.merge(person);
```

Smazání objektu

```
@PersistenceContext
   EntityManager em;
```

em.remove(person);

Dotazování

- Query q = em.createQuery(“…”);
  q.setParameter(name, value);|
  q.setFirstResult(100);
  q.setMaxResuls(50);
  q.getResultList();

JPQL dotazy

- SELECT p FROM Person p
  WHERE p.name = “John”
- SELECT c FROM Car c
  WHERE c.reg LIKE :pref
- SELECT
  NEW myObject(c.type, count(c))
  FROM Car c
  GROUP BY c.type

ORM pomocí JPA – doplnění

- Asociace A -> B
  - Třída A obsahuje vlastnost typu B nebo kolekci B (podle kardinality)
    - + potenciálně inverzní (obousměrný) vztah
      - Anotace @OneToOne, @OneToMany, @ManyToOne, @ManyToMany
      - Reprezentace v relační databázi
        - @JoinColumn, @JoinTable
- Slabé entitní množiny
- Dědičnost

Vložené entity

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
         private String area;
         private String city;
         private String zipcode;

         // getters and setters

}
```

- Položky adresy chceme reprezentovat strukturou

Vložené entity (II)

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    private Address address;

    // getters and setters
}
```

```
public class Address {

    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

- Jak reprezentovat vztah?
- (@OneToOne je v tomto případě neefektivní)

Vložené entity (III)

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @Embedded
    private Address address;

    // getters and setters

}
```

```
@Embeddable
public class Address {

    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

- Sloupce area, city, zipcode budou přímo v tabulce Person

Slabá entitní množina

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @ElementCollection
    @CollectionTable(
        name=„ADDRESSES“,
        joinColumns=
            @JoinColumn(name=„OWNER“))
    private List<Address> addresses;

    // getters and setters

}
```

```
@Embeddable
public class Address {

    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

- Adresy ve zvláštní tabulce ADDRESSES
- Lze i reprezentovat metadata u vztahu: https://blog.zvestov.cz/software%20development/2015/04/15/jpa-vazebni-tabulky-s-metadaty

Základní datový typ

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @ElementCollection
    @CollectionTable(
        name=„ADDRESSES“,
        joinColumns=
            @JoinColumn(name=„OWNER“))
    @Column(name=„PHONE_NUMBER“)
    private List<String> phones;

    // getters and setters

}
```

Pořadí u seznamů

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @ElementCollection
    @CollectionTable(
        name=„ADDRESSES“,
        joinColumns=
            @JoinColumn(name=„OWNER“))
    @OrderColumn(name=„ORD“)
    private List<Address> addresses;

    // getters and setters

}
```

```
@Embeddable
public class Address {

    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

- Nový sloupec ORD v tabulce adres

Pořadí u seznamů

```
@Entity
public class Person {

    @Id
    private String idperson;
    private String name;
    @ElementCollection
    @CollectionTable(
        name=„ADDRESSES“,
        joinColumns=
            @JoinColumn(name=„OWNER“))
    @OrderBy(name=„priority ASC“)
    private List<Address> addresses;

    // getters and setters

}
```

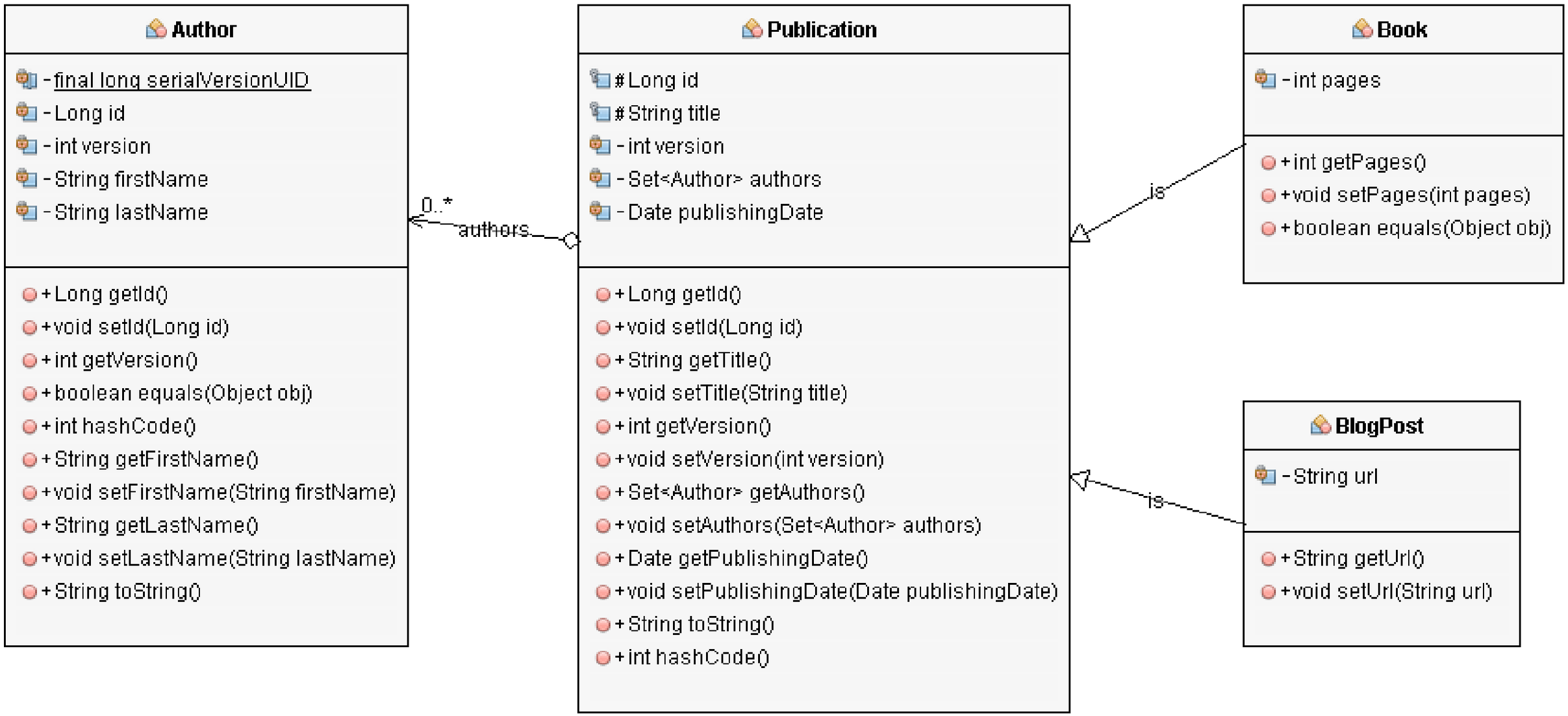
```
@Embeddable
public class Address {

    private int priority;
    private String area;
    private String city;
    private String zipcode;

    // getters and setters
}
```

- Totéž i pro @OneToMany

Dědičnost

- Mapování do relačního schématu
  - Vlastnosti nadtřídy jsou dostupné v odvozených třídách
  - Jedna tabulka nebo více tabulek?
- Typová kompatibilita
  - Odvozená třída je typově kompatibilní s nadtřídou
  - Tvorba extentu jednotlivých tříd?
- Viz např. https://thoughts-on-java.org/complete-guide-inheritance-strategies-jpa-hibernate/

Dědičnost – příklad



Mapped Superclass

```
@MappedSuperclass
public abstract class Publication {

    @Id
    protected Long id;

    protected String title;

    …
}
```

```
@Entity
public class Book extends Publication
{

    private int pages;

    …
}
```

```
@Entity
public class BlogPost extends
Publication {

    private String url;

    …
}
```

Mapped Superclass – výsledek

- Třída *Publication* není entitou
  - Nemá tabulku v databázi
  - **Nelze specifikovat vztah publikace – autor**
- Vhodné pro efektivní definici sdílených vlastností

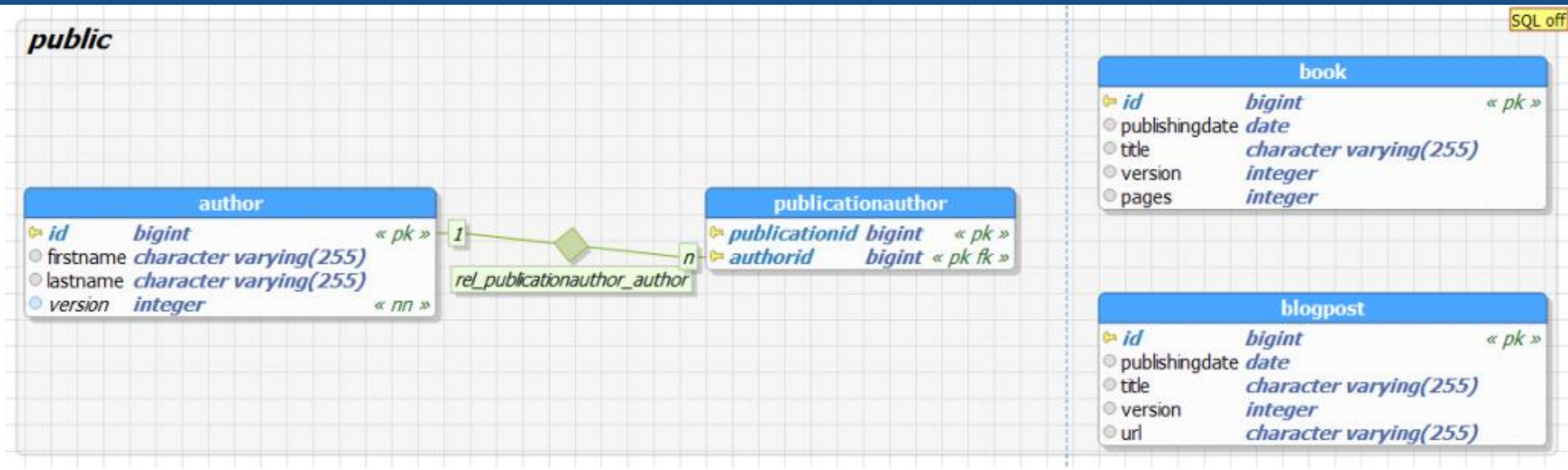
Tabulka pro každou třídu

```
@Entity
@Inheritance(strategy =
    InheritanceType.TABLE_PER_CLASS)
public abstract class Publication {

    @Id
    protected Long id;

    protected String title;

    @ManyToMany
    @JoinTable(…)
    private Set<Author> authors;

    …
}
```

```
@Entity
public class Book extends Publication
{

    private int pages;

    …
}
```

```
@Entity
public class BlogPost extends
Publication {

    private String url;

    …
}
```

Tabulka pro každou třídu – výsledek



- Oddělené tabulky pro jednotlivé třídy
- Třída Publication je entita
  - Lze definovat vztah Publication – Author
- Dotazy nad třídou Publication nejsou efektivní
  - Vede na JOIN nad konkrétními tabulkami
  - Např. `for (Publication p : author.getPublications()) { … }`

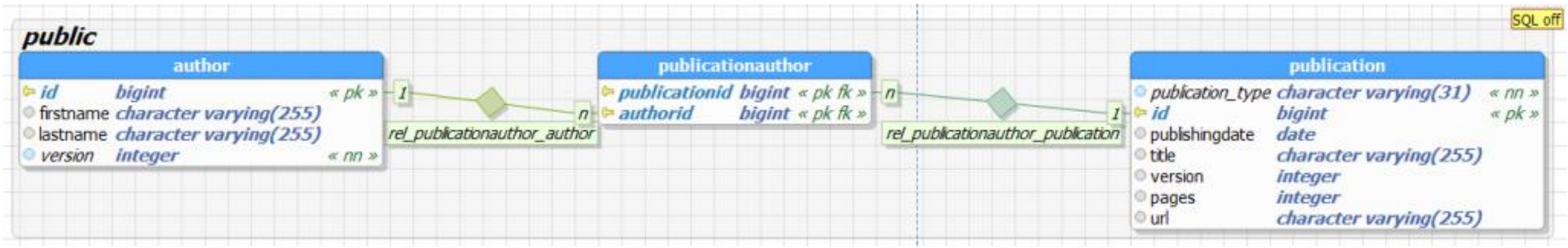
Jediná tabulka

```
@Entity
@Inheritance(strategy =
    InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name =
    "Publication_Type")
public abstract class Publication {

    @Id
    protected Long id;

    protected String title;

    @ManyToMany
    @JoinTable(…)
    private Set<Author> authors;

    …
}
```

```
@Entity
@DiscriminatorValue("Book")
public class Book extends Publication
{

    private int pages;

    …
}
```

```
@Entity
@DiscriminatorValue("Blog")
public class BlogPost extends
Publication {

    private String url;

    …
}
```

Jediná tabulka – výsledek



- Jedna tabulka pro všechny odvozené třídy
  - Jeden sloupec slouží jako diskriminátor
- Efektivní dotazování
  - Filtrování podle hodnoty diskriminátoru
  - Snadná reprezentace vztahu Publication – Author
- Hodnoty nevyužitých vlastností jsou nulové
  - Nelze specifikovat *not null* nad vlastnostmi podtříd – omezuje kontrolu integrity dat

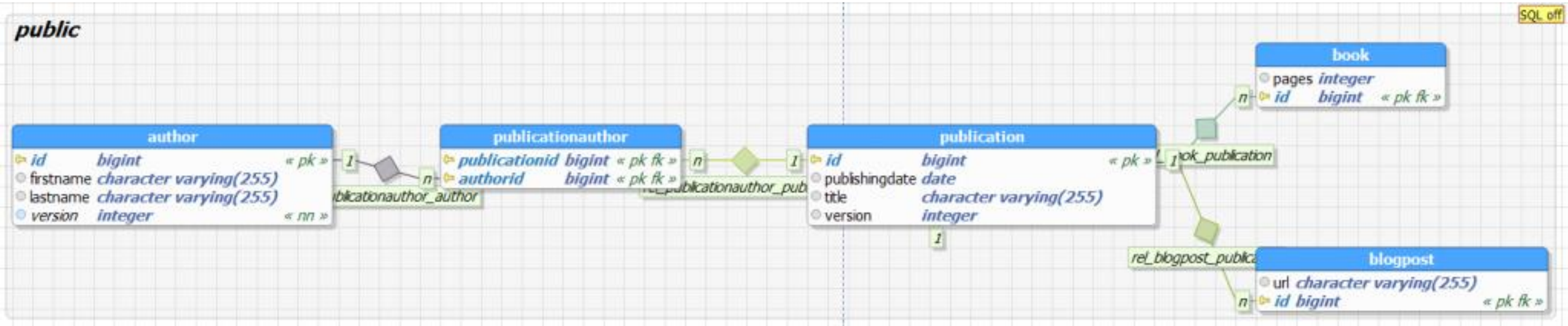
Joined

```
@Entity
@Inheritance(strategy =
    InheritanceType.JOINED)
public abstract class Publication {

    @Id
    protected Long id;

    protected String title;

    @ManyToMany
    @JoinTable(…)
    private Set<Author> authors;

    …
}
```

```
@Entity
public class Book extends Publication
{

    private int pages;

    …
}
```

```
@Entity
public class BlogPost extends
Publication {

    private String url;

    …
}
```

Joined – výsledek



- Tabulka pro každou třídu včetně Publication
- Snadná reprezentace vztahů, možnost integritních omezení
- Neefektivní dotazování
  - Vždy vede na JOIN více tabulek

Otázky?