Objektově orientované programování

Abstraktní třída, vícenásobná dědičnost

2023/2024

Osnova přednášky

- Polymorfismus - důsledky.
- Abstraktní třída.
- Vícenásobná dědičnost.

Polymorfismus - důsledky

Polymorfismus

- *Polymorfismus* je schopnost objektu vystupovat v různých rolích (formách)…
- *Časná* a *pozdní* vazba. Kdy kterou použít?
- Souvisí to se substitučním principem, tedy se zastupitelností předka potomkem.
- V C++ je polymorfismus spojen s dědičností a pozdní vazbou!!!
  - Nejde tedy jen o obyčejné překrytí.

Virtuální metody

- Jednou označená metoda jako virtuální zůstává virtuální ve všech potomcích.
- Uvnitř *konstruktorů* sice můžeme volat virtuální metody, ale ty se budou chovat *nevirtuálně*.
- Virtuální destruktor je potřeba, pokud použijeme polymorfní přiřazení

Nevirtuální destruktor

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);
    ~Account();

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    virtual bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

```
class CreditAccount : public Account
{
private:
    double credit;

public:
    CreditAccount(int n, Client *o, double c);
    CreditAccount(int n, Client *o, double ir, double c);
    ~CreditAccount();

    virtual bool CanWithdraw(double a);
};
```

```
Account::~Account()
{
    cout << "Account destructor" << endl;
}
```

```
CreditAccount::~CreditAccount()
{
    cout << "CreditAccount destructor" << endl;
}
```

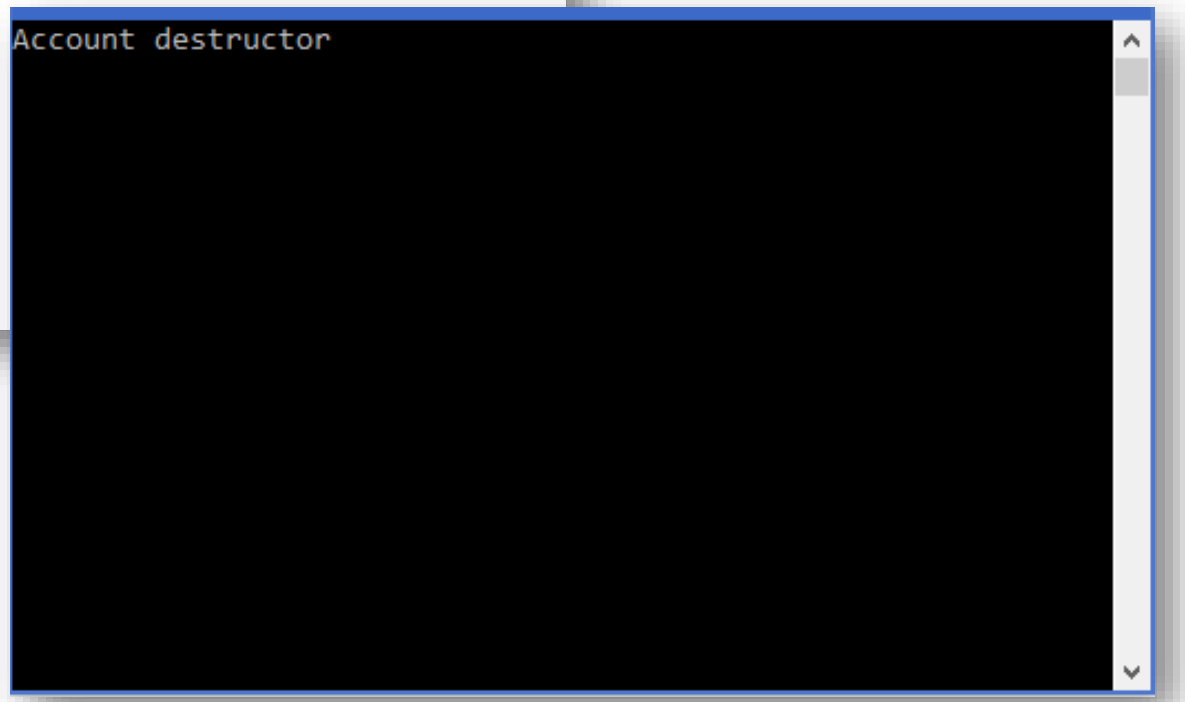
Výsledek

```
int main()
{
    Client *o = new Client(0, "Smith");
    CreditAccount *ca = new CreditAccount(1, o, 1000);

    Account *a = ca;

    delete a;
    delete o;

    getchar();
    return 0;
}
```



Virtuální destruktor

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);
    virtual ~Account();

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    virtual bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

```
class CreditAccount : public Account
{
private:
    double credit;

public:
    CreditAccount(int n, Client *o, double c);
    CreditAccount(int n, Client *o, double ir, double c);
    virtual ~CreditAccount();

    virtual bool CanWithdraw(double a);
};
```

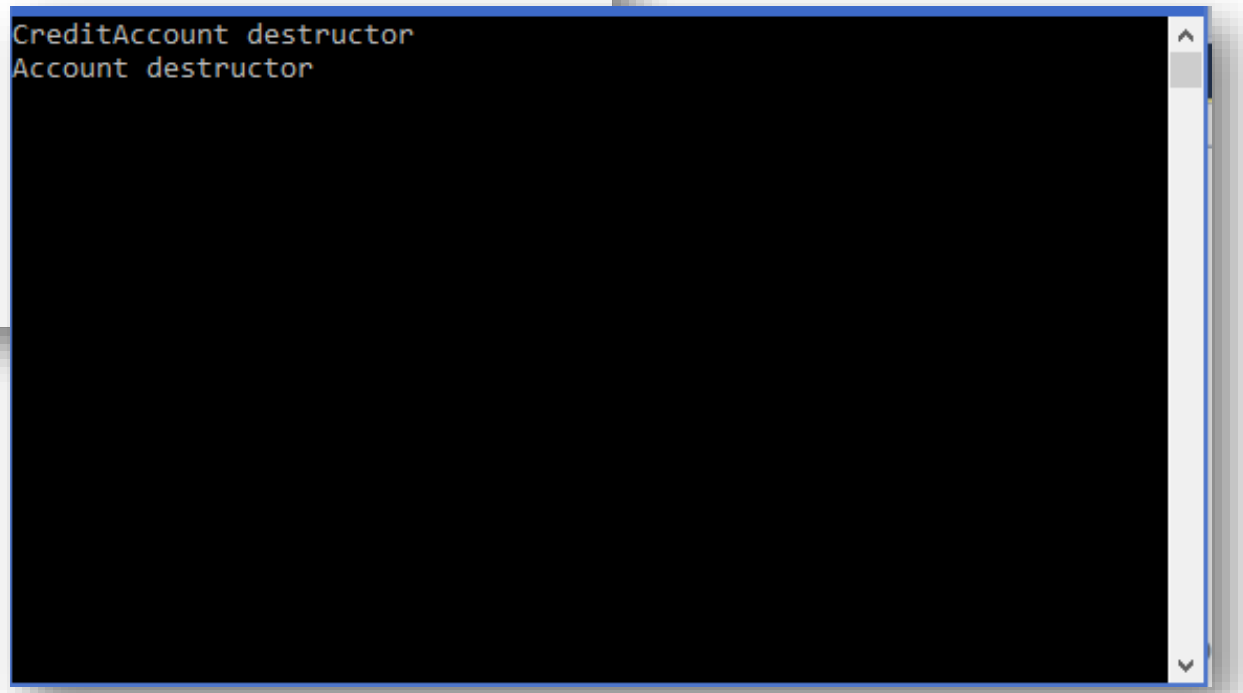
Výsledek

```
int main()
{
    Client *o = new Client(0, "Smith");
    CreditAccount *ca = new CreditAccount(1, o, 1000);

    Account *a = ca;

    delete a;
    delete o;

    getchar();
    return 0;
}
```



Abstraktní třída

Čistě virtuální metoda

- Metoda, která má pouze deklaraci.
- Nemá implementaci (definici).
- K čemu to je dobré?
  - Pro správný návrh…

Čistě virtuální metody

```
class AbstractAccount
{
public:
    AbstractAccount();
    virtual ~AbstractAccount();

    virtual bool CanWithdraw(double a) = 0;
};

AbstractAccount::AbstractAccount()
{
    cout << "AbstractAccount constructor" << endl;
}

AbstractAccount::~AbstractAccount()
{
    cout << "AbstractAccount destructor" << endl;
}
```

Dědičnost z nové třídy

```
class Account : public AbstractAccount
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);
    virtual ~Account();

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    virtual bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

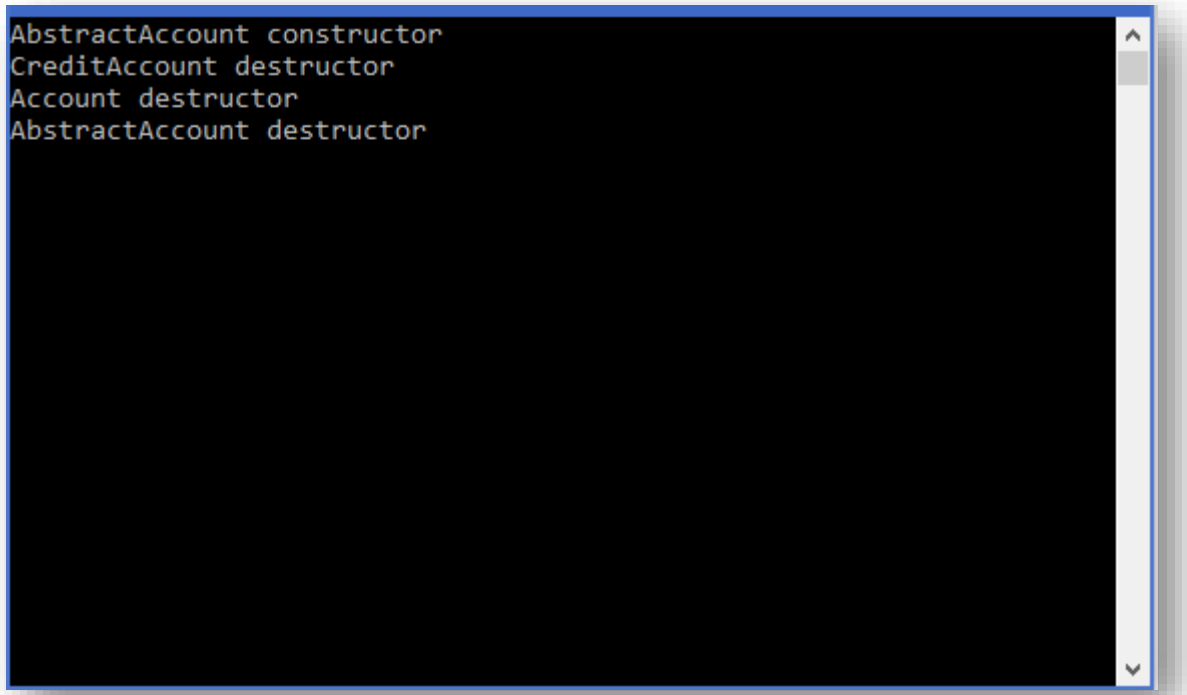
```
int main()
{
    Client *o = new Client(0, "Smith");
    CreditAccount *ca = new CreditAccount(1, o, 1000);

    AbstractAccount *aa = ca;

    delete aa;
    delete o;

    getchar();
    return 0;
}
```

Výsledek


AbstractAccount constructor
CreditAccount destructor
Account destructor
AbstractAccount destructor

Ale…

```
int main()
{
    Client *o = new Client(0, "Smith");
    CreditAccount *ca = new CreditAccount(1, o, 1000);

    AbstractAccount *aa = ca;

    delete aa;
    delete o;

    AbstractAccount *naa = new AbstractAccount();

    getchar();
    return 0;
}
```

- Proč???

Abstraktní třída

- Třída, která má alespoň jednu čistě virtuální metodu se nazývá *abstraktní*.
- Abstraktní proto, že nemůžeme vytvořit její instanci (protože čistě viruální metoda má sice deklaraci, ale *nemá definici/implementaci*).
- Může, ale nemusí, mít členské proměnné a implementované metody.
- Má konstruktor a destruktor. Pro potomka.

Čistě abstraktní třída

- Třída, jejíž všechny metody jsou čistě virtuální.
- K čemu může být taková třída?
  - Jako „prázdný“ vzor pro dědičnost.
- Deklaruje, ale nedefinuje, budoucí společné chování potomků.
- Je jako „interface“ v Java nebo C#.

Dědičnost účtů

AbstractAccount – abstraktní třída

Account

PartnerAccount ***CreditAccount***

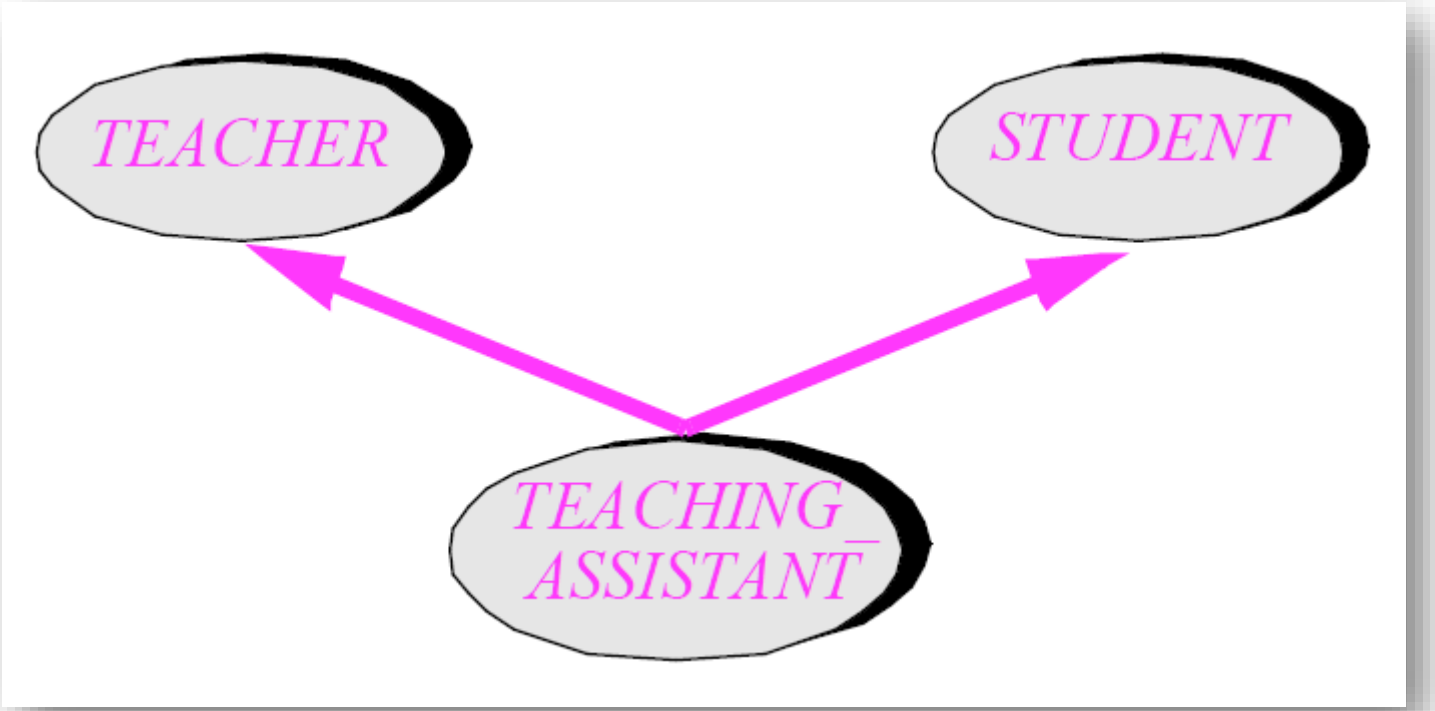
- Kdo tedy má implementovat čistě virtuální metodu? **Potomek**!!!

Vícenásobná dědičnost

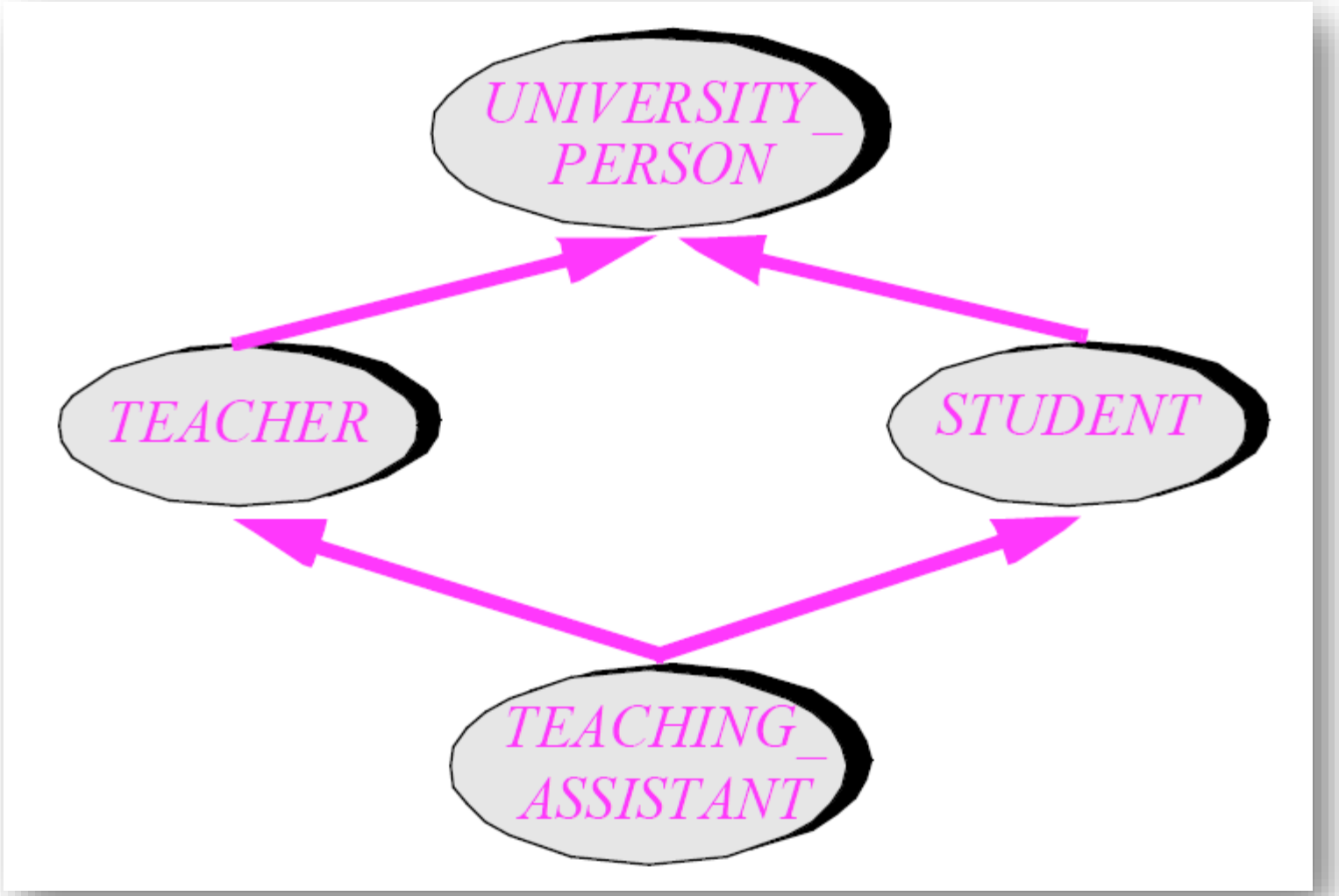
Vícenásobná dědičnost

- Může potomek dědit z více tříd?
- Proč ne?
- Proč ano?
- Je to docela pěkný a mnohdy užitečný koncept - ale je také poněkud nebezpečný a s často obtížně pochopitelným chováním objektů…

Je to správně nebo ne?



Ale proč ne?



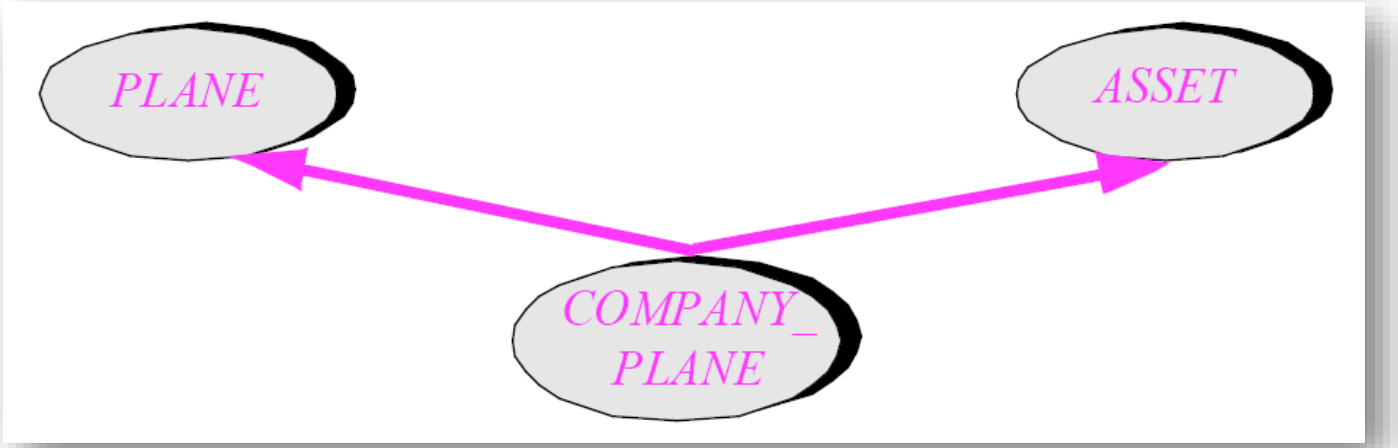
Není to pro začátečníky…

- …a pokročilí se bez toho obejdou.
- Problém je v tom, že *Teacher* a *Student* nejsou oddělené abstrakce.
- Sdílejí společné rysy *University_Person*.
- Je tam i problém technický?

Má to tedy smysl?

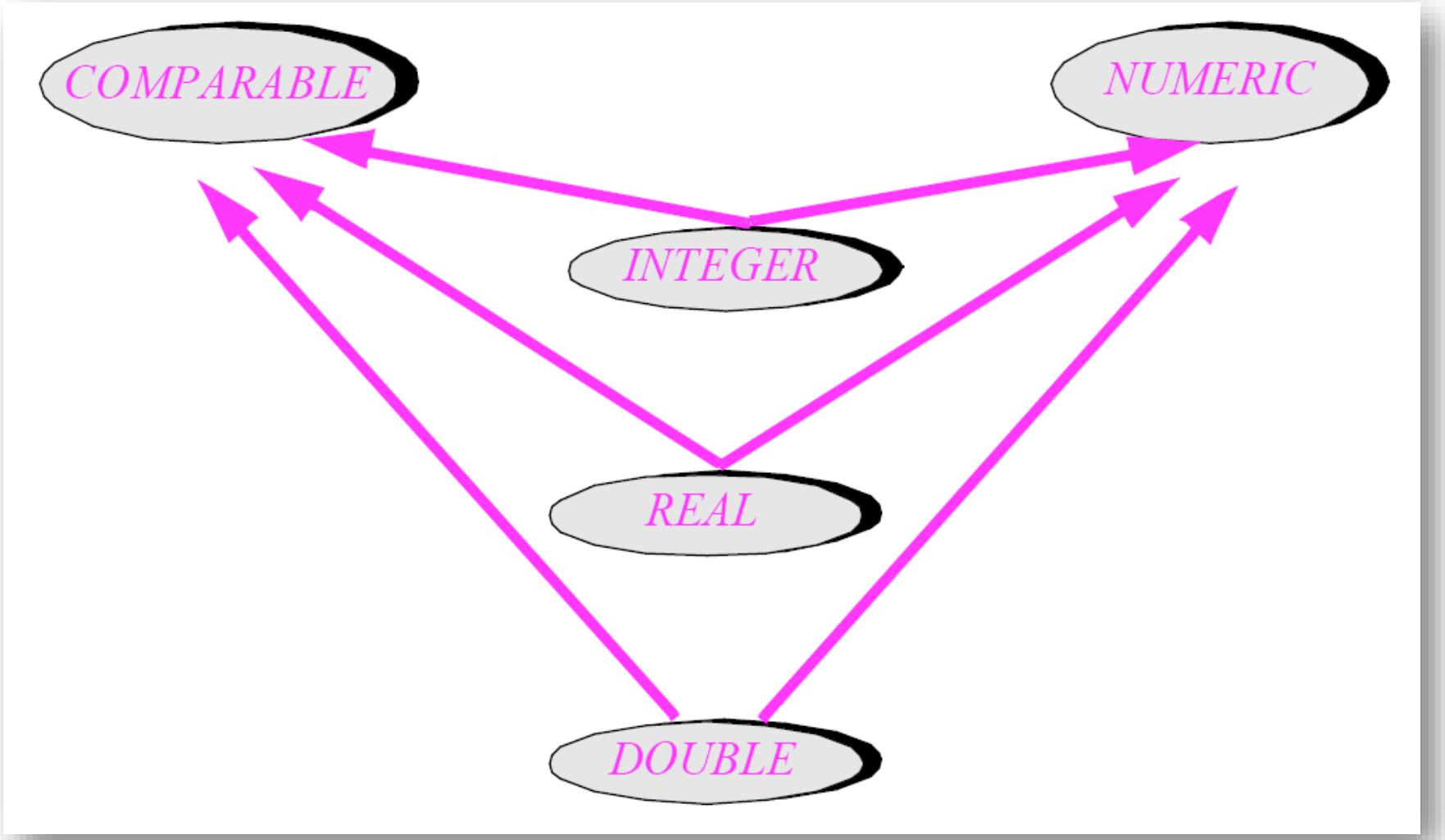
- Předci musí to být různé abstrakce.
- Různé abstrakce můžeme chápat tak, že **NEMAJÍ** společný stav ani chování.
- Pak má smysl o vícenásobné dědičnosti uvažovat.

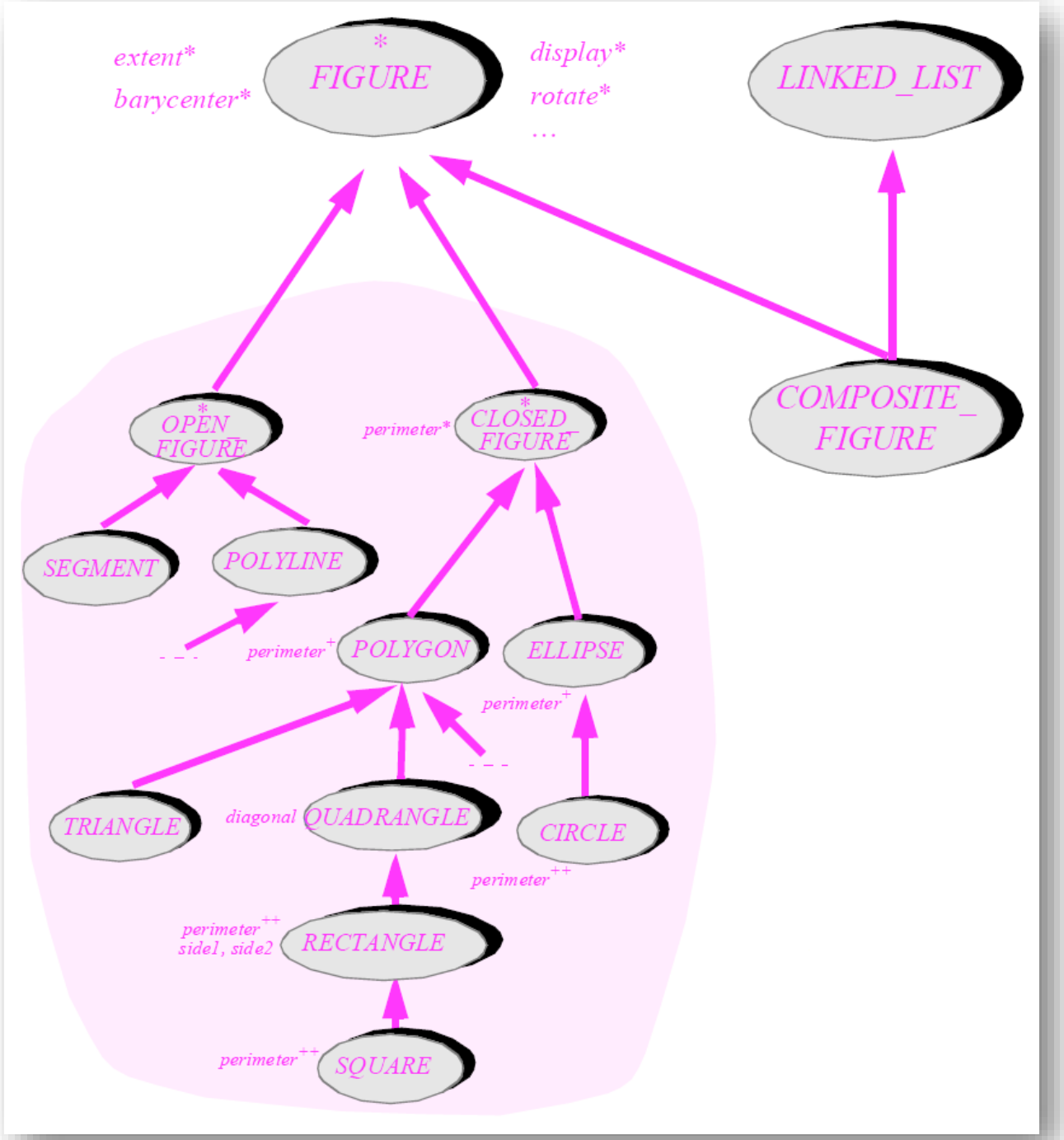
Správný příklad



- Letadlo společnosti je:
  - Letadlem s technickými parametry a funkcemi s nimi spojenými.
  - Majetkem s evidenčními údaji a odpovídající funkčností.

A další…




extent*
barycenter*
*
FIGURE
display*
rotate*
...
LINKED_LIST
*
OPEN_
FIGURE
perimeter*
*
CLOSED_
FIGURE
COMPOSITE_
FIGURE
SEGMENT
POLYLINE
perimeter+
POLYGON
ELLIPSE
perimeter+
TRIANGLE
diagonal
QUADRANGLE
CIRCLE
perimeter++
perimeter++
side1, side2
RECTANGLE
perimeter++
SQUARE

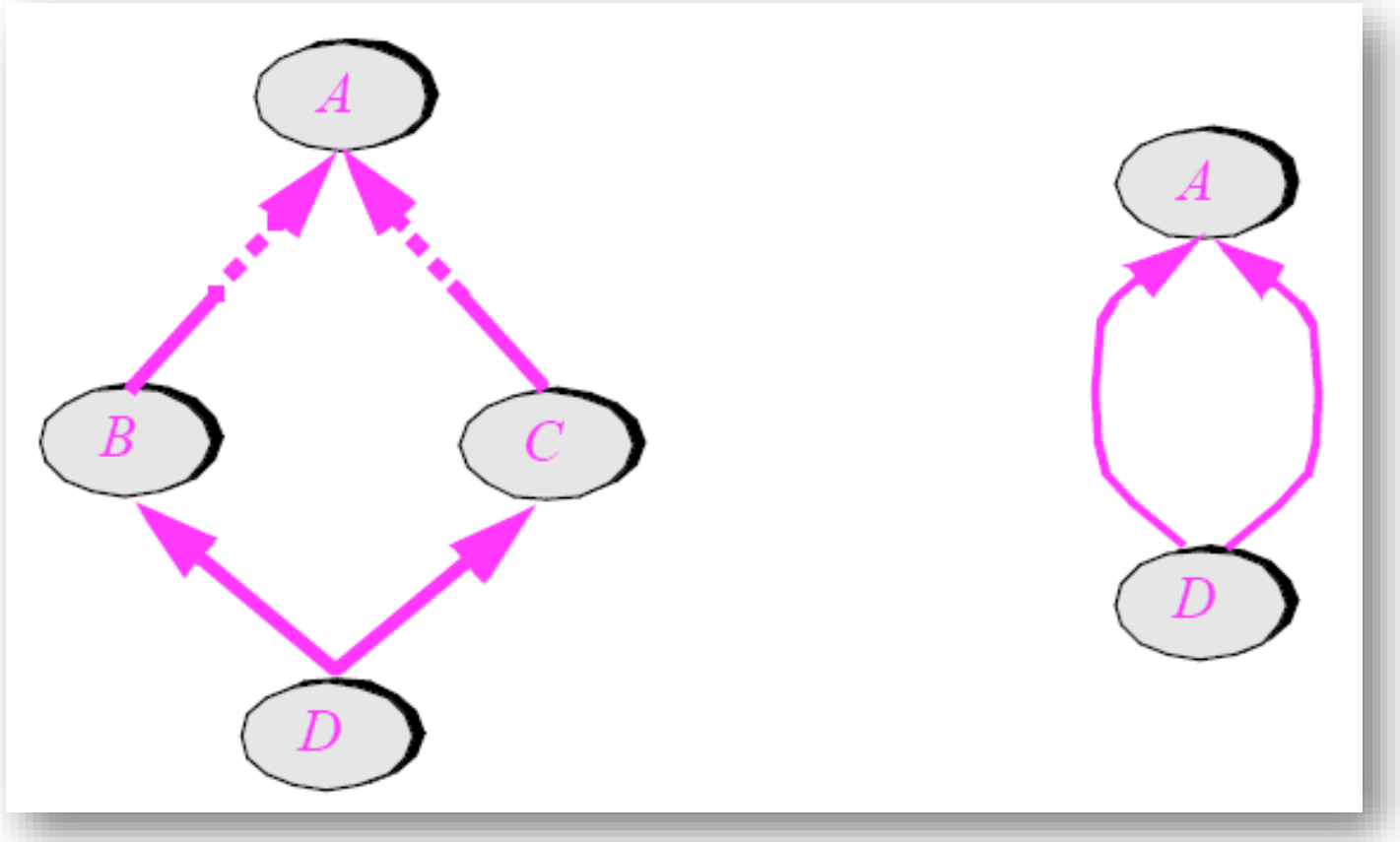
Obejdeme se bez ní?

- Ano, ale…
- Někdy potřebujeme, aby třída měla vlastnosti nad rámec základní abstrakce, kterou jinak popisuje.
- Je to opět o zastupitelnosti předka potomkem.
- Tentokrát ale může potomek vystupovat v rolích, které se chováním podstatně liší.

Problémy, které mohou nastat

- Konflikty jmen
  - Třídy, ze kterých se dědí, mohou mít stejně pojmenované členské položky (proměnné i metody).
  - Dá se vyřešit různými způsoby.
- Opakovaná dědičnost
  - Dá se uhlídat, že se vícekrát nedědí z jedné a téže třídy?
  - Tady je to trošku horší…

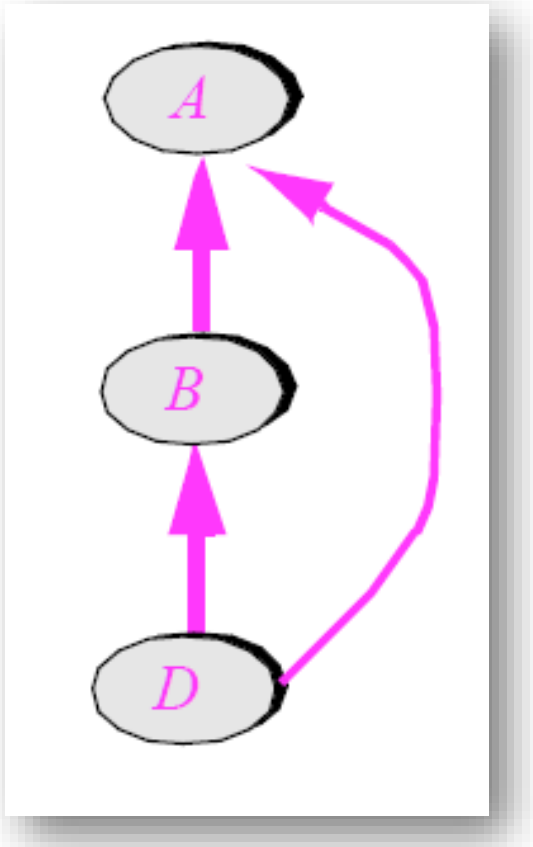
Opakovaná dědičnost



Nepřímá

Přímá

A hlavně nenápadně…



Redundantní dědičnost

Proč se o tom tedy bavíme?

- Je to užitečný koncept, zejména proto, že objekt může zastupovat dvě odlišné abstrakce
- Musí se ale používat s rozmyslem.
- A kdy tedy?

Použití vícenásobné dědičnosti

- Potřebujeme-li, aby objekty reprezentovaly v různých situacích různé abstrakce.
  - Nejlépe natolik různé, že nehrozí konflikty jmen.
  - Souvisí to se zastupitelností předka potomkem.
- Ideální je, pokud jsou předci čistě abstraktní třídy (bez dat).
  - Pak je to totéž, co v moderních objektových jazycích „interface“.
  - Interface je koncept, který se prosadil jako náhrada vícenásobné dědičnosti.

Úkoly na cvičení

- Implementuje příklady z přednášky, zaměřte se na využití abstraktní a čistě abstraktní třídy a na to, jak fungují konstruktory a destruktory.

- Reimplementuje jednoduchou dědičnou hierarchii geometrických objektů tak, aby jste využili abstraktní a čistě abstraktní třídy.

Kontrolní otázky

- Co je čistě virtuální metoda?
- Kdy je vhodné použít čistě virtuální metodu? Uveďte příklad.
- Co je abstraktní třída?
- Kdy je vhodné použít abstraktní třídu? Uveďte příklad.
- Má abstraktní třída konstruktor a destruktor? A proč?
- Může mít abstraktní třída členská data a funkce (metody)?
- Co je čistě abstraktní třída?
- Co je vícenásobná dědičnost?
- Kdy není vhodné použít vícenásobnou dědičnost? Uveďte příklad.
- Kdy je možné použít vícenásobnou dědičnost? Uveďte příklad.
- Jaké problémy mohou nastat při použití vícenásobné dědičnosti? Uveďte příklad
- Co je opakovaná dědičnost? Uveďte příklady.
- Proč můžeme potřebovat vícenásobnou dědičnost? S čím to souvisí?

Ke studiu

- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall 1997. [486-490, 519-529]