Principy OOP

- Objektově orientované programování vychází ze třech základních principů (rysů):
  - zapouzdření (encapsulation)
  - dědičnost (inheritance)
  - polymorfismus

Zapouzdření

- Třída může obsahovat libovolné množství členů
- Při správném objektově orientovaném přístupu by však měla být data skryta (zapouzdřena) uvnitř třídy
- K jednotlivým objektům by se mělo přistupovat prostřednictvím metod nebo vlastností
- Toto snižuje nebezpečí vzniku chyby a umožňuje tvůrci modifikovat vnitřní reprezentaci třídy

Dědičnost (1)

- Jedná se o prostředek, který umožňuje dosáhnout snadné rozšiřitelnosti a znovupoužitelnosti již existujících částí programu
- Dovoluje definovat novou třídu (**`B`**), která vznikne pouze přidáním nových členů k těm, které již byly definovány v rámci jiné třídy (**`A`**)
- Třída:
  - **`A`** se stává bezprostředním předkem třídy **`B`**
  - **`B`** je bezprostředním potomkem třídy **`A`**

Dědičnost (2)

- Nechť je dána třída:

```
class Datum
{
    private byte den, mesic;
    private ushort rok;

    public Datum(byte d, byte m, ushort r)
      {…}
    public string GetDatum()
      {…}
    public bool JePrestupnyRok()
      {…}
}
```

- Na jejím základě lze definovat jejího potomka

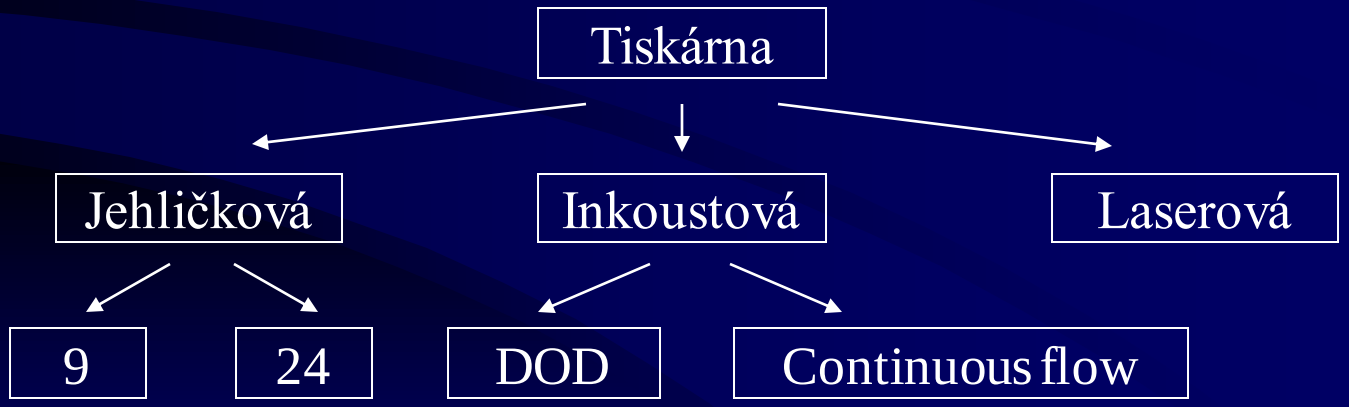
Dědičnost (3)

- Informace o bezprostředním předkovi se zapisuje v definici třídy (za jejím názvem – oddělena dvojtečkou)
- Příklad:

```
class DatumRoz : Datum
{
    public DatumRoz
       (byte d, byte m, ushort r)
       : base(d, m, r) {…}
    public void PrictiDen() {…}
    public void OdectiDen() {…}
    private byte DniVMesici() {…}
}
```

Dědičnost (4)

- Hierarchii tříd obvykle budujeme tak, že k dříve definovaným třídám přidáváme další třídy jako jejich potomky
- Příklad:



Dědičnost (5)

- Jestliže třída **`B`** je potomkem třídy **`A`**, tak jsou třídě **`B`** automaticky dostupné všechny členy (vyjma konstruktorů instancí, statických konstruktorů a destruktorů) třídy **`A`**, aniž by bylo nutné je znovu definovat
- Říkáme, že třída **`B`** dědí členy třídy **`A`**
- Jestliže při definici třídy v jazyku C# neuvedeme žádného předka, je jejím předkem automaticky třída **`object`** (**`System.Object`**)

Dědičnost (6)

- Každá definovaná třída automaticky dědí členy definované ve třídě **`object`** (např. metodu **`ToString`**)
- Zděděné datové položky bývají obvykle inicializovány vyvoláním konstruktoru předka
- Konstruktor předka se volá pomocí klíčového slova **`base`**, které je uvedeno za hlavičkou konstruktoru potomka (odděleno dvojtečkou)
- Příklad:

```
public DatumRoz(byte d, byte m, ushort r)
   : base(d, m, r) {…}
```

Dědičnost (7)

- Pokud volání konstruktoru předka není explicitně provedeno, pak zkouší překladač volat konstruktor implicitní
- Tj. definice:

```
public DatumRoz(byte d, byte m, ushort r)
   {…}
```

bude automaticky přepsána (doplněna) na tvar:

```
public DatumRoz(byte d, byte m, ushort r)
   : base() {…}
```

- Doplněný tvar je funkční pouze v případě, že předek obsahuje (má v programu definovaný) veřejný implicitní konstruktor (v opačném případě dojde k chybě při překladu)

Dědičnost (8)

- Skrývání (maskování) metod:
  - dovoluje na úrovni potomka definovat metodu, která má stejnou signaturu (stejné jméno, stejný počet a typy formálních parametrů) jako metoda na úrovni předka
  - metoda potomka tak skrývá (maskuje) metodu předka
  - metody, které provádějí skrývání by měly být definovány pomocí klíčového slova **`new`** (v opačném případě bude překladač vypisovat varování)

Dědičnost (9)

- Platí, že proměnné typu třída **A** (předchůdce) je možné přiřadit proměnnou typu třída **B** (následníka) – opačné přiřazení není možné
- Lze provést přiřazení tvaru:

```
objekt předka = objekt potomka;
```

- Příklad:

```
Datum d = new Datum(1, 1, 2000);
DatumRoz dRoz = new DatumRoz(2, 8, 2010);
    ⋮
d = dRoz;
```

Polymorfismus (1)

- Rys umožňující, aby akce uskutečňované nad různými objekty byly pojmenovány stejně, přičemž ale jejich realizace je různá (podle toho, nad kterým objektem se provádějí)
- Nástrojem pro realizaci polymorfismu jsou tzv. virtuální metody
- Virtuální metody:
  - obsahují ve své definici klíčové slovo **`virtual`**
  - implementace virtuálních metod může být na úrovni potomka nahrazena implementací jinou

Polymorfismus (2)

- umožňují volat různé verze stejné metody na základě typu objektu určeného dynamicky za běhu programu
- proces nahrazení implementace zděděné virtuální metody se označuje jako předefinování (potlačení, overriding) metody
- metoda realizující odlišnou implementaci na úrovni potomka musí ve své definici obsahovat klíčové slovo **`override`**
- nová implementace metody (na úrovni potomka) může volat původní implementaci téže metody (na úrovni předka) pomocí klíčového slova **`base`**, jež se zapisuje v příkazové části metody

Polymorfismus (3)

– poznámky:
  - klíčová slova **`virtual`** a **`override`** nelze použít u soukromých (privátních) metod
  - signatura metody, která provádí předefinování, musí být stejná (včetně návratového typu) jako signatura původní metody
  - metoda s klíčovým slovem **`override`** je rovněž virtuální a může být na úrovni dalšího potomka opět předefinována

Skrytí vs. předefinování

- Skrytí:
  - jedna metoda nahrazuje metodu jinou
  - tyto metody obvykle nemají žádnou vazbu a mohou provádět zcela odlišné úlohy
- Předefinování:
  - zajišťuje různé implementace stejné metody
  - všechny implementace jsou příbuzné – provádějí v zásadě stejnou úlohu, ale specifickým způsobem pro danou třídu

Rozšiřující metody

- Umožňují rozšířit existující datový typ (třídu nebo strukturu) dodatečnými statickými metodami
- Tyto metody jsou okamžitě k dispozici jakýmkoliv příkazům, které se odkazují na data rozšířeného typu
- Definují se ve statické třídě
- Typ, kterého se rozšiřující metoda týká, se uvádí jako první parametr metody společně s klíčovým slovem **`this`**

Konstanty (1)

- Členy třídy reprezentující konstantní hodnoty, které se v průběhu programu nemohou měnit
- Jedná se o hodnoty, které musí být určitelné v době překladu programu
- Konstanty jsou statickými členy třídy, avšak při jejich definici se nepoužívá klíčové slovo **`static`**
- Lze je definovat pomocí klíčového slova **`const`**

Konstanty (2)

- Příklad:

```
class Fyzika
{
    public const double h = 6.62606957E-34;
    public const double
          hbar = h/(2*Math.PI);
    public const double F = 9.6485E4;
      ⋮
}
```

Vlastnosti (1)

- Členy třídy umožňující přístup k charakteristickým atributům objektu nebo třídy, např.:
  - délka řetězce
  - velikost fontu
  - titulek okna
  - jméno zákazníka
  - apod.
- Představují rozšíření datových položek:
  - pojmenované členy s daným datovým typem
  - používají stejnou syntaxi pro své zpřístupnění

Vlastnosti (2)

- Na rozdíl od datových položek nereprezentují paměťové místo pro uložení hodnoty
- Překladač automaticky převádí zpřístupnění vlastností na volání přístupových metod
- Přístupové metody specifikují příkazy, jež mají být provedeny při:
  - čtení hodnoty vlastnosti
  - změně hodnoty vlastnosti (zápisu)

Vlastnosti (3)

- Přístupové metody jsou zapisovány pomocí klíčových slov
  - **`get`**: uvozuje přístupovou metodu pro čtení
  - **`set`**: uvozuje přístupovou metodu pro zápis
- Zapisovaná data jsou přístupovým metodám (**`set`**) předávána pomocí vestavěného (skrytého) parametru **`value`**
- Konvence:
  - soukromé datové položky jsou psány s malým počátečním písmenem, zatímco veřejné vlastnosti s počátečním písmenem velkým

Vlastnosti (4)

- Příklad:

```
class Ctverec
{
    private int strana;
    public int Strana
    {
        get {
            return strana;
        }
        set {
            strana = value;
        }
    }
    ⋮
}
```

Vlastnosti (5)

- Nechť je dána deklarace:
  ```
  int s;
  Ctverec c = new Ctverec(5);
  ```
- Vlastnost **`Strana`** lze zpřístupnit:
  - pro čtení – např.:
    `s = c.Strana;` (volá **`c.Strana.get`**)
  - pro zápis – např.:
    `c.Strana = 2*s;` (volá **`c.Strana.set`**)
- Vlastnost může obsahovat pouze metodu:
  - **`get`**: vlastnost určena jen pro čtení
  - **`set`**: vlastnost určena jen pro zápis

Vlastnosti (6)

- Omezení vlastnosti:
  - vlastnost nelze použít jako parametr s modifikátorem **`ref`** nebo **`out`** (za vlastností se skrývá přístupová metoda nikoliv paměťové místo)
  - vlastnost může obsahovat jednu přístupovou metodu **`get`** a jednu přístupovou metodu **`set`** (nesmí obsahovat žádné jiné metody, datové položky nebo vlastnosti)
  - přístupové metody nesmí přijímat žádné parametry (zapisovaná data jsou u metody **`set`** předávána prostřednictvím **`value`**)
  - u vlastností nelze používat modifikátor **`const`**

Vlastnosti (7)

- Jestliže metody **`get`**, resp. **`set`** provádějí pouze operace, které slouží k přečtení, resp. přiřazení do datové položky, pak je možné jejich příkazovou část vynechat
- Příklad:

```
class Ctverec
{
    public int Strana { get; set; }
        ⋮
}
```

Vlastnosti (8)

- Překladač automaticky vygeneruje odpovídající programový kód:

```
class Ctverec
{
    private int _strana;
    public int Strana
    {
        get { return _strana; }
        set { _strana = value;}
    }
    ⋮
}
```