Polymorfizmus

Úvod

- polymorfizmus = „schopnost vyskytovat se v různých tvarech“
- polymorfizmus v OOP platí pro metody – metoda se stejným názvem může dělat různé činnosti
  - částečně již používáme – přetěžování metod
- příklad z reálného světa – vypínač čehokoli: stejné rozhraní pro uživatele, uvnitř ale zcela jiná funkčnost
- příklad z .NET – třídy `Console`, `StreamWriter`, `SerialPort` metoda `Write()`

Metody a vazby

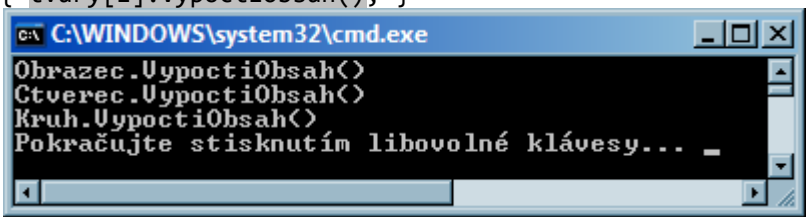
- = spojení volání metody s její definicí
- Časná vazba
  - = vazba při zápisu (překladu) programu
  - To, která metoda se zavolá, určuje programátor – normální volání metod
- Pozdní vazba
  - = za běhu programu
  - Jen OOP jazyky pomocí tzv. virtuálních metod, vytváří polymorfizmus za běhu programu

Polymorfizmus za běhu programu

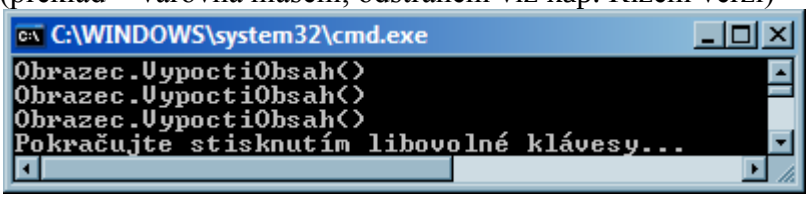
- Pomocí virtuálních metod
- Virtuální = „vypadá jako skutečné, ale není“
- Princip:
  - V bázové třídě definujeme virtuální metodu (`virtual`)
  - Ve třídách odvozených ji předefinujeme jinou metodou (stejná hlavička, označíme `override`)
  - Přetypování potomka na předka a zavolání virtuální metody (tedy `predek.Metoda()`) – vykoná se metoda potomka
- Příklad Tvary (bude využit ještě v kap. o abstraktních třídách a o řízení verzí):

```
class Obrazec
{
	// nějaké atributy
	public virtual double VypoctiObsah()
	{
		Console.WriteLine("Obrazec.VypoctiObsah()");
		return 0.0;  // obrazec nemá z čeho počítat obsah
	}
}
class Ctverec: Obrazec
{
	double strana;
	public Ctverec(double strana) { this.strana = strana; }
	public override double VypoctiObsah()
	{
```

```
                Console.WriteLine("Ctverec.VypoctiObsah()");
                return strana * strana;
        }
}
class Kruh: Obrazec
{
        double polomer;
        public Kruh(double polomer) { this.polomer = polomer; }
        public override double VypoctiObsah()
        {
                Console.WriteLine("Kruh.VypoctiObsah()");
                return Math.PI * polomer * polomer;
        }
}
// v Main()
Obrazec[] tvary = new Obrazec[3];
tvary[0] = new Obrazec();
tvary[1] = new Ctverec(10);
tvary[2] = new Kruh(1);
for (int i = 0; i < tvary.Length; i++)
{ tvary[i].VypoctiObsah(); }
```



- Ten samý příklad bez `virtual` a `override`
  (překlad = varovná hlášení, odstranění viz kap. Řízení verzí)



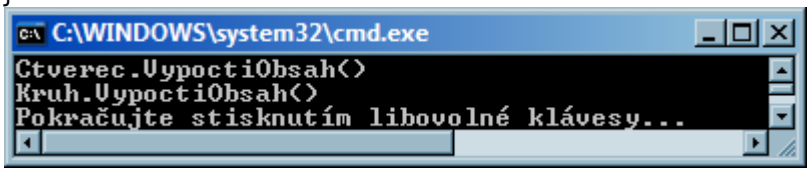
- Pozdní vazba je pomalejší než časná (režie na zjištění, která metoda se má zavolat)
- Kde nejde použít virtual (static??)

Abstraktní třídy

- Chyby příkladu Tvary:
  - Kód v metodě `Obrazec.VypoctiObsah()` je zbytečný (nic užitečného nedělá)
  - Definice `override` metody v odvozené třídě je nepovinná
  - Řešení: abstrakce
- Abstraktní třída (označena `abstract`) = třída, kde 1 nebo více metod (Vlastností) je abstraktních (označena `abstract`)
  - Jinak to může být běžná třída
  - **Nelze z ní vytvořit instanci !!!**
- Abstraktní metoda
  - Definována v bázové třídě
  - nesmí mít tělo (definováno až ve třídě odvozené)
  - nesmí být virtuální

- odvozená třída od abstraktní třídy:
  - musí implementovat všechny abstraktní metody (označit `override`)
- `abstract` opět realizuje polymorfizmus za běhu
- Příklad Tvary s abstraktní třídou (třídy `Ctverec` a `Kruh` jsou stejné – pouze musí implementovat `VypoctiObsah()`)

```
abstract class Obrazec
{
	public string nazev;	// nejaky atribut
	public abstract double VypoctiObsah(); // nemá tělo
}
class Ctverec: Obrazec
{
	double strana;
	public Ctverec(double strana) { this.strana = strana; }
	public override double VypoctiObsah()
	{
		Console.WriteLine("Ctverec.VypoctiObsah()");
		return strana * strana;
	}
}
class Kruh: Obrazec
{
	double polomer;
	public Kruh(double polomer) { this.polomer = polomer; }
	public override double VypoctiObsah()
	{
		Console.WriteLine("Kruh.VypoctiObsah()");
		return Math.PI * polomer * polomer;
	}
}
// v Main()
Obrazec[] tvary = new Obrazec[2];
tvary[0] = new Ctverec(10);
tvary[0].nazev = "ahoj";  // atribut „nazev“ zděděn od abstraktní třídy
tvary[1] = new Kruh(1);
for (int i = 0; i < tvary.Length; i++)
{
	tvary[i].VypoctiObsah();
}
```



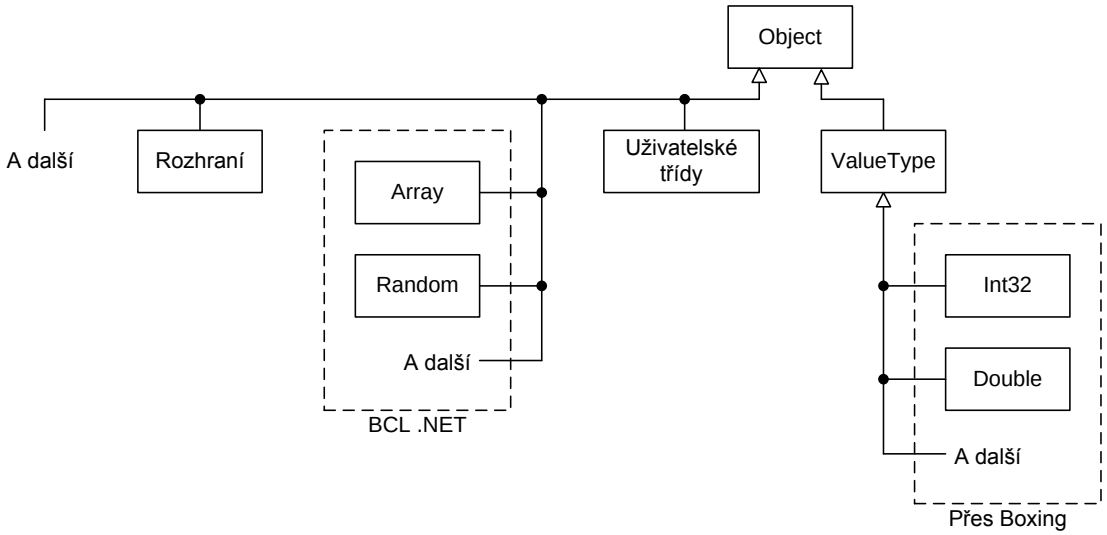
Řízení verzí

- new, base viz. MSDN keyword base
- na příkladu odstranění chyby o nepředefinování metody `VypoctiObsah()` viz příklad Tvary

Objektový model datových typů C#

- všechny typy odvozeny od `object` (alias pro `System.Object`)
  - i hodnotové datové typy !!! – přes tzv. boxing, unboxing (viz později)

- hierarchie typů – `Object` je kořen (root)



Třída `Object`

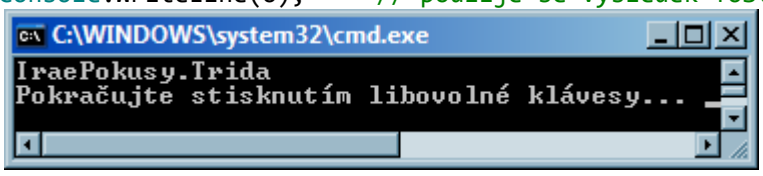
- obsahuje 4 `public` a 2 `protected` metody
- `public virtual bool Equals(Object obj)`
  - přetížení `public static bool Equals(Object objA, Object objB)`
  - referenční typy: vrací `true`, jestliže 2 referenční proměnné ukazují na tentýž objekt
  - hodnotové typy: porovnává hodnoty
  - možno v uživ. třídách přetížit tak, aby porovnávala jejich obsah podle přání tvůrce – vhodné (nutné) také přetížit operátory `==` a `!=` (podrobnosti viz MSDN)
- `public virtual int GetHashCode()`
- `public Type GetType()`
- `public static bool ReferenceEquals(Object objA, Object objB)`
  - vrací `true`, jestliže 2 referenční proměnné ukazují na tentýž objekt
- `public virtual string ToString()`
  - viz dále
- `protected void Finalize()`
  - = destruktor (`~Object()`) – opět „syntaktický cukr“
  - Volána automaticky při uvolnění objektu z paměti
- `protected Object MemberwiseClone()`
  - vytvoří mělkou kopii (shallow copy) objektu (nejprve nový objekt, potom kopie nestatických datových členů do nového objektu)

Metoda ToString()

- jako příklad redefinice virtuální metody z `object` v uživatelské třídě.
- Základní verze vrací datový typ objektu

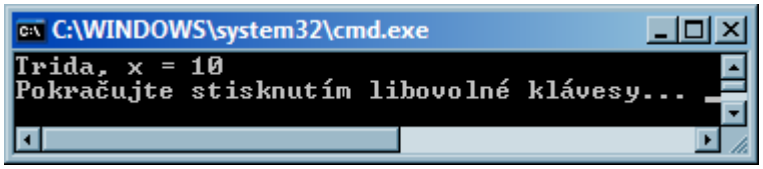
```
class Trida
{
      private int x;
```

```
        public int X
        {
                get { return x; }
        }
        public Trida(int x)
        {
                this.x = x;
        }
}
// v Main()
Trida o = new Trida(10);
Console.WriteLine(o);     // použije se výsledek ToString()
```



- Vhodné přetížit tak, aby vracela textovou reprezentaci objektu

```
// do „Trida“ doplnime
public override string ToString()
{
        return "Trida, x = " + x.ToString();
}
// jinak stejne
```

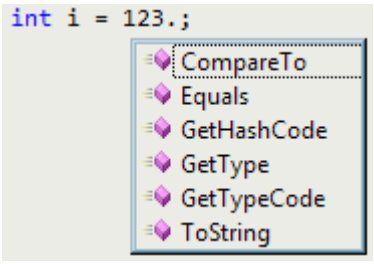


Boxing / unboxing

- Shrnutí referenční / hodnotové typy

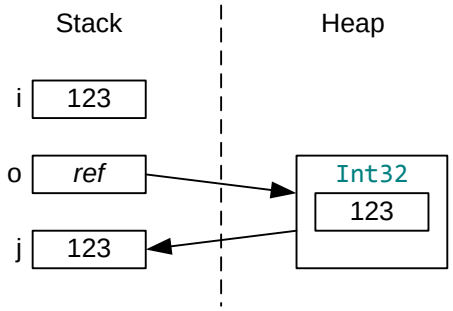
	Hodnotový typ	Referenční typ
Proměnná obsahuje:	Hodnotu	Odkaz
Uloženy na:	Zásobníku (stack)	Hromadě (heap)
Inicializace	`0`, `false` …	`null`
Přiřazení	Kopie hodnoty	Kopie odkazu

- Bylo řečeno: „vše je objekt“



  - Ne tak docela – číslo je číslo, jen když je třeba, lze z něj udělat objekt a obráceně
- Boxing = (implicitní) převod hodnotový typ → referenční (`object`)
- Unboxing – (explicitní) převod referenční typ → hodnotový

```
int i = 123;
object o = i;       // boxing (do Int32)
int j = (int)o;     // unboxing, přetypování nutné
```



- Využití:
  - Tisk hodnotové proměnné přes `Console.Write("promenna = {0}", i);`
    - Použije se `void Console.Write(string format, object arg0)`
    - Vlastní tisk přes (`virtual string`) `Object.ToString()`
  - datové struktury v BCL (viz později)

Přetěžování operátorů

- = forma polymorfizmu
- Příklad: komplexní čísla

```
class Complex
{
        private double re;
        private double im;

        public Complex(double re, double im)
        {
                this.re = re;
                this.im = im;
        }
}
// v Main
Complex c1 = new Complex(1, 2);
Complex c2 = new Complex(3, 1);
```

- Jak realizovat součet dvou instancí `Complex`?
  - Definovat instanční metodu

    ```
    Complex soucet = c1.Add(c2);     // nic moc
    ```
  - Definovat statickou metodu

    ```
    Complex soucet = Complex.Add(c1, c2);  // lepší
    ```
  - Přetížit aritmetický operátor

    ```
    Complex soucet = c1 + c2;        // nejelegantnější
    ```
- Přetížitelnost operátorů

Operátory	přetižitelnost
+, -, !, ~, ++, --, `true`, `false`	unární operátory – lze, `true`, `false` jen v páru.
+, -, *, /, %, &, |, ^, <<, >>	binární operátory – lze.

==, !=, <, >, <=, >=	relační operátory – lze, ale vždy jen v párech (== a !=, < a >, ≤ a ≥). Při přetížení == a != nutné předefinovat i virtuální metodu `Equals()` (zděděna z `Object`)
&&, ||	Logické operátory – nelze (ale realizováno automaticky překladačem při přetížení binárních & and |).
[]	Indexování polí – nelze, ale lze definovat tzv. indexer).
()	Operátory přetypování – nelze, ale lze definovat nové konverzní operátory (`explicit` a `implicit`).
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Složené přiřazení – nelze, ale automaticky zařídí překladač při přetížení příslušného binárního aritmetického operátoru.
=, ., ?:, ->, `new`, `is`, `sizeof`, `typeof`	nelze.

- Kdy přetěžovat? Třídy reprezentující matematické, grafické objekty
- Kdy nepřetěžovat?

```
Auto a1 = new Auto();
Auto a2 = new Auto();
Auto a3 = a1 * a2;        // nesmysl
```

- Jak to funguje? Představa → operátor = metoda
- Realizace: Definice veřejné, statické metody `operator ?`

Aritmetické operátory

- Příklady níže = doplnění `Complex`:
- Binární:

```
// binarni Complex + Complex
public static Complex operator +(Complex p1, Complex p2)
{
      return new Complex(p1.re + p2.re, p1.im + p2.im);
}
// binarni Complex + double
public static Complex operator +(Complex p1, double p2)
{
      return new Complex(p1.re + p2, p1.im + p2);
}
// binarni double + Complex
public static Complex operator +(double p1, Complex p2)
{
      return p2 + p1;
      // vyuzito "Complex operator +(Complex p1, double p2)"
}
```

  - Využití:

```
Complex c1 = new Complex(1, 2);
Complex c2 = new Complex(3, 1);
Complex s1 = c1 + c2;     //  4 + 3i
Complex s2 = c1 + 3.3;    //  4.3 + 5.3i
```

- Unární:

```
// unární -
public static Complex operator -(Complex p)
{
	p.re = -p.re; p.im = -p.im;
	return p;
}
// unární ++
public static Complex operator ++(Complex p)
{
	p.re++; p.im++;	// matematicky nesmysl
	return p;
}
```

  - Využití:

```
Complex c1 = new Complex(1, 2);
c1++;				//  2 + 3i
Complex minusC1 = -c1;	// -2 - 3i
```

Relační operátory

- Na příkladu == a!=
- Funkce operátorů bez jejich přetížení – porovnávají shodnost referencí

```
Complex c1 = new Complex(1, 2);
Complex c2 = new Complex(1, 2);
Complex c3 = c1;
if (c1 == c2)
	Console.WriteLine("c1 == c2");  // nevytiskne
if (c1 == c3)
	Console.WriteLine("c1 == c3");  // vytiskne
```

-
- Přetěžovací metoda **musí** vracet `bool`
- Doplnění `Complex`:

```
public static bool operator ==(Complex p1, Complex p2)
{
	// jsou-li oba null, nebo stejné reference
	if (Object.ReferenceEquals(p1, p2))
	{
		return true;
	}
	// je-li jeden z nich null
	if (((object)p1 == null) || ((object)p2 == null))
		// přetypování nutné, jinak rekurze a StackOverflowException
		return false;
	// porovnání dle obsahu
	return (p1.re == p2.re) && (p1.im == p2.im);
}
public static bool operator !=(Complex p1, Complex p2)
{
	return !(p1 == p2);
}
```

- V Main()

```
Complex c1 = new Complex(1, 2);
Complex c2 = new Complex(1, 2);
Complex c3 = c1;
Complex c4 = null;
if (c1 == c2)
```

```
        Console.WriteLine("c1 == c2");  // vytiskne
if (c1 == c3)
        Console.WriteLine("c1 == c3");  // vytiskne
if (c4 == null)
        Console.WriteLine("c4 == null");// vytiskne
```

- Jak je to s tou Equals()?