Dědičnost (inheritance)

Úvod

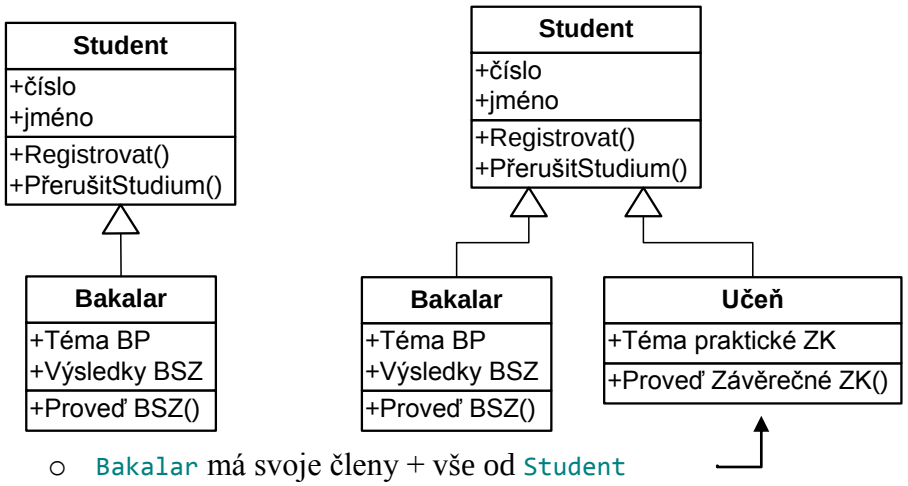
- Umožňuje objektům převzít (zdědit) členy jiných objektů a pouze je rozšířit
  - Auto: lze odvodit
- Vztah **„je“** – osobní auto, cisterna jsou auta

Základní pojmy

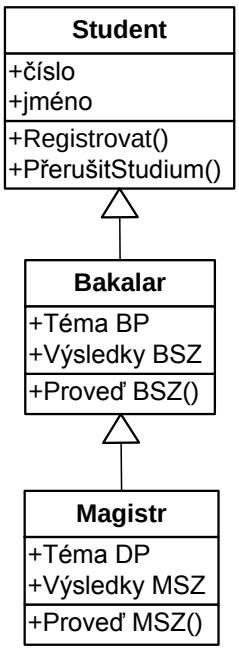
- Bázová třída (rodič) – base class, parent
- Odvozená třída (potomek) – derived class, children
- Hierarchie tříd – class hierarchy

Typy dědičnosti

Jednoduchá

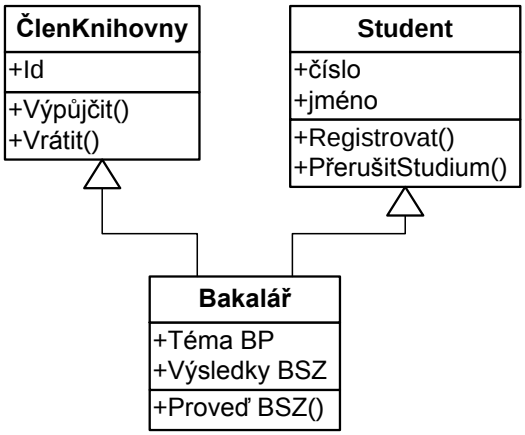


  - `Bakalar` má svoje členy + vše od `Student`
- třída může být rodičem pro několik odvozených tříd
- Násobná jednoduchá dědičnost – potomek je rodič pro další odvozenou třídu



Vícenásobná dědičnost

- potomek dědí od více rodičů
- C#, Java nelze, C++ ano



Dědičnost v C#

- Jen jednoduchá, vícenásobná pouze přes tzv. rozhraní – viz později

Řízení viditelnosti

- `protected` – chráněný člen
- viditelnost členů

Člen v rodiči deklarován jako:	Potomek vidí	Vně tříd (R i P)
`public`	Ano	Ano
`private`	Ne	Ne
`protected`	Ano	Ne

  - `private` – člen viditelný pouze ve třídě, kde byl definován

- `protected` – člen viditelný ve třídě, kde byl definován + ve všech odvozených třídách

Odvozené třídy

```
// definice bázové třídy - nic se nemění
class Bazova
{
	private int atrB1;
	protected int atrB2;
	public void MetodaB() { }
}
// definice odvozené třídy
class Odvozena: Bazova
{
	private int atrO1;
	public void MetodaO()
	{
		atrO1 = atrB2;	// atrB2 je protected
	}
}
// v Main()
Bazova b = new Bazova();
b.MetodaB(); // jediny clen viditelny zvenci "Bazova"
Odvozena o = new Odvozena();
o.MetodaB(); // metoda zdedena od "Bazova"
o.MetodaO();
```

Dědičnost a konstruktory

- při vzniku instance odvozené třídy se automaticky jako první provede **bezparametrický** konstruktor třídy bázové

```
class Bazova
{
	public Bazova()
	{
		Console.WriteLine("Bazova ctor");
	}
	public Bazova(int x)
	{
		Console.WriteLine("Bazova ctor({0})", x);
	}
}
class Odvozena: Bazova
{
	public Odvozena()
	{
		Console.WriteLine("Odvozena ctor");
	}
}
// v Main()
Odvozena o = new Odvozena();
```

  - vytiskne:

```
Bazova ctor
Odvozena ctor
```

- důvod: potomek přejímá atributy třídy bázové, ty musí projít inicializací. Konstruktor potomka jen inicializuje nově přidané atributy.

- násobná jednoduchá dědičnost – volají se všechny konstruktory počínaje první bázovou třídou a konče posledním potomkem
- volání parametrického konstruktoru bázové třídy ve třídě odvozené – `base`
  - (další využití `base` viz za později)
  - Nutno si vynutit ručně

```
class Bazova
{
	public Bazova(int x)
	{
		Console.WriteLine("Bazova ctor({0})", x);
	}
}
class Odvozena: Bazova
{
	public Odvozena(int y): base(5*y)
	{
		Console.WriteLine("Odvozena ctor({0})", y);
	}
}
// v Main()
Odvozena o = new Odvozena(10);
```

volání ctoru bázové třídy se sk. parametrem 5*y

  - vytiskne:

```
Bazova ctor(50)
Odvozena ctor(10)
```

- shrnutí:

`class Bazova`
`{`
`}`
`class Odvozena: Bazova`
`{`
`  public Odvozena(int x)`
`  {…}`
`}`
	`class Bazova`
	`{`
	`  public Bazova()`
	`  {…}`
	`}`
	`class Odvozena: Bazova`
	`{`
	`  public Odvozena(int x)`
	`  {…}`
	`}`

`// vytvoření instance:`
`Odvozena b = new Odvozena(5)`
`// provede se:`
`Výchozí Bazova()`
`Odvozena(5)`
	`// vytvoření instance:`
	`Odvozena b = new Odvozena(5)`
	`// provede se:`
	`Bazova()`
	`Odvozena(5)`

`class Bazova`
`{`
`  public Bazova(int x)`
`  {…}`
`}`
`class Odvozena: Bazova`
`{`
`  public Odvozena(int x)`
`  {…}`
`}`
	`class Bazova`
	`{`
	`  public Bazova(int x)`
	`  {…}`
	`}`
	`class Odvozena: Bazova`
	`{`
	`  public Odvozena(int x): base(x)`
	`  {…}`
	`}`

`// vytvoření instance:`
`Odvozena b = new Odvozena(5)`
`// provede se:`
`Compiler Error!!`
`error CS1501: No overload for method 'Bazova' takes '0' arguments`
	`// vytvoření instance:`
	`Odvozena b = new Odvozena(5)`
	`// provede se:`
	`Bazova(5)`
	`Odvozena(5)`

Rodič zastupuje potomka

- někdy může být výhodné (viz virtuální metody později)
- rodič může potomka zastoupit – má všechny jeho členy
- použito i v BCL. NET, např. `TextWriter tw = new StreamWriter("Soubor.txt"))`
  - `StreamWriter` je potomek `TextWriter`
- Jak provést?

```
class Bazova
{
        public void MB() { }
}
class Odvozena : Bazova
{
        public void MO() { }
}
```

  - Při deklaraci

```
// v Main
Bazova o1 = new Odvozena();
o1.MB();     // o.MO() nelze
```

- Za běhu (lepší přetypovat)

```
Odvozena o2 = new Odvozena();
Bazova o3 = o2;          // sice OK, ale nedělat
Bazova o4 = (Bazova)o2;  // lepší
```

- Objekt přetypovaný na předka lze zpětně přetypovat na potomka (nevyužívá se)

```
Odvozena o5 = (Odvozena)o4;     // Odv. o2 na Baz. o4 a zpět na Odv. o5
o5.MO();     // OK
```

- Předka nelze přetypovat na potomka (rozšiřujeme funkčnost)

```
Bazova o1 = new Bazova();
Odvozena o2 = (Odvozena)o1;     // InvalidCastingException
```

Uzavřené třídy

- modifikátor `sealed` – z třídy nelze odvodit žádnou další

```
sealed class Uzavrena
{
      // Definice členů
}
```

- výhoda – kompilátor může optimalizovat
- nevýhoda – „zavírám“ si dveře pro možné další rozšiřování
- závěr – používat s rozvahou

Dědičnost vs. Skládání tříd

- skládání („má“) převažuje nad dědičností („je“)

Výjimky II

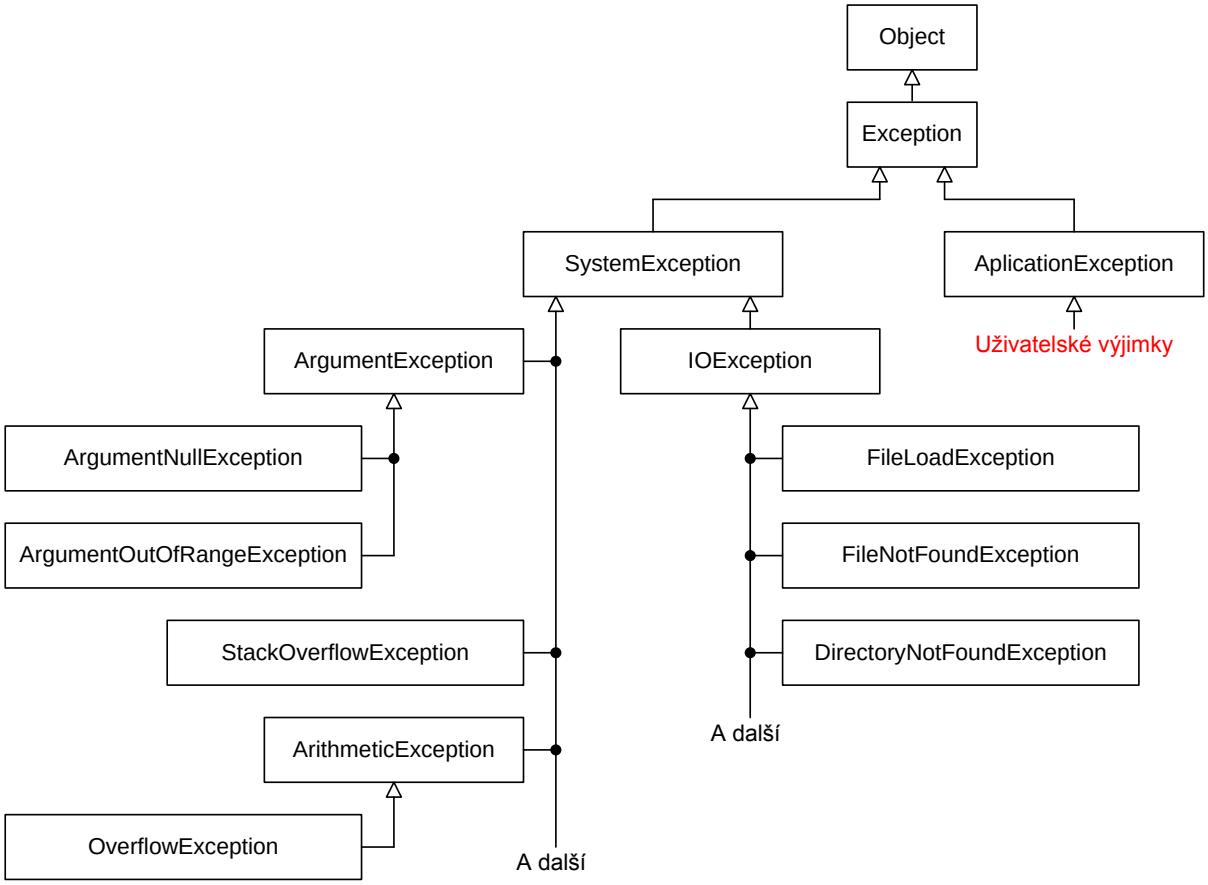
Opakování

- Princip:
  1) V programu dojde k běhové chybě
  2) Běh program se přeruší → **vyhodí se výjimka (throw exception)**
  3) Program se pokusí **výjimku zachytit** (**catch exception**) = ošetřit chybu ve spec. části kódu
- Zachycení výjimky v .NET:
  - výjimka = objekt
  - spousty typů výjimek → společný předek (viz. OOP příští-tento semestr), třída `Exception`
    - vlastnost `Message` (`string`) = popis výjimky

```
static void Main(string[] args)
{
      int x;
      try
      {
            int a = Int32.Parse(Console.ReadLine());
            // Int32.Parse() několik typu vyjimek
            x = 255 / a;      // možná výjimka typu
                              DivideByZeroException
      }
      catch (Exception e)
      {
            Console.WriteLine("Chyba!!!");
```

```
                Console.WriteLine(e.Message); // vytiskneme popis
                return;     // ukoncime program
            }
            Console.WriteLine("Vysledek deleni je {0}", x);
        }
```

Hierarchie výjimek v BCL



- `SystemException` – Systémové chyby (soubor se nepodařilo nalézt, argument metody musí být větší než nula, …)
- `ApplicationException` – pro uživatelské výjimky, příklad:

Vyhození vlastní výjimky

- Příkaz `throw`

```
class Trida
{
        public void SetVek(int vek)
        {
                if (vek < 0)
                {
                        throw new ArgumentOutOfRangeException
                                ("parametr vek musi byt > 0");
                }
                // ...
        }
        public void FindData(double[] data)
        {
                if (data == null)
                {
```

```
                        throw new ArgumentNullException("Pole data je prazdne");
                }
                // ...
        }
        // ...
}
```

Vlastní typy výjimek

- = třídy odvozené od `ApplicationException`
- Doporučeno (Microsoft) explicitně definovat 3 konstruktory
- Příklad:

```
class AutoException: ApplicationException
{
        public AutoException() { }
        public AutoException(string message): base(message) { }
        public AutoException(string message, Exception inner) :
                base(message, inner) { }
}
class Auto
{
        double aktRychlost;
        public enum Rychlost
        {
                Jedna, Dva, Tri, Neutral, Zpatecka
        }
        public void Zarad(Rychlost stupen)
        {
                if ((stupen == Rychlost.Zpatecka) && (aktRychlost > 20))
                {
                        throw new AutoException
                                ("Zaradil jsi zpatecku..., destrukce prevodovky");
                }
        }
}
```