# TŘÍDY – POKRAČOVÁNÍ

## Události – pokračování

***Příklad***

```
public delegate void ZmenaSouradnicEventHandler
   (object sender, EventArgs e);

class Bod
{
   private int x;
   private int y;

   public event ZmenaSouradnicEventHandler ZmenaSouradnic;
   public int X
   {
      get { return x; }
      set {
         if (x != value) {
            x = value;
            OnZmenaSouradnic(EventArgs.Empty);
         }
      }
   }
   public int Y
   {
      get { return y; }
      set {
         if (y != value) {
            y = value;
            OnZmenaSouradnic(EventArgs.Empty);
         }
      }
   }
   public Bod(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   protected virtual void OnZmenaSouradnic(EventArgs e)
   {
      ZmenaSouradnicEventHandler handler = ZmenaSouradnic;
      if (handler != null) handler(this, e);
   }
}
class Obrazek
{
   public static void ZmenaSouradnic(object sender, EventArgs e)
   {
      Console.WriteLine("Došlo ke změně souřadnic bodu");
      Console.WriteLine("Nové souřadnice bodu jsou: x = {0}, y = {1}",
         (sender as Bod).X, (sender as Bod).Y);
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      Bod bod = new Bod(10, 20);
      bod.ZmenaSouradnic +=
         new ZmenaSouradnicEventHandler(Obrazek.ZmenaSouradnic);
      // dtto od verze .NET 2.0:
      // bod.ZmenaSouradnic += Obrazek.ZmenaSouradnic;
      bod.X = 30;
      // Chyba - nelze volat událost mimo její třídu:
      // bod.ZmenaSouradnic(typeof(Program), new EventArgs());
      Console.ReadKey();
   }
}
```

V metodě `OnZmenaSouradnic` je vytvořena kopie události, aby nevznikla kolize při přístupu k události z více podprocesů. Pokud by se nevytvořila kopie, tak by po vyhodnocení výrazu v příkazu `if`, který testuje zda událost není `null`, mohl jiný podproces zrušit registraci handleru této události a pokud se jednalo o jediného předplatitele události, událost by měla hodnotu `null` a při jejím vyvolání by vznikla výjimka typu `System.NullReferenceException`. Lokální kopie události se sice odkazuje na stejnou instanci vícenásobného delegátu, ale pokud se k události registruje další handler, lokální kopie se již bude odkazovat na jinou instanci.

## Události jako vlastnosti

Obsahuje-li třída větší množství událostí a jejich handlerů, může být rozumné deklarovat je jako vlastnosti. To totiž umožňuje použít vlastní způsob ukládání handlerů, např. do hešovací tabulky nebo jiné kolekce. Událost se deklaruje jako vlastnost také v případě, pokud je potřebné ošetřovat volání každého zaregistrovaného handleru zvlášť.

Syntaxe:

*části_add_remove:*

*část_add část_remove*

*část_remove část_add*

*část_add:*

**add** *blok*

*část_remove:*

**remove** *blok*

*Blok* – složený příkaz.

*Část add* – přístupová metoda pro přidání handleru do kolekce handlerů.

*Část remove* – přístupová metoda pro odebrání handleru z kolekce handlerů.

Událost deklarovaná jako vlastnost musí obsahovat obě přístupové metody `add` a `remove` v libovolném pořadí. Obě se chovají jako metody, vracející `void` a mající jeden parametr typu delegát této události. Instance tohoto delegátu je v těchto metodách přístupná pomocí klíčového slova `value`.

### *Příklad*

Jedná se o předchozí příklad upravený tak, že událost `ZmenaSouradnic` je deklarována s přístupovými metodami `add` a `remove`, které přidají nebo odeberou handler z kolekce

`zmenaSouradnic`, což je instance vícenásobného delegátu. Pro zajištění bezpečného volání přístupových metod z jiných vláken, je tělo metody součástí příkazu `lock`.

```
class Bod
{
   private int x;
   private int y;
   private ZmenaSouradnicEventHandler zmenaSouradnic;

   public event ZmenaSouradnicEventHandler ZmenaSouradnic
   {
      add
      {
         lock (this) zmenaSouradnic += value;
      }
      remove
      {
         lock (this) zmenaSouradnic -= value;
      }
   }
   protected virtual void OnZmenaSouradnic(EventArgs e)
   {
      ZmenaSouradnicEventHandler handler = zmenaSouradnic;
      if (handler != null) handler(this, e);
   }
// zbytek stejný jako v předchozím příkladu
```

Je-li událost deklarována první variantou, tj. bez použití přístupových metod, překladač doplní přístupové metody a soukromou instanci vícenásobného delegátu tak, jak je uvedeno v příkladu včetně příkazu `lock`. Pros statickou událost deklarovanou první variantou by příkaz `lock` měl tvar `lock(typeof(Bod))`. Příkaz `lock` se nepoužije pro události deklarované ve strukturách.

## Vnořené třídy

Součástí třídy (nebo struktury) může být deklarována vnořená třída (nebo struktura), která je stejně jako v C++ jednou z jejích složek, a tudíž se ni vztahují přístupová práva stejně jako na ostatní její složky.

V C++ metody vnější třídy nemají žádné výjimky při přístupu ke složkám vnitřní třídy a naopak.

V C# složky vnitřní třídy (struktury) mohou přistupovat ke všem složkám vnější třídy (struktury) včetně soukromých, zatímco složky vnější třídy (struktury) mohou přistupovat pouze k veřejným nebo interním složkám vnitřní třídy (struktury).

***Doporučení***

Doporučuje se nedeklarovat veřejné vnořené typy. Vnořené typy by měly být využívány jen vnějším typem.

***Příklad***

```
class A
{
   private int x;

   public class B
   {
      private int y;
      public void f()
      {
         A a = new A();
         a.x = 15; // OK
      }
   }
   public void g()
   {
      B b = new B();
      // b.y = 25; // Chyba
   }
}
```

Vytvoření instance vnořené třídy `B` mimo tělo vnější třídy lze provést příkazem

```
A.B b = new A.B();
```

## Dědičnost

Jazyk C# podporuje pouze jednoduchou dědičnost. Pokud v deklaraci nějaké třídy není uveden předek, doplní překladač jako přímého předka třídu `object`. Podobně jako v C++ i v C# může potomek zastoupit předka.

### Odvozená třída

Odvozeným objektovým typem v C# může být pouze třída. V C++ jím může být i struktura. V C# deklarace struktury nemůže obsahovat předka a od struktury nelze odvodit potomka.

Specifikace předka v odvozené třídě má tento tvar.

Syntaxe:

*předek*:

**:** *jméno*

*Jméno* – identifikátor bázové třídy, v případě potřeby kvalifikovaný jménem prostoru jmen.

Na rozdíl od C++ nelze v specifikaci předka uvést modifikátory přístupových práv a modifikátor `virtual`. Veškeré dědění v C# je veřejné.

Potomek zdědí všechny složky předka kromě konstruktorů (i statických) a destruktorů, a to bez ohledu na jejich přístupová práva. Ovšem některé složky nemusí být v odvozené třídě přístupné. To se týká soukromých složek předka a dále interních složek předka, je-li potomek deklarován v jiném sestavení.

### Zastíněné složky

Žádnou ze zděděných složek nelze v potomkovi odstranit. Lze ji však zastínit tím, že se deklaruje složka s týmž identifikátorem (v případě metod s toutéž signaturou, tj. se stejným seznamem formálních parametrů) doplněným modifikátorem `new`. Zastínit lze i virtuální metody, vlastnosti, události nebo indexery (přetížené operátory indexování).

V C++ metoda potomka zastiňuje metodu předka tehdy, pokud má metoda potomka stejný identifikátor jako metoda předka bez ohledu na její parametry.

Zastíněním složky se neztrácí možnost přístupu ke zděděné složce. Zděděná složka se v takovém případě kvalifikuje klíčovým slovem `base`.

Klíčovým slovem `base` lze zpřístupnit pouze složky, které jsou dostupné v přímém předkovi dané třídy.

***Příklad***

```
class A
{
   public void f() {}
   public void g() {}
}
class B : A
{
   public new void f() // zastiňující metoda
   {
      base.f(); // volá se metoda f() předka
      // ...
   }
   public void g(int i) {} // OK – new se neuvádí
}
class Program
{
   static void Main(string[] args)
   {
      B b = new B();
      b.f();
      // b.base.f(); // Chyba
      ((A)b).f(); // OK - volá se metoda f třídy A
      b.g(); // #1 - v C# OK, v C++ chyba
      b.g(10); // #2 – OK v C# i C++
   }
}
```

V uvedeném příkladu se metoda `f()` třídy `B` musí deklarovat s modifikátorem `new`, protože má stejnou signaturu jako metoda `f()` třídy `A`. U metody `g(int i)` třídy `B` se naopak modifikátor `new` uvést nesmí, protože tato metoda má jinou signaturu než metoda `g()` třídy `A`. Příkaz #1 je v C# správný, kdežto v C++ je chybný, protože v C++ metoda `g(int i)` zastiňuje metodu `g()`. Příkaz #2 je správný jak v C#, tak i v C++.

V C# na rozdíl od C++ se při volbě, zda volat zastiňující metodu potomka nebo původní (zastíněnou) metodu předka, uplatňují přístupová práva. Jazyk C# se řídí filozofií „co není přístupné, o to nevím, to neberu v úvahu“. Filozofii jazyka C++ lze shrnout do věty „změna přístupových práv nesmí změnit chování programu“.

***Příklad***

```
class A
{
   public void f() {}
}
class B : A
{
   protected new void f() {}
}
```

```
class Program
{
   static void Main(string[] args)
   {
      B b = new B();
      b.f(); // #1 – V C# se volá metoda f třídy A, v C++ chyba
      Console.ReadKey();
   }
}
```

Příkaz #1 způsobí zavolání metody `f` třídy `A`, protože metoda `f` třídy `B` není přístupná. V C++ by tento příkaz byl chybný – překladač by oznámil, že metoda `f` třídy `B` není přístupná.

Změna přístupových práv v jazyce C# může vést k poměrně zásadní změně v chování programu – mohou se volat jiné metody.

## Zděděné a vlastní metody

Zděděné metody mají odlišné postavení než vlastní metody třídy stejného jména s odlišnou signaturou, byť vlastní metody s odlišnou signaturou nezastiňují metody předka.

***Příklad***

```
class A
{
   public void f(int i) { Console.WriteLine("A.f(int i)"); }
}
class B : A
{
   public void f(long i) { Console.WriteLine("B.f(long i)"); }
}
class Program
{
   static void Main(string[] args)
   {
      B b = new B();
      int i = 10;
      b.f(i); // #1 - volá f(long i)
      Console.ReadKey();
   }
}
```

Jak v C#, tak i v C++ příkaz #1 zavolá metodu `f(long i)` třídy `B`, protože typ `int` lze implicitně zkonvertovat na typ `long`.

## Konstruktor předka

Konstruktor odvozené třídy vždy nejprve volá konstruktor přímého předka. V inicializátoru konstruktoru potomka lze předepsat konstruktor předka, který se má zavolat. Není-li uveden, volá se konstruktor bez parametrů. Volání konstruktoru předka v inicializátoru se uvádí klíčovým slovem `base`, za nímž se uvede v kulatých závorkách seznam parametrů.

***Příklad***

```
class A
{
   int x;
   public A(int x) { this.x = x; }
}
```

```
class B : A
{
   int y;
   public B(int x, int y) : base(x) { this.y = y; }
}
```

## Polymorfismus

Dědění je jednou z cest, jak lze v C# implementovat polymorfní chování objektů. Alternativu představují rozhraní – viz dále. Podobně jako v C++ se v C# polymorfně chovají pouze složky, které jsou deklarovány jako virtuální. Virtuální mohou být metody, vlastnosti, události a indexery.

Virtuální složka třídy se v předkovi deklaruje s modifikátorem `virtual`. Předefinovaná virtuální složka třídy se v potomkovi deklaruje s modifikátorem `override` a musí mít stejná přístupová práva jako v předkovi.

Statické složky třídy nemohou být virtuální.

***Příklad***

```
class Vlak
{
   public void VypisNevirtualni() { Console.WriteLine("Vlak"); }
   public virtual void VypisVirtualni() { Console.WriteLine("Vlak"); }
}
class NakladniVlak : Vlak
{
   public new void VypisNevirtualni()
      { Console.WriteLine("Nakladní vlak"); }
   public override void VypisVirtualni()
      { Console.WriteLine("Nakladní vlak"); }
}

class Program
{
   static void Main(string[] args)
   {
      Vlak vlak1 = new Vlak();
      Vlak vlak2 = new NakladniVlak();
      vlak1.VypisNevirtualni();
      vlak2.VypisNevirtualni();
      vlak1.VypisVirtualni();
      vlak2.VypisVirtualni();
      Console.ReadKey();
   }
}
```

V uvedeném příkladu má bázová třída `Vlak` nevirtuální metodu `VypisNevirtualni`, která je v potomkovi `NakladniVlak` zastíněna. Pokud se tato metoda zavolá pro referenci typu `Vlak`, vždy se zavolá metoda třídy `Vlak`, ať se tato reference odkazuje na instanci třídy `Vlak` nebo třídy `NakladniVlak`.

Třída `Vlak` dále obsahuje virtuální metodu `VypisVirtualni()`, která je v třídě `NakladniVlak` předefinována. Pokud se tato metoda zavolá pro referenci typu `Vlak`, která se odkazuje na instanci třídy `NakladniVlak`, zavolá se její předefinovaná verze v třídě `NakladniVlak`.

Výpis programu bude následující:

```
Vlak
Vlak
Vlak
Nakladní vlak
```

Nelze deklarovat metodu (vlastnost, událost nebo indexer) s modifikátory `new override`. Lze však deklarovat metodu s modifikátory `new virtual`, která znamená přerušení hierarchie virtuálních metod a zavedení nové virtuální metody, která zastiňuje virtuální metodu předka, s níž má společné pouze jméno.

## Destruktory

Na rozdíl od C++ nemůže být uveden modifikátor `virtual` u destruktoru. Destruktor je automaticky virtuální, protože představuje předefinovanou virtuální metodu `Finalize()` třídy `object`.

Při destrukci dané instance se nejprve zavolá destruktor potomka, vykonají se jeho příkazy a potom se automaticky zavolá destruktor předka. Programátor volání předka nespecifikuje.

Překladač přeloží destruktor třídy `A`

```
class A
{
   ~A()
   {
      // příkazy destruktoru třídy A
   }
}
```

do tvaru

```
class A
{
   protected override void Finalize()
   {
      try {
         // příkazy destruktoru třídy A
      }
      finally {
         base.Finalize();
      }
   }
}
```

## Abstraktní třídy a metody

Abstraktní třída je třída, pro kterou nelze vytvořit instanci. Abstraktní třída může ale nemusí obsahovat abstraktní metody nebo jiné abstraktní složky. Pokud třída obsahuje abstraktní metodu nebo jinou abstraktní složku, musí být tato třída abstraktní.

Abstraktní třída se deklaruje s modifikátorem `abstract`.

Abstraktní metoda je virtuální metoda, která nemá tělo. Deklaruje se s modifikátorem `abstract` bez modifikátoru `virtual`. Místo jejího těla je uveden středník. V odvozené třídě musí být předefinována. Pokud není v odvozené třídě předefinována, musí být odvozená třída deklarována také jako abstraktní.

Kromě metod mohou být abstraktní také vlastnosti, události a indexery. Jejich deklarace je stejná jako u abstraktní metody. U abstraktní vlastnosti mají přístupové metody `get` a `set` místo svého těla středník.

***Doporučení***

Doporučuje se, aby abstraktní třída neobsahovala veřejný nebo chráněný interní konstruktor, protože nelze vytvořit její instanci. Konstruktor by měl být pouze chráněný nebo interní.

***Příklad***

```
abstract class A
{
   public abstract int X { get; set; }
   public abstract void f();
}
abstract class B : A // musí být abstraktní
{
   private int x;
   public override int X
   {
      get { return x; }
      set { x = value; }
   }
   public void g() { Console.WriteLine("Metoda g třídy B"); }
}
class C : B
{
   public override void f() { Console.WriteLine("Metoda f třídy C"); }
}
class Program
{
   static void Main(string[] args)
   {
      A a = new C();
      a.X = 10; // zavolá přístupovou metodu vlastnosti X třídy B
      a.f(); // zavolá metodu f() třídy C
      Console.ReadKey();
   }
}
```

### Zapečetěné třídy a složky

Zapečetěná třída je třída deklarovaná s modifikátorem `sealed`. Od zapečetěné třídy nelze odvodit potomka.

Modifikátor `sealed` lze také použít v deklaraci virtuální složky třídy, tj. virtuální metody, vlastnosti, události nebo indexeru. V takovém případě se jedná o zapečetěnou složku, kterou nelze v potomkovi předefinovat. Zapečetit lze pouze složku s modifikátorem `override`, nikoli složku s modifikátorem `virtual` nebo nevirtuální složku.

Abstraktní třídy a abstraktní složky nemohou být zapečetěné.

Zapečetěné jsou např. třídy `string` a `StringBuilder`.

## Statické třídy

Od verze .NET 2.0 lze deklarovat tzv. statickou třídu. Statická třída může obsahovat jen statické složky. Nelze vytvořit její instanci a nemůže být předkem jiné třídy. Deklaruje se s modifikátorem `static`.

***Příklad***

```
static class A
{
   static private int x;
   // void g() { } // Chyba - třída je statická
   static public int X
   {
      get { return x; }
      set { x = value; }
   }
   public static void f()
   {
      Console.WriteLine("x = " + x);
   }
}
class Program
{
   static void Main(string[] args)
   {
      // A a = new A(); // Chyba - nelze vytvořit instanci statické třídy
      A.X = 10;
      A.f();
      Console.ReadKey();
   }
}
```

## Částečné třídy

Od verze .NET 2.0 lze rozdělit deklaraci třídy, struktury nebo rozhraní do několika zdrojových souborů. V takovém případě se ve všech deklaracích třídy, struktury nebo rozhraní musí uvést modifikátor `partial`.

Vývojové prostředí Visual Studio využívá částečné třídy např. pro formuláře, které mají generovaný kód umístěn do samostatného souboru.

Rozdělení třídy na části lze využít, pokud více programátorů pracuje na jedné velké třídě. Každý programátor potom vytváří složky této třídy v samostatném souboru.

Složky deklarované v jedné části jsou dostupné v ostatních částech částečného typu (třídy, struktury, rozhraní).

Specifikaci předka může obsahovat některá nebo všechny části třídy. Jednotlivé části třídy mohou obsahovat odlišný seznam rozhraní, která implementují. V tom případě třída implementuje všechna rozhraní, která jsou uvedena v jednotlivých částech třídy.

Obdobně atributy částečného typu (třídy, rozhraní, struktury) mohou být specifikovány odlišně v jednotlivých deklaracích. Typ má potom všechny atributy, které jsou uvedeny u částí jeho deklarace.

Všechny části třídy (struktury, rozhraní) musí mít stejný modifikátor přístupových práv nebo u některé části může být vynechán a všechny části musí být deklarovány ve stejném sestavení.

Je-li jedna část třídy deklarována jako abstraktní nebo zapečetěná, je celá třída abstraktní nebo zapečetěná.

Vnořené typy mohou být také deklarovány s modifikátorem `partial`, a to i v případě, že vnější typ není deklarován po částech.

***Příklad***

```
// Soubor B1.cs
[Atribut1]
partial class B : A
{
   public void f(){}
   // ...
}

// Soubor B2.cs
[Atribut1]
[Atribut2]
public partial class B // nemusí obsahovat specifikaci předka A
{
   public void g(){}
   // ...
}
```

Při překladu projektu, jehož součástí jsou zdrojové soubory `B1.cs` i `B2.cs`, bude vytvořena jediná třída `B`, která bude veřejná, jejím předkem bude třída `A`, bude mít atributy `Atribut1` a `Atribut2` a bude obsahovat metody `f()` a `g()`.