ROZHRANÍ

Rozhraní (angl. *interfaces*) nemají v C++ bezprostřední analogii. Obsahují seznam signatur metod, indexerů, vlastností a událostí bez těla. Třída nebo struktura může rozhraní implementovat, tj. zavázat se, že implementuje jeho složky.

V C++ si lze rozhraní přibližně představit jako abstraktní třídu, která obsahuje pouze čisté virtuální metody.

Deklarace rozhraní

Syntaxe:

deklarace rozhraní:

modifikátory$_{nep}$ **partial**$_{nep}$ **interface** *jméno předek*$_{nep}$ **{** *složky_rozhraní* **}** **;**$_{nep}$

předek:

: *seznam_rozhraní*

Modifikátory – specifikují přístupová práva. U rozhraní, deklarovaného uvnitř třídy lze také použít modifikátor `new`.

Jméno – identifikátor deklarovaného rozhraní. Podle konvence pojmenování má začínat velkým písmenem `I`, za nímž následuje vlastní jméno rozhraní s velkým počátečním písmenem, např. `IEnumerable`.

Seznam_rozhraní – seznam rozhraní oddělených čárkou, která představují předky deklarovaného rozhraní. Deklarované rozhraní obsahuje všechny složky svých předků.

Složky_rozhraní – seznam metod, indexerů, vlastností a událostí. Místo těla metody, přístupové metody vlastnosti a indexeru se uvádí středník. Jde vlastně o deklaraci odpovídající deklaraci abstraktních metod, indexerů a vlastností. V deklaraci složek nelze použít jakékoli modifikátory, všechny složky jsou v rozhraní veřejné.

Rozhraní nemůže obsahovat deklaraci konstruktoru nebo destruktoru ani datových složek. Nelze vytvořit instanci rozhraní.

Implementace rozhraní

Pokud třída nebo struktura implementuje nějaká rozhraní, uvádí se za jménem třídy nebo struktury v jejich deklaraci.

Syntaxe specifikace předka třídy:

předek:

: *jméno_předka*

: *seznam_rozhraní*

: *jméno_předka* **,** *seznam_rozhraní*

Syntaxe specifikace implementovaných rozhraní struktury:

rozhraní_struktury:

: *seznam_rozhraní*

Jméno_předka – identifikátor bázové třídy.

Seznam_rozhraní – seznam rozhraní oddělených čárkou, které třída nebo struktura implementuje.

Třída může být odvozena od jiné třídy nebo implementovat jedno nebo více rozhraní nebo obojí. V takovém případě se nejprve uvádí jméno předka a potom seznam implementovaných rozhraní.

Struktura nemůže mít předka, ale může implementovat jedno nebo více rozhraní.

Třída nebo struktura (dále jen třída), která implementuje nějaké rozhraní, musí obsahovat deklaraci všech složek daného rozhraní. Tyto složky již mohou obsahovat obvyklé modifikátory, jako jiné složky třídy, ale musí být deklarovány jako veřejné (s modifikátorem `public`). Tyto složky však nejsou v dané třídě automaticky virtuální nebo abstraktní. Pokud mají být virtuální nebo abstraktní, musí se deklarovat s modifikátorem `virtual` nebo `abstract`.

Rozhraní jako typ

Jestliže třída `T` implementuje rozhraní `R`, je každá instance třídy `T` také instancí rozhraní `R` a lze s ní zacházet prostřednictvím referencí na `R`. Jedná se o analogii v případě dědičnosti, kdy potomek může zastoupit předka.

Příklad

```
interface IVypisovací
{
   void Vypis();
}
interface IBod : IVypisovací
{
   int X { get; set; }
   int Y { get; set; }
   event EventHandler ZmenaSouradnic;
}
class Bod : IBod
{
   int x, y;
   public int X
   {
      get { return x; }
      set
      {
         if (x != value) {
            x = value;
            OnZmenaSouradnic(EventArgs.Empty);
         }
      }
   }
   public int Y
   {
       // analogie vlastnosti X
   }
   public event EventHandler ZmenaSouradnic;
   public void Vypis()
   {
      Console.WriteLine("x = {0}, y = {1}", x, y);
   }
   protected virtual void OnZmenaSouradnic(EventArgs e)
   {
      EventHandler handler = ZmenaSouradnic;
      if (handler != null) handler(this, e);
   }
} // konec Bod
```

```
class Sestava : IVypisovací
{
   public void Vypis()
   {
      Console.WriteLine("Výpis údajů sestavy");
   }
}
class Program
{
   static void ProvedVypis(IVypisovací iv)
   {
      iv.Vypis();
   }
   static double VzdalenostBodu(IBod bodA, IBod bodB)
   {
      return Math.Sqrt(Math.Pow(bodA.X - bodB.X, 2) +
         Math.Pow(bodA.Y - bodB.Y, 2));
   }
   static void Main(string[] args)
   {
      Bod bodA = new Bod();
      bodA.X = 10; bodA.Y = 20;
      Bod bodB = new Bod();
      bodB.X = 30; bodB.Y = 40;
      Sestava sestava = new Sestava();
      ProvedVypis(sestava);
      ProvedVypis(bodA);
      ProvedVypis(BodB);
      Console.WriteLine("Vzdálenost bodů je {0}",
         VzdalenostBodu(bodA, bodB));
      Console.ReadKey();
   }
}
```

V příkladu je deklarováno rozhraní `IVypisovaci`, které implementuje třída `Sestava` a je předkem rozhraní `IBod`, které implementuje třída `Bod`. Metoda `Vypis` v třídě `Sestava` a `Bod` není virtuální. Metoda `ProvedVypis` pracuje s instancemi dvou naprosto rozdílných tříd, jejichž společným znakem je pouze to, že implementují rozhraní `IVypisovací`.

Explicitní implementace rozhraní

Může se stát, že dojde ke konfliktu mezi identifikátorem složky implementovaného rozhraní a identifikátorem složky třídy, která toto rozhraní implementuje. V takovém případě lze složku rozhraní implementovat explicitně.

Explicitně implementovaná složka rozhraní je v těle třídy kvalifikována jménem rozhraní. Na rozdíl od implicitně implementované složky rozhraní nemůže být v těle třídy deklarována jako abstraktní nebo virtuální a nemůže být deklarována s modifikátorem přístupu – je automaticky soukromá.

Jestliže se přistoupí k explicitně implementované složce rozhraní prostřednictvím instance třídy, použije se stejně pojmenovaná složka třídy. Pokud se k ní přistoupí prostřednictvím reference na rozhraní, použije se složka rozhraní.

Příklad

```
interface IVypisovací
{
   void Vypis();
}
```

```
class A : IVypisovací
{
   public void Vypis()
   {
      Console.WriteLine("Výpis údajů třídy A");
   }
}
class B : A, IVypisovací
{
   void IVypisovací.Vypis() // explicitní implementace
   {
      Console.WriteLine("Výpis údajů třídy B");
   }
}
class Program
{
   static void Main(string[] args)
   {
      B b = new B();
      b.Vypis(); // metoda A.Vypis
      ((IVypisovací)b).Vypis(); // metoda B.Vypis
      Console.ReadKey();
   }
}
```

Třída `B` obsahuje dvě metody `Vypis`. Jednu zděděnou z třídy `A` a druhou explicitně implementovanou z rozhraní `IVypisovací`. Voláním metody `Vypis` prostřednictvím instance třídy, se zavolá metoda `Vypis` třídy `A`. Má-li se zavolat explicitně implementovaná metoda `Vypis`, musí se volat pro referenci na rozhraní. Výpis programu bude následující:

```
Výpis údajů třídy A
Výpis údajů třídy B
```

Složku rozhraní má také smysl explicitně implementovat v případě, kdy je žádoucí, aby se k ní přistupovalo pouze pomocí reference na rozhraní a ne pomocí reference na instanci třídy. Např. v rozhraní je deklarována metoda `Pridej`, ale v třídě chceme deklarovat metodu `PridejNaKonec` a `PridejNaZacatek`. Rozhraní proto implementujeme explicitně a z metody `Pridej` voláme metodu např. `PridejNaKonec`.

Přetypování

Referenci na instanci třídy lze přetypovat na referenci na rozhraní, pokud daná třída implementuje dané rozhraní. Tato konverze může proběhnout implicitně. Např. příkaz

```
IVypisovací iv = b;
```

z předchozího příkladu je správný. Zatímco poslední z následujících příkazů

```
object o;
o = b;
IVypisovací iv = o; // Chyba – musí se přetypovat
```

je chybný a reference `o` se musí přetypovat na požadované rozhraní pomocí operátoru (*typ*) nebo `as`

```
IVypisovací iv = (IVypisovací)o; // OK
```

Kontrola správnosti přetypování proběhne za běhu programu. Nebude-li přetypování možné, vyvolá se výjimka typu `System.InvalidCastException`.

Lze také nejprve otestovat, zda je přetypování možné pomocí operátoru `is`

```
if (o is IVypisovací) iv = (IVypisovací)o;
else iv = null;
```

Podobně lze explicitně přetypovat referenci na jedno rozhraní na referenci na jiné rozhraní za předpokladu, že jde o referenci na instanci třídy, jež implementuje obě tato rozhraní. Implicitně se přetypuje reference na rozhraní `IB` na referenci na rozhraní `IA`, pokud rozhraní `IB` je odvozeno od rozhraní `IA`.

Předefinování metody rozhraní

Jestliže se v předkovi třídy implementuje metoda rozhraní, lze ji v potomkovi předefinovat nebo zastínit.

Virtuální metody

Deklaruje-li se metoda rozhraní v třídě jako virtuální a v potomkovi této třídy se předefinuje pomocí modifikátoru `override`, funkčnost je stejná jako v případě „obyčejných“ virtuálních metod.

Příklad

```
interface IVypisovací
{
   void Vypis();
}
class A : IVypisovací
{
   public virtual void Vypis()
   {
      Console.WriteLine("Výpis údajů třídy A");
   }
}
class B : A
{
   public override void Vypis()
   {
      Console.WriteLine("Výpis údajů třídy B");
   }
}
class Program
{
   static void Main(string[] args)
   {
      A a = new A();
      A b = new B();
      IVypisovací ia = a;
      IVypisovací ib = b;
      a.Vypis();
      b.Vypis();
      ia.Vypis();
      ib.Vypis();
      Console.ReadKey();
   }
}
```

Výpis programu bude podle očekávání následující:

```
Výpis údajů třídy A
Výpis údajů třídy B
Výpis údajů třídy A
Výpis údajů třídy B
```

Nevirtuální metody

V případě implementace metody `Vypis` v podobě nevirtuální metody se pro referenci na rozhraní volá metoda třídy, která jako první v dědické hierarchii tříd implementuje dané rozhraní.

Příklad

Jedná se o předchozí příklad, přičemž třídy jsou deklarovány následovně.

```
class A : IVypisovací
{
   public void Vypis()
   {
      Console.WriteLine("Výpis údajů třídy A");
   }
}
class B : A
{
   public new void Vypis()
   {
      Console.WriteLine("Výpis údajů třídy B");
   }
}
```

Program zavolá metodu `Vypis` třídy `A` ve všech uvedených případech. Výpis programu bude následující:

```
Výpis údajů třídy A
Výpis údajů třídy A
Výpis údajů třídy A
Výpis údajů třídy A
```

Metoda `Vypis` třídy `B` by se zavolala pouze pro proměnnou typu `B`:

```
B b2 = new B();
b2.Vypis(); // metoda B.Vypis()
```

Pokud je žádoucí, aby se prostřednictvím reference na rozhraní volala nevirtuální zastiňující metoda potomka, musí se znovu implementovat rozhraní v potomkovi (implicitně nebo explicitně).

Příklad

Jedná se o předchozí příklad s následující deklarací tříd.

```
class A : IVypisovací
{
   public void Vypis()
   {
      Console.WriteLine("Výpis údajů třídy A");
   }
}
```

```
class B : A, IVypisovací
{
   public new void Vypis()
   {
      Console.WriteLine("Výpis údajů třídy B");
   }
}
```

Výpis programu bude následující:

```
Výpis údajů třídy A
Výpis údajů třídy A
Výpis údajů třídy A
Výpis údajů třídy B
```

Předdefinovaná rozhraní

Knihovna BCL obsahuje řadu předdefinovaných rozhraní. Některá z nich jsou uvedena v této kapitole. Uvedená rozhraní jsou součástí prostoru jmen `System`.

IClonable

```
public interface ICloneable {
   object Clone();
}
```

Slouží pro klonování instancí tříd nebo struktur. Metoda `Clone` může zavolat chráněnou metodu `MemberwiseClone` třídy `object`, která provede mělkou kopii, nebo může definovat vlastní způsob klonování, který se postará o hloubkovou kopii.

Příklad

```
class Bod : ICloneable
{
   int x, y;
   public int X
   {
      get { return x; }
      set { x = value; }
   }
   public Bod(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   public virtual object Clone()
   {
      return MemberwiseClone();
   }
   public override string ToString()
   {
      return "(" + x + ", " + y + ")";
   }
   // ...
}
class Usek : ICloneable
{
   Bod pocatek, konec;
   int delka;
```

```
   public Bod Pocatek
   {
      get { return pocatek; }
      set { pocatek = value; }
   }
   public Usek(int x1, int y1, int x2, int y2, int delka)
   {
      pocatek = new Bod(x1, y1);
      konec = new Bod(x2, y2);
      this.delka = delka;
   }
   public virtual object Clone()
   {
      Usek t = (Usek)MemberwiseClone();
      t.pocatek = (Bod)pocatek.Clone();
      t.konec = (Bod)konec.Clone();
      return t;
   }
   public override string ToString()
   {
      return "Počátek: " + pocatek + ", konec: " + konec +
         ", délka: " + delka;
   }
   // ...
}
class Program
{
   static void Main(string[] args)
   {
      Usek u1 = new Usek(1, 2, 3, 4, 100);
      Usek u2 = (Usek)u1.Clone();
      u2.Pocatek.X = 10;
      Console.WriteLine(u1);
      Console.WriteLine(u2);
      Console.ReadKey();
   }
}
```

Instanci třídy `Bod` lze kopírovat po složkách, proto je v metodě `Clone` volána pouze metoda `MemberwiseClone`. V třídě `Usek` by metoda `MemberwiseClone` provedla pouze kopii referencí na instance bodů, nikoli kopii jejich složek. Proto se nejprve zavolá metoda `MemberwiseClone`, která mj. zkopíruje datovou složku `delka` a potom se volá metoda `Clone` pro oba body. Metody `Clone` jsou deklarovány jako virtuální, aby byla zajištěna jejich správná funkčnost i pro případné odvozené třídy.

IComparable

```
public interface IComparable {
   int CompareTo (object obj);
}
```

Rozhraní implementují typy, jejichž instance lze nějakým způsobem seřadit. Metoda `CompareTo` má vracet:

- zápornou hodnotu, je-li instance `this` menší než instance `obj`,
- nulu, je-li instance `this` rovna instanci `obj`,
- kladnou hodnotu, je-li instance `this` větší než instance `obj`.

Rozhraní `IComparable` např. vyžaduje pro prvky kolekce jedna z metod `Sort` a `BinarySearch` tříd `System.Array` a `System.Collections.ArrayList`. Toto rozhraní implementují všechny základní datové typy v C#.

IComparer

```
public interface IComparer {
   int Compare(object x, object y);
}
```

Slouží ke stejnému účelu jako rozhraní `IComparable`. Metoda `Compare` má vracet:

- zápornou hodnotu, je-li instance `x` menší než `y`,
- nulu, je-li instance `x` rovna `y`,
- kladnou hodnotu, je-li instance `x` větší než `y`.

Rozhraní `IComparer` je např. parametrem jedné z metod `Sort` a `BinarySearch` tříd `System.Array` a `System.Collections.ArrayList`.

Příklad

Příklad demonstruje použití rozhraní `IComparer` pro třídění pole prvků typu `Osoba` podle čísla nebo jména osoby.

```
class Osoba
{
   private int cislo;
   private string jmeno;

   public int Cislo
   {
      get { return cislo; }
      set { cislo = value; }
   }
   public string Jmeno
   {
      get { return jmeno; }
      set { jmeno = value; }
   }
   public Osoba(int cislo, string jmeno)
   {
      this.cislo = cislo;
      this.jmeno = jmeno;
   }
   public static int CompareCislo(Osoba x, Osoba y)
   {
      return x.cislo.CompareTo(y.cislo);
   }
   public static int CompareJmeno(Osoba x, Osoba y)
   {
      return x.jmeno.CompareTo(y.jmeno);
   }
   public override string ToString()
   {
      return cislo + "\t" + jmeno;
   }
}
delegate int CompareOsobaCallback(Osoba x, Osoba y);
```

```
class OsobaComparer : IComparer
{
   CompareOsobaCallback compare;
   public OsobaComparer(CompareOsobaCallback compare)
   {
      this.compare = compare;
   }
   public int Compare(object x, object y)
   {
      return compare((Osoba)x, (Osoba)y);
   }
}
class Program
{
   static void Vypis(Osoba[] osoby, string text)
   {
      Console.WriteLine(text);
      foreach (Osoba item in osoby) {
         Console.WriteLine(item.ToString());
      }
   }
   static void Main(string[] args)
   {
      Osoba[] osoby = new Osoba[] {
         new Osoba(10, "Karel"),
         new Osoba(5, "Pavel"),
         new Osoba(20, "Jana"),
         new Osoba(1, "Soňa") };
      Vypis(osoby, "Původní seznam");
      Array.Sort(osoby, new OsobaComparer(Osoba.CompareCislo));
      Vypis(osoby, "Utříděný seznam podle čísla");
      Array.Sort(osoby, new OsobaComparer(Osoba.CompareJmeno));
      Vypis(osoby, "Utříděný seznam podle jména");
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
Původní seznam
10      Karel
5       Pavel
20      Jana
1       Soňa
Utříděný seznam podle čísla
1       Soňa
5       Pavel
10      Karel
20      Jana
Utříděný seznam podle jména
20      Jana
10      Karel
5       Pavel
1       Soňa
```

IDisposable

```
public interface IDisposable {
   void Dispose ();
}
```

Rozhraní implementují třídy, které zapouzdřují neřízené prostředky, jako např. soubory, síťová a databázová připojení. Tyto prostředky neumí uvolnit automatická správa paměti a musí je uvolnit sama třída, která je zapouzdřuje. To lze provést na dvou místech:

- v destruktoru,
- v metodě `Dispose`.

Destruktor se volá při uvolnění instance třídy z paměti automatickou správou paměti. Metoda `Dispose` rozhraní `IDisposable` je volána buď přímo nebo automaticky prostřednictvím příkazu `using`.

Uživatel by měl uvolnit neřízené prostředky v okamžiku, kdy je již nepotřebuje, a to voláním metody `Dispose` přímo nebo prostřednictvím příkazu `using`. Pokud však zapomene metodu `Dispose` zavolat, měla by existovat bezpečnostní pojistka, která zajistí uvolnění neřízených prostředků automaticky. To se provede v destruktoru v okamžiku úklidu paměti.

Uvolnění neřízených prostředků se tedy doporučuje provést na obou uvedených místech. Třída, která zapouzdřuje neřízené prostředky resp. jiné třídy, které implementují rozhraní `IDisposable`, by měla být definována podle návrhového vzoru uvedeného v následujícím příkladu.

Příklad

```
public class ResourceHolder : IDisposable
{
   private bool isDisposed;
   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }
   protected virtual void Dispose(bool disposing)
   {
      if (!isDisposed) {
         if (disposing) {
            // Úklid řízených prostředků voláním jejich metod Dispose()
         }
         // Úklid neřízených prostředků
         isDisposed = true;
      }
   }
   ~ResourceHolder()
   {
      Dispose(false);
   }
   public void NějakáMetoda()
   {
      if (isDisposed)
         throw new ObjectDisposedException("ResourceHolder");
      // implementace metody
   }
}
```

```
public class DerivedResourceHolder : ResourceHolder
{
   protected override void Dispose(bool disposing)
   {
      if (disposing) {
         // Úklid řízených prostředků voláním jejich metod Dispose()
      }
      // Úklid neřízených prostředků
      base.Dispose(disposing);
   }
}
```

Třída `ResourceHolder` obsahuje dvě přetížené verze metody `Dispose`. Metoda `Dispose(bool)` provádí vlastní uvolnění neřízených prostředků. Je volána z destruktoru s hodnotou `false` a z metody `Dispose()` s hodnotou `true`. Tento parametr určuje místo, odkud byla metoda volána. Metoda `Dispose(bool)` by neměla být volána ze žádného jiného místa kódu, než je uvedeno.

Pokud je metoda `Dispose(bool)` volána z metody `Dispose()`, provede uvolnění neřízených i řízených prostředků této třídy. Pokud je volána z destruktoru, provede uvolnění pouze neřízených prostředků. Řízené prostředky, tj. reference na instance jiných tříd, které zapouzdřují neřízené prostředky, se v destruktoru třídy `ResourceHolder` neuvolňují, protože pro tyto třídy automatická správa paměti zavolá jejich destruktor, který uvolnění prostředků provede.

Datová složka `isDisposing` udává, zda instance třídy `ResourceHolder` byla již zlikvidována. Pokud by uživatel zavolal metodu `Dispose()` vícekrát, jen první její volání způsobí uvolnění prostředků, další volání již nic neprovede. Na začátku každé veřejné metody třídy `ResourceHolder` byl se mělo testovat, zda instance této třídy byla zlikvidována a pokud tomu tak je, měla by se vyvolat výjimka typu `System.ObjectDisposedException` s parametrem obsahujícím název zlikvidované třídy – viz metoda `NějakáMetoda`.

Metoda `Dispose()` po zavolání metody `Dispose(bool)` volá metodu `SuppressFinalize` třídy `System.GC`. Třída `GC` reprezentuje automatickou správu paměti a metoda `SuppressFinalize(object obj)` sděluje automatické správě paměti, že již není nutné volat destruktor pro instanci třídy `obj`.

Pokud potomek třídy `ResourceHolder` zapouzdřuje řízené nebo neřízené prostředky, stačí v něm předefinovat virtuální metodu `Dispose(bool)`.