Generické programování

- Od C# verze 2.0
- = vytváření kódu s obecným datovým typem
- Příklad – generická metoda, zamění dva parametry:

```
static void Swap<T>(ref T p1, ref T p2)
{
	T temp;
	temp = p1;
	p1 = p2;
	p2 = temp;
}
static void Main(string[] args)
{
	int a = 5, b = 10;
	double c = 5.0, d = -88;

	Swap<int>(ref a, ref b);
	// prekladac vytvori metodu void Swap(ref int p1, ref int p2)
	Swap<double>(ref c, ref d);
	// prekladac vytvori metodu void Swap(ref double p1, ref double p2)
}
```

- Důvody:
  - znovupoužitelnost kódu
  - typová bezpečnost, rychlost (není třeba užívat `object` (boxing) – lze zaměnit cokoli za cokoli!!!)
- Jako generické lze vytvořit: rozhraní, třídy, metody, události a delegáty
- Jako T může být i uživatelská třída

```
class Trida
{
	// implementace
}
// v Main:
Trida o1 = new Trida();
Trida o2 = new Trida();
Swap<Trida>(ref o1, ref o2);
```

Rozhraní (interface)

Úvod

- rozhraní (`interface`) = způsob, jak přidat různým věcem stejnou funkčnost
- příklad z windows – GUI: všechny programy mají stejné základní prvky a způsob ovládání (otevřít, uložit, ovládání myší …)
- v C# (a Java, C++ ne) = způsob, jak přidat vícenásobnou dědičnost
- = cca abstraktní třída se všemi členy abstraktními – implicitně všechny členy `public abstrakt` (`virtual`)
- může obsahovat:
  - Metody, Vlastnosti
  - Delegáty, události (později)

- Indexem

- Nesmí obsahovat: proměnné, konstanty, konstruktory, destruktory, operátory.
- **Třída může dědit od 1 třídy a od 1 a více rozhraní**
- Příklad – stroje v továrně, všechny by měly umět zastavit svou činnost (předpoklad: neexistuje společný předek pro všechny stroje):

```
interface IZastavit
{
	void Zastavit();
}
class Vrtacka: IZastavit  // třída implementuje rozhraní
{
	public void Vrtat()
	{
		Console.WriteLine("Vrtacka.Vrtat");
	}
	public void Zastavit()    // musí být definována
	{
		Console.WriteLine("Vrtacka.Zastavit");
	}
}
class Auto
{
	public void Jed() { }
}
class Cisterna: Auto, IZastavit // „vícenásobná dědičnost“
{
	public void Nacerpat()
	{
		Console.WriteLine("Cisterna.Nacerpat");
	}
	public void Zastavit()
	{
		Console.WriteLine("Cisterna.Zastavit");
	}
}
// v Main()
Vrtacka vrtacka = new Vrtacka();
Cisterna tanker = new Cisterna();
vrtacka.Zastavit();// přímé volání metody rozhraní -
                   // - sice lze, ale obvykle se nedělá
IZastavit zastaveni;
zastaveni = (IZastavit)vrtacka; // lepší (ale nejlepší je „as“ – viz dále)
zastaveni.Zastavit();           // vrtačka stojí
zastaveni = (IZastavit)tanker;
zastaveni.Zastavit();           // tanker také
```

- Rozhraní s Vlastnostmi:

```
interface IRozhrani
{
	int Delka
	{
		set;
		get;
	}
	string Nazev
	{
		get;
```

```
        }
}
```

Práce s rozhraním

- = přetypování objektu na dané rozhraní
- Ručně
  - `IZastavit zastaveni = (IZastavit)vrtacka;`
  - Problém: implementuje třída rozhraní? Pokud ne – neošetřená výjimka
- Operátor `is` = test, implementuje-li třída rozhraní (vrací `true` / `false`)

```
IZastavit zastaveni;
if (vrtacka is IZastavit)
{       // OK, můžeme přetypovat
        zastaveni = (IZastavit)vrtacka;
        zastaveni.Zastavit();     // vrtačka stojí
}
```

- Operátor `as` = `is` + přetypování (selže-li, vrací `null`, jinak odkaz na dané rozhraní) – preferovaný způsob

```
IZastavit zastaveni = vrtacka as IZastavit;
if (zastaveni != null)
{       // pokud vrtacka implementuje IZastavit
        zastaveni.Zastavit();     // vrtačka stojí
}
```

Více rozhraní

- Počet rozhraní, od kterých třída dědí, není omezen

```
interface IPrvni
{
        double Prvni();
}
interface IDruhe
{
        void Druha(int param);
}
class MyClass: IPrvni, IDruhe
{
        public double Prvni() { /* impl. Prvni() */ }
        public void Druha(int param) { /* impl. Druha() */ }
}
```

Skládání rozhraní

- = sdružení několika rozhraní do jednoho

```
interface I1
{
        void M1();
}
interface I2
{
        void M2();
}

interface I3 : I1, I2
{
}
```

```
class MyClass: I3
{
	public void M1() { }
	public void M2() { }
}
```

Dědění rozhraní

- Bázová třída implementuje rozhraní → odvozená jej samozřejmě zdědí (logické)
- Minulý příklad, jen změna

```
class Auto: IZastavit
{
	public void Jed() { }
	public void Zastavit()
	{
		Console.WriteLine("Auto.Zastavit");
	}
}
class Cisterna: Auto	// také obsahuje metodu Zastavit()
{
	public void Nacerpat()
	{
		Console.WriteLine("Cisterna.Nacerpat");
	}
}
// v Main()
Cisterna tanker = new Cisterna();
IZastavit zastaveni = tanker as IZastavit;
if (zastaveni != null)
{
	zastaveni.Zastavit();	// vypise „Auto.Zastavit“
}
```

- Pozor: uvažme tuto situaci (nedělat!!!)
  - Bázová třída implementuje rozhraní s metodou `Zastavit()`
  - Odvozená třída má svoji vlastní metodu `Zastavit()` (ale zdědila jiné `Zastavit()` také od rozhraní)

```
// třída Auto je stejná jako minule
class Cisterna: Auto
{
	public void Nacerpat()
	{
		Console.WriteLine("Cisterna.Nacerpat");
	}
	public new void Zastavit()	// bez new varování o překrytí metod
	{
		Console.WriteLine("Cisterna.Zastavit");
	}
}
// v Main()
Cisterna tanker = new Cisterna();
tanker.Zastavit();		// Cisterna.Zastavit
IZastavit zastaveni = tanker as IZastavit;
if (zastaveni != null)
{
	zastaveni.Zastavit();	// Auto.Zastavit
}
```

Explicitní implementace

- implementace dvou rozhraní se stejným členem = problém

```
interface IControl
{
	void Paint();
}
interface ISurface
{
	void Paint();
}
class SampleClass : IControl, ISurface
{
	public void Paint() { }   // ale ze ktereho rozhrani
}
```

- explicitní implementace = uvedení jména rozhraní při implementaci

```
public class SampleClass : IControl, ISurface
{
	void IControl.Paint() { }
	void ISurface.Paint() { }
}
```

  - která metoda se zavolá – určuje přetypování

Rozhraní v BCL

- BCL = spoustu rozhraní; jejich implementace = využití možností BCL, např. třídění, vyhledávání …
- Příklad rozhraní – `IComparable`
  - Vyžadováno např. metodami `Array.Sort()`, `Array.BinarySearch()`
  - Jediná metoda `int CompareTo(object obj)`
    - Porovnává aktuální instanci (její atribut) s `obj`
    - Musí vracet:
      - < 0, je-li instance < `obj`
      - 0, je-li instance == `obj`
      - > 0, je-li instance > `obj`
- Příklad: Třída `Nadoba`, bude schopna řazení podle svého objemu

```
class Nadoba: IComparable
{
	public double objem;
	public Nadoba(double objem)
	{
		this.objem = objem;
	}
	public int CompareTo(object obj)// implementace metody rozhraní
	{
		Nadoba dalsi = obj as Nadoba;
		if (dalsi == null) // test je velmi vhodny
		{
			throw new ArgumentException("obj neni typu Nadoba");
		}
		return this.objem.CompareTo(dalsi.objem);
		// porovnavame 2 cisla typu double (double zdědilo CompareTo()
		// z Object
	}
```

```
}
Nadoba[] sklenice = new Nadoba[3];
sklenice[0] = new Nadoba(10);
sklenice[1] = new Nadoba(0.5);
sklenice[2] = new Nadoba(5);
Array.Sort(sklenice);
```

- .NET 2.0 obsahuje také generická rozhraní – není nutné přetypování, rychlejší

```
class Nadoba: IComparable<Nadoba>
{
	public double objem;
	public Nadoba(double objem)
	{
		this.objem = objem;
	}
	public int CompareTo(Nadoba other)
	{
		return this.objem.CompareTo(other.objem);
	}
}
```

Co kdy použít

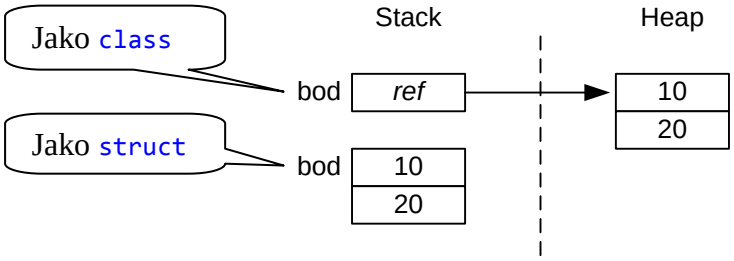
- proč tedy ne abstraktní třída?
  - Problém C# chybějící vícenásobné dědičnosti
  - Zdědění od třídy = zavření dveří pro další rozšiřování

Struktury

- Cca třída, ale **hodnotový typ**
- pro zapouzdření malého počtu spolu souvisejících proměnných – bod (x,y), Obdélník (x, y, š, v) nebo barva (složky RGB)
- Mohou obsahovat: konstruktory, metody, členské proměnné, Vlastnosti, indexery, události
- Nemohou obsahovat: bezparametrický konstruktor, destruktor, inicializované členské proměnné (jen přes konstruktor)
- Nelze aplikovat dědičnost

```
struct Bod
{
	double x;
	double y;
	public Bod(double x, double y)
	{
		this.x = x;
		this.y = y;
	}
}
// v Main()
Bod A = new Bod(10, 20);
```

- Umístění v paměti



- Struktury ve WinForms:

Structure	Description
CharacterRange	Specifies a range of character positions within a string.
Color	Represents an ARGB color.
Point	Represents an ordered pair of integer x- and y-coordinates that defines a point in a two-dimensional plane.
PointF	Represents an ordered pair of floating-point x- and y-coordinates that defines a point in a two-dimensional plane.
Rectangle	Stores a set of four integers that represent the location and size of a rectangle. For more advanced region functions, use a Region object.
RectangleF	Stores a set of four floating-point numbers that represent the location and size of a rectangle. For more advanced region functions, use a **Region** object.
Size	Stores an ordered pair of integers, typically the width and height of a rectangle.
SizeF	Stores an ordered pair of floating-point numbers, typically the width and height of a rectangle.