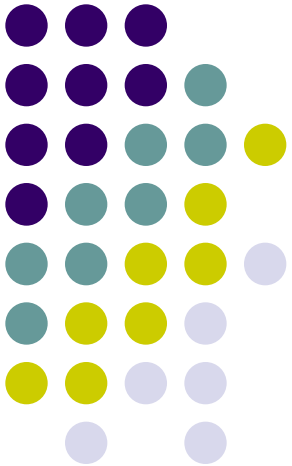
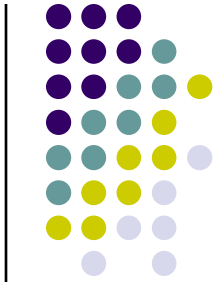
Programovací jazyk C#

Programové konstrukce

Ing. Marek Běhálek
Katedra informatiky FEI VŠB-TUO
A-1018 / 597 324 251
http://www.cs.vsb.cz/behalek
marek.behalek@vsb.cz

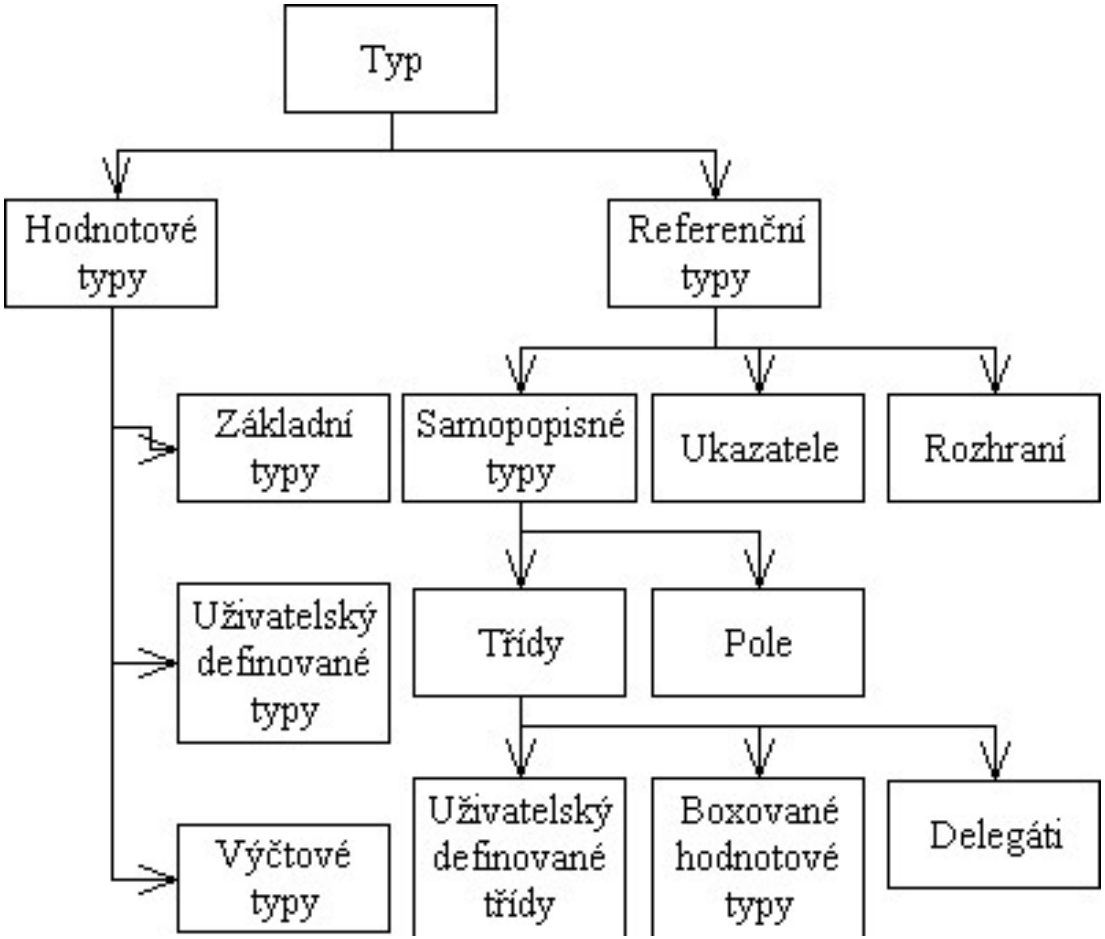


Náplň kapitoly



- Typový systém
- Základní prvky jazyka
  - Direktivy preprocesoru
  - Jmenné prostory
  - Třídy
  - Pole
  - Rozhraní
  - Operátory
  - Výjimky
  - Delegáti
  - …

Typový systém – Základní rozdělení



Typový systém – Základní charakteristika (1)

- C# využívá společný typový systém prostředí .NET (CTS)
- Základní dělení je na hodnotové a referenční typy.
- Automatická konverze z hodnotového na referenční (*boxing*) a z referenčního na hodnotový (*unboxing*).
- Objektově orientovaný typový systém. Vše je objekt. Vychází z třídy `System.Object`.

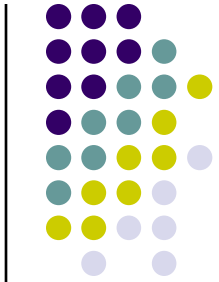
Typový systém – Základní charakteristika (2)

- C# se snaží být čistě objektový jazyk.
- C# obsahuje hodnotové typy (ty jsou důležité zejména pro rychlost provádění programu).
- Tento rozpor je vyřešen pomocí automatické konverze (*Boxing / Unboxing*).

```
object o;
SomeClass someObject = new SomeClass();
int a = 10;
o = someObject;
o = a;
```

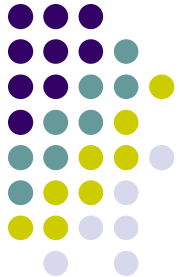
- V posledním kroku je vytvořena instance třídy *System.Int32*, do které je hodnotová proměnná *a* zabalena.

Typový systém – Hodnotové typy



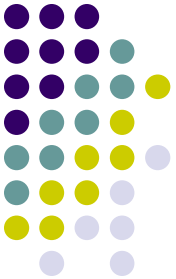
- Obsahují pouze hodnotu daného typu.
- Instancím hodnotového typu říkáme proměnné.
- Hodnoty tohoto typu jsou ukládány na zásobník.
- Jejich výhodou je zejména rychlost.
- Hodnotové typy musí být vždy inicializovány.
- Při přiřazení je přímo kopírována hodnota.
- Všechny hodnotové typy vycházejí z třídy System.ValueType.
- Nepodporují dědičnost.
- Hodnotové typy se dělí na:
  - Základní hodnotové typy;
  - Uživatelsky definované hodnotové typy – struktury, výčtové typy.

Typový systém – Celočíselné typy (1)



- `sbyte` (*System.Sbyte*) - znaménkový typ s rozsahem -128 až +127
- `byte` (*System.Byte*) - neznaménkový typ s rozsahem 0 až 255
- `short` (*System.Int16*) - -32768 až +32767
- `ushort` (*System.UInt16*) 0 až 65535
- `int` (*System.Int32*) - 32 bitový, znaménkový celočíselný typ
- `uint` (*System.UInt32*) - 32 bitový, neznaménkový celočíselný typ
- `long` (*System.Int64*) - 64 bitový, znaménkový celočíselný typ
- `ulong` (*System.UInt64*) - 64 bitový, neznaménkový celočíselný typ
- `char` (*System.Char*) - Tento typ slouží k uložení znaků v Unicode

Typový systém – Celočíselné typy (2)

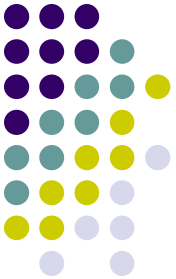


- Celočíselné konstanty lze zapisovat bud v desítkové nebo šestnáctkové soustavě (Ox).
- Neznaménkovou konstantu lze zapsat pomocí u/U.
- Celočíselnou konstantu typu long lze vytvořit pomocí l/L.
- Pokud konstanta překročí rozsah typu `ulong`, jde o chybu při překladu.

```
int a = 10;
uint b = 10u;
short c = 0x0FFF;
long d = 10000L*10000L*10000L
```

- Konstanta je vyčíslena v době překladu.
- Je nutné použít modifikátor L, jinak by došlo k přetečení.

Typový systém – Celočíselné typy (2)



- Konverze celočíselných typů
  - Pokud například používáme `byte` dochází k automatické konverzi na `int`. Musí se „explicitně“ přetypovat.

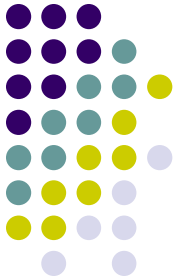
```
byte x = 1, y = 2, sum = 0;
sum = x + y; // nelze (operator +)
sum = (byte)(x + y); // explicitni konverze
```

  - Podobně je tomu i u celočíselných konstant.

```
public void MaMetoda(int x) {}
public void MaMetoda(byte x) {}

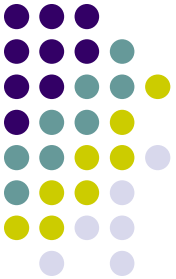
MaMetoda(3); // zavola se metoda s parametrem typu int
MaMetoda((byte)3); // zavola se metoda s parametrem typu byte
```

Typový systém – Celočíselné typy (3)



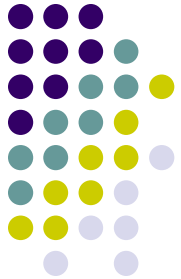
- Speciální je typ `char`
  - Jazyk C# pracuje se znaky v *unicode* o délce 16 bitů.
  - Pro hodnoty typu `char` neexistuje implicitní konverze na žádný celočíselný typ.
  - Znakové konstanty lze zapisovat jako:
    - znak v apostrofech;
      - char a = ‘a’;
    - jednoduchou escape sekvencí;
      - char b = ‘\n’;
    - hexadecimální escape sekvencí;
      - char c = ’\x0041’;
    - unicode escape sekvence.
      - char d = ‘\u0041’

Typový systém – Celočíselné typy (4)



- Escape sekvence v jazyce C#
  - \' Apostrof 0x0027
  - \” Uvozovka 0x0022
  - \\ Opačné lomítko 0x005C
  - \a Alert 0x0007
  - \b Backspace 0x0008
  - \f Form feed 0x000C
  - \n New line 0x000A
  - \t Horizontální tabelátor 0x0009
  - \v Vertikální tabelátor 0x000B

Typový systém – Reálné typy (1)



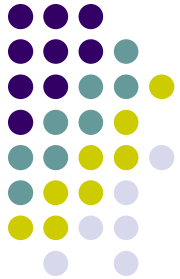
- `float` (*System.Single*) - reálný typ s nejmenším rozsahem přesnosti 1.5e-45 až 3.4e38, 7 desetinných míst
- `double` (*System.Double*) - 64 - bitový reálný typ, 15-16 desetinných míst
- `decimal` (*System.Decimal*) - reálný typ s největší přesností (128 bitový), 28-29 desetinných míst

Typový systém – Reálné typy (2)

- Neexistuje implicitní konverze mezi typem `decimal` a ostatními reálnými typy.
- Reálná konstanta je implicitně typu double.
- Přípony jednoznačně určující typ jsou:
  - f, F - float
  - d, D - double
  - m, M - decimal

```
float a = 1.1f;
double b= -1.1e-1;
decimal c = 1.23456789123456789M;
```

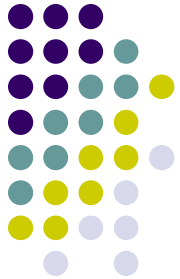
Typový systém – Logický hodnotový typ



- `bool` (*System.Boolean*) - typ pro ukládání logických hodnot `true`, `false`
- Hodnotu logické proměnné lze inicializovat pomocí true (pravda) a false (nepravda).

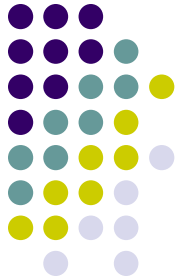
```
bool a = true;
bool b = (a!=a);
```

Typový systém – Struktury (1)



- Uživatelsky definovaný hodnotový typ.
- Někdy také hovoříme o odlehčených třídách.
- Struktura může obsahovat:
  - položky (atributy);
  - metody;
  - operátory;
  - vlastnosti;
  - indexery.

Typový systém – Struktury (2)

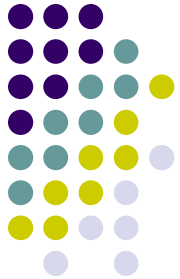


- Od třídy se struktury zásadně liší.
  - Proměnné tohoto typu jsou umístěny na zásobník.
  - Struktury nepodporují dědičnost.
  - Struktura se nemůže stát základem pro vytvoření nového typu.
- K položkám struktury lze přistupovat pomocí tečkového operátoru.

```
struct Point
{
   public int x;
   public int y;
}
```

- Strukturu je nutné před použitím instanciovat pomocí operátoru `new`.

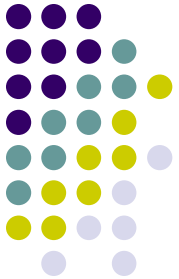
Typový systém – Struktury (3)



```
struct Zvirata_A_Nohy
{
    int pocetNohou;
    string nazevZvirete;

    public Zvirata_A_Nohy(int novyPocetNohou, string
    novyNazevZvirete) {
        pocetNohou = novyPocetNohou;
        nazevZvirete = novyNazevZvirete;
    }
    public int vratPocetNohou() {
        return pocetNohou;
    }
    public string vratNazevZvirete() {
        return nazevZvirete;
    }
}
```

Typový systém – Výčtový typ (1)



- Uživatelsky definovaný hodnotový typ.
- Množina pojmenovaných konstant.
- Výčtové typy jsou deklarovány pomocí klíčového slova `enum`.
- Existuje pro ně řada omezení.
  - Nemohou definovat své vlastní metody.
  - Nemohou implementovat rozhraní.
  - Nemohou definovat své indexery nebo vlastnosti.

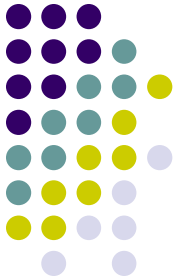
Typový systém – Výčtový typ (2)

```
enum days1{Monday, Tuesday, Wednesday, Thursday,
   Friday, Saturday, Sunday};
enum days2{Monday=1, Tuesday, Wednesday, Thursday,
   Friday, Saturday=0, Sunday=0};
enum days3:byte{Monday, Tuesday, Wednesday,
   Thursday, Friday, Saturday, Sunday};
Console.WriteLine("Today is {0}.",days2.Tuesday);
Console.WriteLine("It is {0}. day in
   week.",(int)(days.Tuesday));
```

Výstup programu bude:

```
Today is Tuesday.
It is 2. day in week.
```

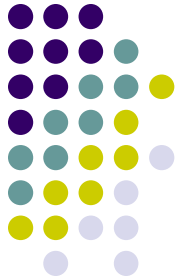
Typový systém – Nullable typy (1)



- Problém u hodnotových typů může být s *null*.
  - Někdy *null* chápeme spíše jako „bez hodnoty“. Tomu například u typu *int* nic neodpovídá.
  - Další problém by mohl být při použití databáze a podobně.
- Řešení, které přináší C# 2.0 nový generický typ *Nullable<T>*
  - Tento typ může uchovávat jak hodnotový a referenční typ.
  - V jazyce přibyl nový operátor `?`, který usnadňuje použití *Nullable*.

```
Nullable<int> a;
int? b;
```

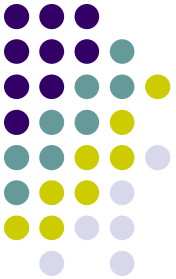
Typový systém – Nullable typy (2)



- Nullable typ je struktura, která kombinuje hodnotu základního typu s booleovským indikátorem null.
  - `bool HasValue` - je true pro instance, které neposkytují hodnotu typu null a false pro null instance.
  - `T Value` - má typ podle základního typu nullable typu. Vrací obsažený typ, nebo vyvolá výjimku.
- Existuje implicitní konverze z jakéhokoliv non-nullable typu na nullable formu tohoto typu.
- Existuje implicitní konverze z *null*, na kterýkoliv nullable typ.

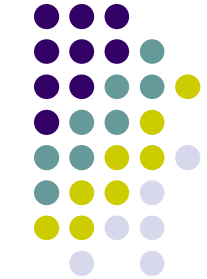
```
int? x = 123;
int? y = null;
if (x.HasValue) Console.WriteLine(x.Value);
double? y = 123;                    // int? --> double?
int? z = (int?)y;                   // double? --> int?
int j = (int)z;              // int? --> int
```

Typový systém – Referenční typy (1)



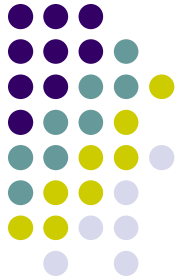
- Uživatelsky definovaný hodnotový typ.
- Množina pojmenovaných konstant.
- Podobné ukazatelům z C ale jsou typově bezpečné.
- Instance referenčního typu je alokována na hromadě.
- Reference může mít hodnotu `null`.
  - I další typy jsou „nullable“, více později.
- Před použitím musí být inicializovaná operátorem `new`.
- O uvolnění paměti se stará Garbage collector.
- Podporují jednoduchou dědičnost a polymorfismus.
- Všechny vycházejí ze třídy `System.Object`.

Typový systém – Referenční typy (2)



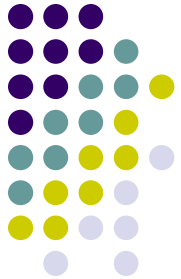
- Jazyk C# definuje tyto skupiny referenčních typů:
  - typ `object` – jde o alias třídy *System.Object*.
  - typ `string` – slouží k uložení textových řetězců. Jde opět o alias k třídě *System.String*.
  - typ třída (`class`);
  - typ rozhraní (`interface`);
  - typ pole;
  - typ delegát (`delegate`).

Typový systém – System.Object



- V CTS .NET Frameworku je vše objekt. Základem každého typu je třída *System.Object*. Tato třída definuje tyto metody:
- `bool Equals()` - tato metoda porovnává referenci dvou objektů;
- `int GetHashCode();`
- `Type GetType()` - poskytuje informace o typu daného objektu;
- `string ToString()` - tato metoda standardně poskytuje název objektu.

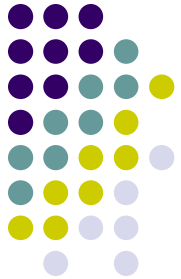
Typový systém – System.String



- Slouží k uložení textových řetězců a manipulaci s nimi.
- Třída *System.String* definuje množství metod a operátorů pro práci s řetězci.
- Při porovnávání se vždy porovnávají hodnoty objektů, tedy samotné řetězce.

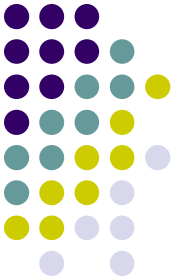
```
string s="Some text";
s="Hello"+" "+"world";
if (s=="Hello world")
   s=Console.ReadLine();
```

Typový systém – Jednoduchý vstup a výstup (1)



- Jednoduchý vstup a výstup programu zprostředkovává třída *System.Console*. Tato třída obsahuje tyto veřejně přístupné vlastnosti:
  - `Error (System.IO.TextWriter)`
  - `Out (System.IO.TextWriter)`
  - `In (System.IO.TextReader)`
- Některé veřejné metody jsou:
  - `int Read()` - čte další znak ze standardního vstupu nebo -1 pro konec vstupu.
  - `string ReadLine()` - čte celý řádek ze standardního vstupu.
  - `Write()` - tato přetížená metoda vypíše hodnotu parametru. To může být text, číslo, ...
  - `WriteLine()` - podobně jako Write vypíše hodnotu parametru následovanou koncem řádku.

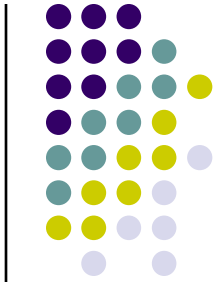
Typový systém – Jednoduchý vstup a výstup (1)



- Metody *Write* a *WriteLine* mohou obsahovat proměnný počet parametrů. Pak je první parametr formátovací řetězec (typu *string*). Znaky {0},{1} budou v tomto formátovacím řetězci nahrazeny hodnotou druhého respektive třetího parametru.

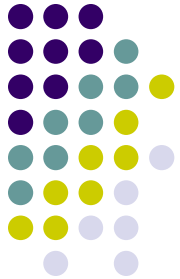
```
Console.Write("Hola ");
Console.Out.Write("Mundo!\n");
Console.WriteLine("What is your name: ");
string name = Console.ReadLine(); //používá
  namespace System
Console.WritLine("Your name is {0}.Is it really
  {0}?",name);
Console.WriteLine("{0}+{1}={2}",1,2,3);
Console.WriteLine("{0}",new System.Object())
```

Direktivy preprocesoru



- I jazyk C# umožňuje využít direktivy preprocesoru
  - #define, #undef, #if, #endif, #elif, #else.
  - #pragma – zakáže či povolí výpis některých varování
  - #region, #endregion – není zpracováván preprocesorem, ale slouží Visula Studiu k definování oblastí. Takto lze zpřehlednit zdrojový kód.

Jmenné prostory (1)



- Knihovna tříd .NET Framework je uspořádána do hierarchické struktury jmenných prostorů.
- Výsledkem může být poměrně dlouhý název.
- Pomocí příkazu *using* lze "připojit" nějaký jmenný prostor (Ne třídu!).

```
using System;
```

- Příkaz using lze využít i pro definování alternativních jmen (aliasů) tříd.

```
using alias = třída;
using output = System.Console;

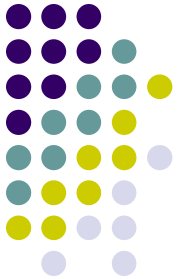
output.Write("Hello world!");
```

Jmenné prostory (2)

```
using SomeNamesapce;
using AnotherNamesapce;

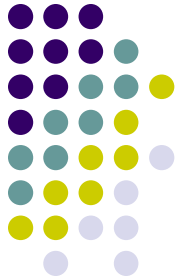
namespace A
{
  class Class1 {}
  class Class2 {}
}
namespace B
{
  namespace C
  {
  }
  class Class1 {}
}
```

Jmenné prostory (3)



- Jmenný prostor **System** obsahuje mimo jiné tyto užitečné typy
  - **Array** - umožňuje práci s poli
  - **Console** - práce s konzolí
  - **Math** - třída pro matematické funkce a operace
  - **Random** - generování náhodných čísel
  - **String** - práce s řetězci
  - **DateTime** - struktura pro práci s datem a časem
- **System.Collections** obsahuje třídy kolekcí a rozhraní pracující s kolekcemi.
- **System.IO** obsahuje třídy pro souborové operace
- **System.Data** obsahuje třídy pro vytváření architektury datového přístupu ADO.NET
- **System.Net** používá síťové operace (služby pro přístup k Internetu apod)
- **System.Xml** slouží pro práci s XML daty
- **System.Threading** pro práci s vlákny
- **System.Security** obsahuje třídy zajišťující bezpečnost.

Třídy

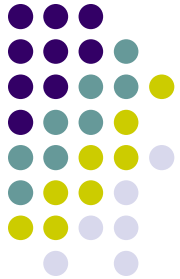


- Jde o jeden z referenčních typů.
- Třída obsahuje zapouzdření dat a metod.
- Deklarace třídy:

```
class SomeName
{
   ...
}
```

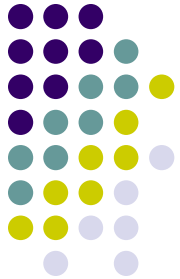
- Jazyk C# neodděluje deklaraci a definici třídy.

Třídy – Členové třídy



- Položky (field) - členské proměnné, udržují stav objektu.
- Metody - funkce implementující služby objektem poskytované.
- Vlastnosti (property) - je také označována za chytrou položku. Navenek vypadají jako položky, ale umí kontrolovat přístup k jednotlivým datům.
- Indexer - u některých tříd je výhodné definovat operátor *[]*. Indexer je speciální metoda, která umožňuje aby se daný objekt choval jako pole.
- Operátory – v jazyce C# máme možnost definovat množinu operátorů sloužících pro manipulaci s jejími objekty.
- Událost (event) – jejím účelem je upozorňovat na změny, které nastaly např. v položkách tříd.

Třídy – Modifikátory přístupu



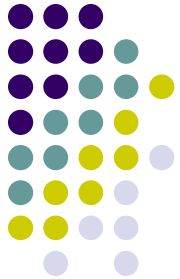
- public – člen označený tímto modifikátorem je dostupné bez omezení;
- private – člen je přístupny pouze členům stejné třídy;
- protected – přistupovat k takovému členu můžeme uvnitř vlastní třídy a ve všech třídách, pro které je třída základem;
- internal – člen je přístupný všem v rámci jedné assembly.
- Při přiřazování modifikátorů přístupu platí tato pravidla:
  - implicitně jsou položky privátní (private);
  - všechny modifikátory mohou být aplikovány na libovolný člen třídy;

Třídy – Jednoduchý příklad (1)

```
class SomeClass
{
  int someValue;

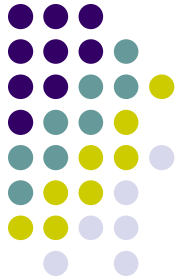
  public int GetSomeValue()
  {
    return someValue;
  }
  public void SetSomeValue(int newValue)
  {
    someValue=newValue;
  }
  public static void Main() {
    System.Console.WriteLine("Do nothing");
  }
}
```

Třídy – Jednoduchý příklad (2)



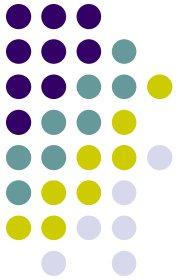
- Implicitně jsou členové třídy deklarovaní jako privátní (private). Položka *someValue* je tedy *private*.
- Modifikátor přístupu musí být explicitně použit u každé položky (jejíž přístupová práva měníme).
- Metoda která nevrací žádnou hodnotu je uvozena klíčovým slovem *void*.
- V případě, že metoda nemá žádné parametry, obsahuje její deklarace pouze prázdné závorky.
- Aplikace je spuštěna pomocí metody Main.

Třídy – Metody Main (1)



- Každá třída může obsahovat pouze jednu metodu Main.
- V projektu může být několik tříd, které obsahují metodu Main. To může být výhodné například pro lazení aplikace.
- Při kompilaci je nutné zvolit jednu z tříd, které obsahují metodu Main (Project - Project properties - General - Statup object)
- Referenční typy je před použitím nutné instanciovat pomocí operátoru new.

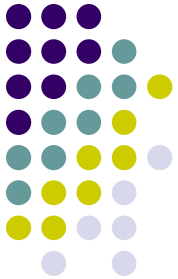
Třídy – Metody Main (2)



```
using System;

class RunApp
{
  public static void Main()
  {
    Console.WriteLine("Running...");
    RunApp run=new RunApp();
  }
}
class DebugApp
{
  public static void Main()
  {
    Console.WriteLine("Debuging...");
  }
}
```

Třídy – Metody Main (3)

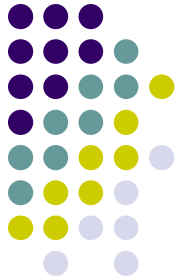


- Parametry z příkazové řádky jsou předávány jako parametr metody Main. Jde o pole textových řetězců (string).

```
using System;
class SomeClass
{
  public static void Main(string args[])
  {
    foreach(string arg in args)
    {
      Console.WriteLine("{0}",arg);
    }
  }
}
```

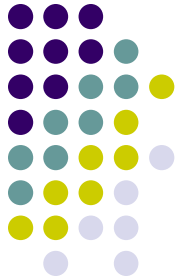
- Metoda Main může také vracet hodnotu typu int (pak je ukončena příkazem return).

Třídy – Statické a nestatické součásti třídy (1)



- Další možný modifikátor členů třídy je *static*.
- Nestatická data jsou individuální pro každý objekt, mohou používat ukazatel *this*.
- Statická data jsou společná pro všechny "uživatele" třídy.
- Ke statickým položkám lze přistupovat pouze přes název třídy, ne přes instanci.
- Statické metody mohou pracovat pouze se statickými položkami.
- Statické položky nemohou používat ukazatel *this*.

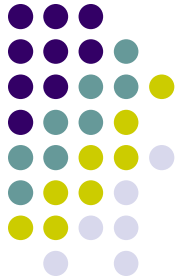
Třídy – Statické a nestatické součásti třídy (2)



```
class SomeClass
{
  static int staticValue=10;
  int instanceValue;

  public static void PrintValue()
  {
    System.Console.WriteLine("{0}",staticValue);
  }
  public void SetInstanceValue(int
    instanceValue)
  {
    this.instaceValue=instanceValue;
  }
}
```

Třídy – Konstruktory (1)



- Konstruktory slouží ke korektní inicializaci objektů.
- Konstruktor je volán v době vytvoření instance příslušného objektu.
  - instance objektů jsou vytvářeny pomocí operátoru *new*.
- Syntaxe konstruktoru je:

```
[public] [static] ClassName([prametr1, ...])
```

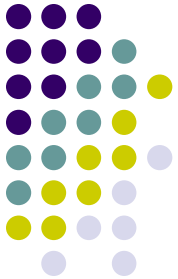
- Konstruktor nemá žádný návratový typ.
- Název konstruktoru je shodný s názvem třídy.
- Jedna třída může mít více konstruktorů lišících se v parametrech (signatuře).
- Konstruktor nesmí mít návratovou hodnotu.

Třídy – Konstruktory (2)

```
using System;

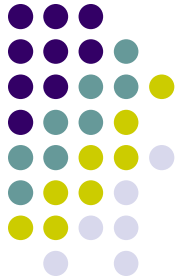
class A
{
  public A(){
    Console.WriteLine("instance of
     class A was created.");
   }
  public A(int a)
   {
    Console.WriteLine("{0}",a);
   }
}
```

Třídy – Konstruktory (3)



- Konstruktor bez parametrů je označován jako implicitní konstruktor.
- Pokud nevytvoříme žádný konstruktor, je tento implicitní konstruktor vygenerován.
- Automaticky vygenerovaný konstruktor má prázdné tělo.
- Pokud vytvoříme nějaký konstruktor, pak automaticky vygenerovaný není.
- Konstruktor může volat další konstruktory a to pomocí ukazatele *this*.

Třídy – Konstruktory (4)

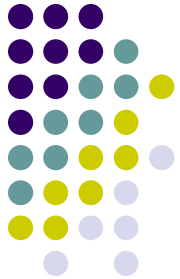


```
class A
{
  public A(int a)
  {
    Console.WriteLine("{0}",a);
  }
  public A(int a,int b):this(a)
  {
    Console.WriteLine("{0}",b);
  }
  public static void Main() {
    A example1=new A(1,2);
    A example2=new A();
  }
}
```

Výstup programu bude:

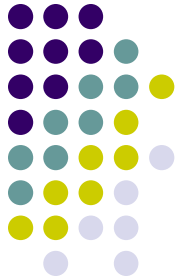
```
1
2
```

Třídy – Statické konstruktory (1)



- **Jazyk C# povoluje statické konstruktory.**
  - **Musí být uvozeny klíčovým slovem *static*.**
  - **Mohou pracovat pouze se statickými položkami třídy.**
  - **Nesmí mít žádné parametry.**
  - **Statické konstruktory nelze volat. Jsou automaticky spuštěny před prvním "použitím" třídy**
  - **Je-li v aplikaci více tříd se statickým konstruktorem, pořadí jejich volaní není definováno.**

Třídy – Statické konstruktory (2)



```
using System;

class A
{
  static A()
  {
    Console.Write("A");
  }
}
class B
{
  static B()
  {
    Console.Write("B");
  }
  public static void Main()
  {
    A a;
  }
}
```

Třídy – Statické konstruktory (3)

- Představený příklad nevypíše nic.
- Doplníme-li tělo metody Main takto:

```
a=new A();
```

- Bude výstup programu:

```
A
```

- Doplníme-li tělo metody Main takto:

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

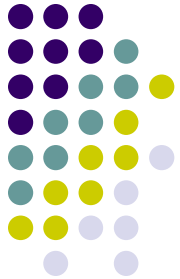
- Výstup programu není exaktně specifikován a bude bud:

```
AB
```

  - nebo

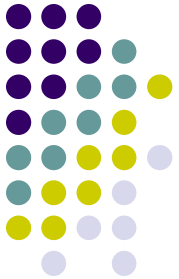
```
BA
```

Třídy – Destruktory



- Pro rušení objektu lze definovat destruktor. Jde o metodu s těmito vlastnostmi:
  - Název je složen z tildy(~) a jména třídy.
  - Nemá návratovou hodnotu a to ani typ void.
  - Nemá žádné formální parametry.
  - Nelze u něj použít žádné modifikátory přístupu.
  - Nemůže být přetížen.
  - Nemusí být definován, pokud to není potřeba.
- ```
  class A {
    ~A()  { //cleaning }
  }
  ```
- Překladač zpracuje destruktor a interně si jej uloží jako metodu *Finalize*. Destruktor volá automaticky *garbage collector*. Není garantováno kdy nebo zda-li vůbec bude zavolán. Pro deterministické uvolňování zdrojů slouží rozhraní *IDisposable*.
  ```

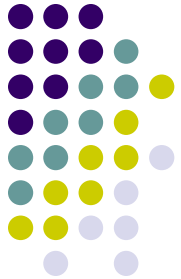
Třídy – Konstanty



- Jde o členská data uvozená klíčovým slovem *const*.
- Jejich hodnota musí být známá v době definice.
- Konstanty jsou interně reprezentovány jako statické položky třídy.
- Konstanty nemohou být v programu modifikovány.

```
class Example
{
  public const double PI=3.14;
  public static void Main()
  {
    System.Console.WriteLine(PI);
  }
}
```

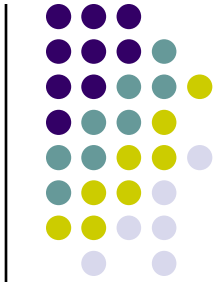
Třídy – Read-only položky



- Jde o členská data uvozená klíčovým slovem *readonly*.
- Jejich hodnota může být naplněna v konstruktoru.
- Žádná jiná metoda nesmí tuto položku modifikovat.
- Read-only položky jsou vázány na konkrétní instanci třídy. Přístup k nim je tedy prostřednictvím konkrétního objektu.

```
class Building {
  public readonly int NumberOfWindows;
  public Building(int NumberOfWindows)  {
    this.NumberOfWindows=NumberOfWindows;
  }
  public static void Main() {
    Building myHome=new Building(10);
    Console.WriteLine("Building has {0}
          windows.",myHome.NumberOfWindows);
  }
}
```

Třídy – Metody

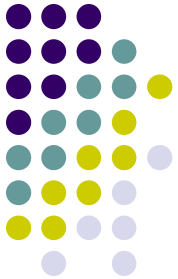


- Metody třídám dodávají funkcionalitu. Metoda má:
  - jméno;
  - návratovou hodnotou. Nevrací li funkce nic, je označena klíčovým slovem *void*;
  - libovolný počet parametrů;
  - viditelnost.

```
class A
{
  public int SomeMethod(int a,string b,object c)
  {
  }
}
```

- Deklarované metody v třídě mohou mít stejný název, musí se ale lišit v signatuře.

Třídy – Parametry metody



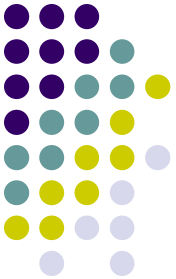
- Standardně jsou parametry metody předávány *hodnotou*.
- Funkce získává kopii skutečných parametrů.
- Jakákoliv změna hodnot parametrů se neprojeví po skončení funkce.

```
class A
{
  public void SomeMethod(int a)
  {
    a=5;
  }
  public static void Main()
  {
    int a=1;
    SomeMethod(a);
    System.Console.WriteLine("{0}",a);
  }
}
```

- Výstupem programu bude:

```
1
```

Třídy – Předávání parametrů odkazem



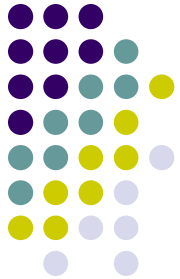
- Volající i metoda sdílí jednu proměnnou, jedno místo v paměti.
- Takové parametry označujeme klíčovým slovem *ref*.
- Předávané parametry musí být inicializovány.

```
class A
{
  public void SomeMethod(ref object a)
  {
    a=new object();
  }
  public static void Main()
  {
    object a,b;
    a=b=new object(); //a musí být inicializovaná
    SomeMethod(ref a);
    System.Console.WriteLine(a==b);
  }
}
```

- Výstupem programu bude:

```
False
```

Třídy – Výstupní parametry



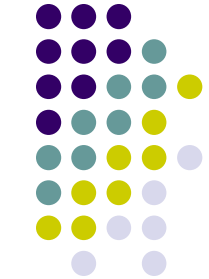
- Volající a metoda opět sdílejí jedno paměťové místo, ale předaný parametr je brán jako neinicializovaný.
- Takovýto parametr označujeme klíčovým slovem *out*.
- Výhodou oproti předávání parametrů pomocí reference je, že proměnné nemusí být inicializovány.

```
class A
{
  public void SomeMethod(out string a)
  {
    a="hello world";
  }
  public static void Main()
  {
    string a; //nemusí být inicializovaná
    SomeMethod(out a);
    System.Console.WriteLine(a);
  }
}
```

- Výstupem programu bude:

```
hello world
```

Třídy – Metody s proměnným počtem parametrů (1)

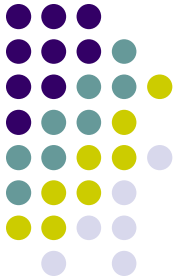


- Příkladem metody s proměnným počtem parametrů je metoda *System.Console.WriteLine*.
- Chceme-li deklarovat metodu s proměnným počtem parametrů použijeme klíčové slovo *params*:

```
NávratovýTyp JménoFunkce(Typ parametr1, ... ,
  params Typ[] jménoParametru) {...}
```

- Položka deklarující proměnný počet parametrů musí být posledním parametrem funkce.
- Typ u položky reprezentující proměnný počet parametrů musí vždy být pole.

Třídy – Metody s proměnným počtem parametrů (2)



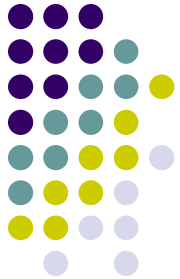
```
using System;

class MySimpleSet
{
  public void PrintIt(params object[] mySet)
  {
    foreach(object item in mySet)
Console.Write(item+",");
    Console.WriteLine();
  }
  public static void Main() {
    PrintIt(1,1.2,"hello",new object());
    PrintIt("one","two");
  }
}
```

- **Výstupem programu bude:**

```
1,1.2,"hello",System.Object,
  "one","two"
```

Třídy – Přetěžování metod

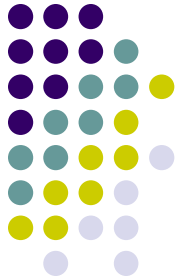


- Jazyk C# umožňuje definovat v jedné třídě více metod stejného názvu.
- Jednotlivé metody se musí lišit v počtu parametrů, typech parametrů nebo v počtu i typech parametrů.
- Pokud se metody liší pouze v návratovém typu, jde o chybu.

```
using System;

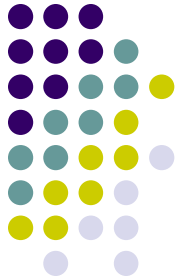
class Example
{
  public void MyPrint(string text)
  {
    Console.WriteLine("Text: "+text);
  }
  public void MyPrint(int number)
  {
    Console.WriteLine("Number: "+number);
  }
}
```

Třídy – Dědičnost



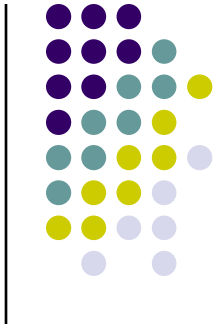
- Vytváříme nové typy na základě existujících - *dědičnost*.
- Odvozená třída obsahuje všechny položky třídy základní.
- C# podporuje pouze jednoduchou dědičnost. Třída tedy může rozšiřovat maximálně jednu třídu.
- V odvozené třídě máme k dispozici všechny veřejné (public) a chráněné (protected) členy třídy základní.
- Dědičnost vyjádříme dvojtečkou uvedenou za jménem třídy a názvem třídy základní.

Třídy – Dědičnost (2)



```
class Point
{
  protected int x,y;
  public void Count() {
  }
}
class Circle : Point
{
  public void SomeMethod()
  {
    x=10;
    Count()
  }
}
```

Třídy – Konstruktory v odvozených třídách (1)



- Při vytváření objektu jsou postupně volány všechny konstruktory a to od třídy základní až po vytvářenou třídu.

```
class A
{
  public A()
  {
    Console.WriteLine("Creating A");
  }
}
class B : A
{
  public B()
  {
    Console.WriteLine("Creating B");
  }
  public static void Main()
  {
    B b=new B();
  }
}
```

Třídy – Konstruktory v odvozených třídách (1)

- Výstup programu bude:

```
Creating A
Creating B
```

- Pokud bychom modifikovali konstruktor třídy A takto:

```
public A(string text)
{
  Console.WriteLine("Creating A, "+text);
}
```

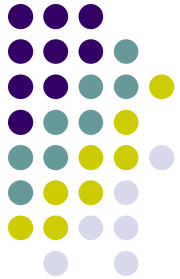
- Překlad nebude úspěšný. Implicitní konstruktor nebyl vygenerován. To, který konstruktor rodičovské třídy bude volán, můžeme ovlivnit pomocí klíčového slova *base*.
- Konstruktor třídy B tedy upravíme:

```
public B() : base ("Hello from B")
{
  Console.WriteLine("Creating B");
}
```

- Výstup programu bude:

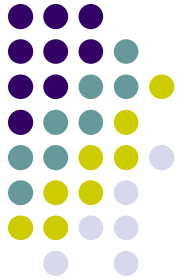
```
Creating A, Hello from B
Creating B
```

Třídy – Překrytí členů třídy (1)



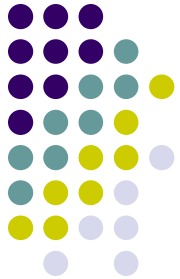
- Chceme-li v odvozené třídě změnit metodu(nebo jiného člena) třídy základní, označíme takovéhoto člena klíčovým slovem *new*.
- Klíčové slovo *new* musí být uvedeno před deklaraci typu metody.
- Překlad bude úspěšný i pokud klíčové slovo new neuvedeme. Výsledkem bude warning a "správné" chování.

Třídy – Překrytí členů třídy (2)



```
using System;
class A
{
  protected int x=10;
  public void Test()
  {
    Console.WriteLine("Test A");
  }
}
class B : A
{
  protected int x=11;
  //warning, správně: protected new int x=11;
  public new void Test()
  {
    Console.WriteLine("Test B");
  }
}
```

Třídy – Překrytí členů třídy (3)



- To, která metoda případně položka třídy bude použita, je nyní závislé na typu reference.

```
class RunApp
{
  public static void Main()
  {
    B b=new B();
    A a=b;
    b.Test();
    a.Test();
  }
}
```

- Výstup programu bude:

```
Test B
Test A
```

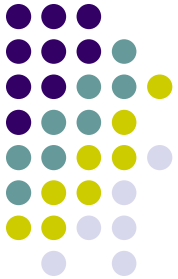
Třídy – Polymorfismus (1)

- Chceme, aby se volala metoda podle typu objektu a ne reference na něj (polymorfní chování).
- Použijeme následující postup:
- Metoda, která se má chovat polymorfně, je u základní třídy označena klíčovým slovem *virtual*.
- Předefinovanou metodu ve všech odvozených třídách označíme klíčovým slovem *override*.
- Opět platí, že tato klíčová slova je nutné uvést před deklarací návratového typu.

Třídy – Polymorfismus (2)

```
using System;
class A
{
  public virtual void Test()
  {
    Console.WriteLine("Test A");
  }
}
class B : A
{
  public new void Test()
  {
    Console.WriteLine("Test B");
  }
}
class C : A
{
  public override void Test()
  {
    Console.WriteLine("Test C");
  }
}
```

Třídy – Polymorfismus (3)

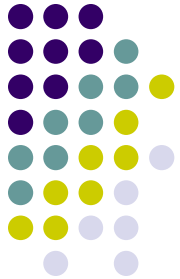


```
class RunApp
{
  public static void Main()
  {
    A a=new A();
    A b=new B();
    A c=new C();
    a.Test();
    b.Test();
    c.Test();
  }
}
```

- Výstup programu bude:

```
Test A
Test A
Test C
```

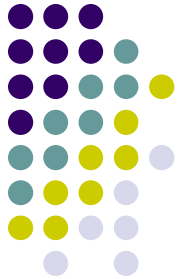
Třídy – Vlastnosti (1)



- Vlastnosti se navenek chovají jako veřejné členské položky, interně jsou však tvořeny množinou *přístupových* metod.

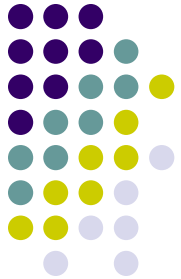
```
class SomeClass
{
  protected string propertyName;
  public string PropertyName
  {
    get
    {
      Console.WriteLine("get "+this.propertyName);
      return this.propertyName;
    }
    set
    {
      this.propertyName=value;
      Console.WriteLine("set "+value);
    }
  }
}
```

Třídy – Vlastnosti (2)



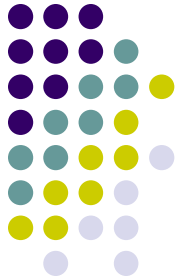
- Vynecháním metody set lze vytvořit read-only vlastnost. Pokusíme-li se do takovéto položky přiřadit, překladač nahlásí chybu.
- Vynecháním metody *get* lze vytvořit write-only vlastnost.
- Vlastnosti nemusí být interně svázány s konkrétní členskou položkou.
- Vlastnosti mohou být statické.
- Vlastnosti jsou interně realizovány jako metody, lze na ně aplikovat modifikátory jako u metod (*virtual*, *override*).

Třídy – Vlastnosti (3)



```
class A
{
  protected static string propertyName;
  public virtual string PropertyName
  {
    get
    {
      return propertyName;
    }
    set
    {
      propertyName=value;
    }
  }
}
class B : A
{
  public override string PropertyName
  {
    get
    {
      return propertyName+", hello from B";
    }
  }
}
```

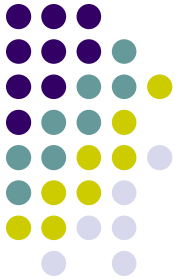
Třídy – Vlastnosti (4)



- Od verze jazyka 2.0 mohou mít přístupové metody *get* a *set* nastaven modifikátor viditelnosti (public, private a nebo protected).

```
public string Name
{
  get
  {
    return name;
  }
  protected set
  {
    name = value;
  }
 }
```

Třídy – Indexer (1)

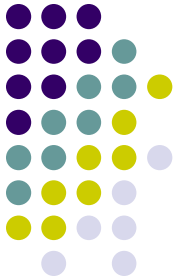


- Umožňuje pracovat s typem jako s polem.
- Obecná syntaxe indexeru používá klíčové slovo *this* s hranatými závorkami.

```
class SetOfPoints {
  Point[] points;
  public SetOfPoints(int size)
  {
    points=new Point[size];
  }

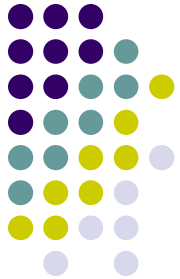
  public Point this[int index]
  {
    set
    {
      if (index<points.Length) points[index]=value;
    }
    get
    {
      if (index<=points.Length) return points[index];
      else return new Point(0,0);
    }
  }
}
```

Třídy – Indexer (2)



```
class RunApp
{
  public static void Main()
  {
    SetOfPoints setOfPoints=new
       SetOfPoints(2);
    setOfPoints[3]=new Point(1,1);
    //pokud by indexer nefungoval, skončí
       výjimkou IndexOutOfRangeException
    Point p=setOfPoints[0]; //bude null...
  }
}
```

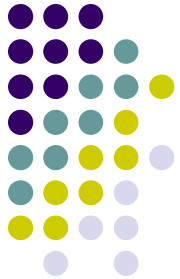
Třídy – Modifikátor sealed (1)



- Další modifikátor použitelný u metod, tříd nebo vlastností (také událostí).
- Je-li použit u třídy, definuje, že tato tříd již nemůže být rozšiřována pomocí dědičnosti.

```
public sealed class A
{
}
public class B : A
  //chyba při překladu
{
}
```

Třídy – Modifikátor sealed (2)



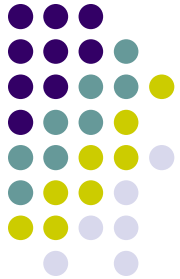
- Je-li použit u metody, daná metoda již nemůže být dále předefinovaná.
- Metoda označená jako sealed musí být virtuální a předefinovaná.
- Modifikátor sealed pomáhá překladači při optimalizaci kódu.

```
class A
{
  public virtual void Test() {}
}
class B : A
{
  public override sealed void Test() {}
}
class C : B
{
  public new void Test() {}
}
```

Třídy – Modifikátor abstract (1)

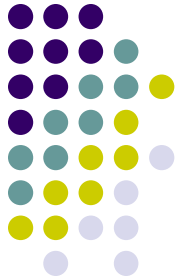
- Další modifikátor použitelný u metod, tříd a vlastností (také událostí).
- Modifikátor abstract říká, že tato položka má být v odvozené třídě předefinovaná.
- Modifikátor abstract lze aplikovat na celou třídu.
- Je-li modifikátor aplikován na metodu, nemá tato metoda tělo.
- Je-li metoda abstraktní je automaticky i virtuální.
- Obsahuje-li třída alespoň jednu abstraktní metodu, je považována za abstraktní (musí mít modifikátor abstract).
- Nelze vytvářet instance abstraktních tříd.
- Vytvářet reference abstraktního typu je povoleno.
- Nelze zároveň použít modifikátor abstract a sealed nebo private.

Třídy – Modifikátor abstract (2)



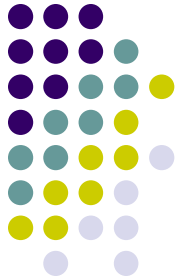
```
abstract class A
{
  abstract void Test();
}
class B : A
{
  public override void Test() {}
}
class C : A
{
  public new void Test() //chyba!
  {}
}
class RunApp
{
  public static void Main()
  {
    A a=new A();
    //chyba! Abstraktní třídu nelze instanciovat.
    A b=new B();  // OK.
  }
}
```

Třídy – Statické třídy



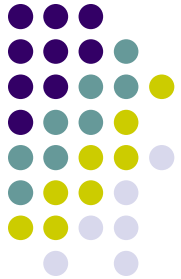
- Statické třídy jsou třídy, u nichž neexistují žádné členské metody nebo data.
  - Možnost definovat statické třídy se objevila až e verzi 2.0.
- Pokud třídu deklarujeme klíčovým slovem `static`, překladač si vynutí dodržení pravidel:
  - zabrání založení instancí a dědičnosti;
  - zabrání použití členských metod a dat.

Neúplné typy (1)



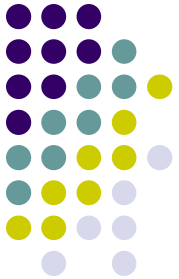
- **Neúplné typy** umožňují programátorovi rozdělit implementaci do několika samostatných souborů.
- Tato vlastnost byla přidána ve verzi 2.0.
- Přesněji platí pro zdrojový kód jedné třídy, struktury a rozhraní.
- Neúplné typy jsou deklarovány pomocí typového modifikátoru `partial` a musí být umístěny ve stejném jmenném prostoru jako ostatní rozdělené části.
- Výhody využití neúplných typů:
  - několik vývojářů může ve stejnou chvíli pracovat nad zdrojovým kódem jednoho datového typu
  - uživatelský kód může být oddělený od generovaného kódu
  - možnost rozdělení větších implementací
  - možné ulehčení údržby a řízení změn ve zdrojovém kódu
  - umožňuje psaní kódu code-beside narozdíl od code-behind, což je využito u tvorby webových aplikací v ASP.NET 2.0

Neúplné typy (2)



```
public partial class Customer {
    private int id;
    private string name;
    private string address;
    private List<Order> orders;
    public Customer() { ... }
}
public partial class Customer {
    public void SubmitOrder(Order order) {
          orders.Add(order);
    }
    public bool HasOutstandingOrders() {
          return orders.Count > 0;
    }
}
```

Pole (1)



- Všechny pole vycházejí z třídy *System.Array*.
- Deklarace pole:

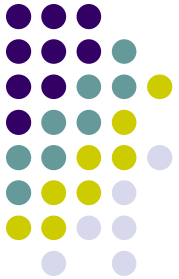
```
type[] arrayName;
```

- Před použitím musí být pole alokováno:

```
arrayName = new type[size];
```

- Při adresování elementu mimo rozsah pole je vygenerovaná výjimka *IndexOutOfRange*.
- Existují dva typy vícerozměrných polí:
  - pravidelná;
  - nepravidelná.

Pole (2)



- Pravidelná vícerozměrná pole

```
type[,,,....] name = new type[number,number,...];
```

- Následující příklad ukazuje jak lze deklarovat dvourozměrné pole celých čísel. Velikost pole je 5x10.

```
int[,] someArray = new int[5,10];
```

- Nepravidelná vícerozměrná pole

```
type[][]...[] name;
```

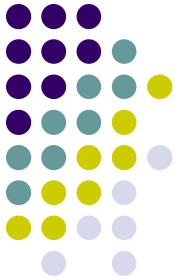
- Pole musí být alokováno postupně.

```
int[][] a=new int[3][];
for (int i=0;i<3;i++)
{
  a[i]=new int[i];
}
```

Pole (3)

- Někteří členové třídy *System.Array*:
  - vlastnost **int Rank** - vrátí dimensi pole;
  - vlastnost **int Length** - vrátí velikost pole;
  - metoda **int GetLength(int rozměr)** - vrátí velikost pole v konkrétní dimensi (počítáno od 0).
  - metoda CopyTo(Array array, int index)
- Některé statické položky třídy *System.Array*:
  - CreateInstance()
  - Sort()
  - Reverse()
  - IndexOf()
  - LastIndexOf()
  - ForEach()
  - Clear()
  - Find()
  - …

Pole (4)



```
int[,]  a = new int[5,10];
int[][] b = new int[2][];
b[0]=new int[5];
b[1]=new int[3];

Console.WriteLine("Length: {0}, Rank: {1}, Lenngth in 1
dimension: {2}",
  a.Length,a.Rank,a.GetLength(1));

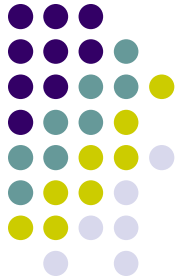
Console.WriteLine("Length: {0}, Rank:
{1}",b.Length,b.Rank);

Console.WriteLine("Length: {0}, Rank:
{1}",b[1].Length,b[1].Rank);
```

- **Výstup programu bude:**

```
Length: 50, Rank: 2, Length in 1 dimension: 10
Length: 2, Rank: 1
Length: 3, Rank: 1
```

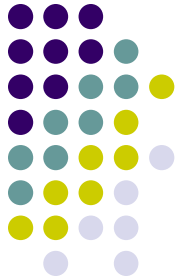
Pole (5)



- Pole lze přímo inicializovat a to uvedením výčtu jeho prvků ve složených závorkách. Toto lze použít i pro vícerozměrná pole.

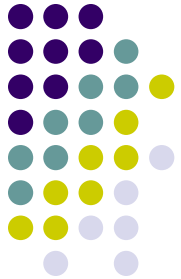
```
int[] a = new int[]{1,2,3};
int[,] b = new int[,]{{1,2,3},{4,5,6}};
int[][] c = new int[][]
    {new int[]{1,2,3},
     new int[]{1,2}}
```

Rozhraní (1)



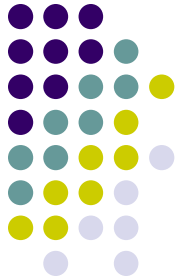
- Jazyk C# podporuje pouze jednoduchou dědičnost. Právě rozhraní dává možnost přidat třídám nějaké charakteristiky nebo schopnosti nezávisle na hierarchii tříd.
- Rozhraní může obsahovat:
  - metody;
  - události;
  - vlastnosti;
  - indexery.
- Definice rozhraní začíná klíčovým slovem *interface*.
- V definic rozhraní jsou uvedeny pouze podpisy členů (u metody je to například pouze deklarace bez těla).
- Všechny členy jsou automaticky veřejné (public).

Rozhraní (2)



```
public interface IExample
{
  void SimpleMethod();
  int WriteOnlyProperty { set; }
  int ReadOnlyProperty { get; }
  int CommonProperty { get; set;}
  object this[int index] { get; set; }
}
class Element : IExample
{
  public void SimpleMethod()
  {
    ...
  }
  ...
}
```

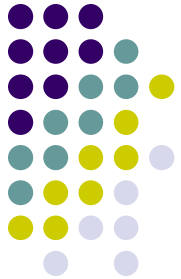
Rozhraní (3)



- V C# může třída implementovat několik rozhraní. Může nastat kolize jmen.
- Můžeme "kvalifikovat" členské jméno jménem rozhraní.
- Takovýto člen pak bude přístupný pouze pomocí daného rozhraní.
- Nesmí mít modifikátor public a není přímo přístupný přes instanci dané třídy.

```
public interface IA
{
  void Test();
}
public interface IB
{
  void Test();
}
```

Rozhraní (3)



```
public class SomeClass : IA, IB
{
  void IA.Test() {}
  void IB.Test() {}

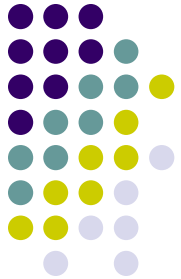
  public static void Main()
  {
    SomeClass test = new SomeClass();
    test.Test(); //error
    ((IA)test).Test()
    IB b=new SomeClass();
    b.Test();
  }
}
```

Rozhraní – Kombinace rozhraní (1)

- Rozhraní lze rozšiřovat pomocí dědičnosti.
- U rozhraní lze používat vícenásobnou dědičnost. Jde vlastně o kombinování několika rozhraní.

```
public interface IA
{
  void Test();
}
public interface IB
{
  void Test();
}
public interface IC : IA, IB
{
  new void Test();
  void AnotherMethod();
}
```

Rozhraní – Kombinace rozhraní (2)



- Při implementaci takovéhoto kombinovaného rozhraní můžeme opět použít kvalifikaci jednotlivých členů. Můžeme bud:
  - Implementovat jednu veřejnou metodu Test.

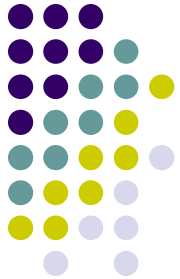
```
class SomeClass : IA, IB, IC
{
  public void Test() {}
  public void AnotherMethod() {}
}
```

  - Implementovat tři kvalifikované metody. Pro každé rozhraní jednu.

```
class SomeClass : IA, IB, IC
{
  void IA.Test {}
  void IB.Test {}
  void IC.Test {}
  public void AnotherMethod() {}
}
```

  - Implementovat kombinaci obou přístupů.

Operátory

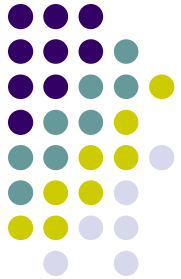


- Primární - (x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
- Unární - + - ! ~ ++x --x (T)x
- Multiplikativní - * / %
- Aditivní - + -
- Posuvu - << >>
- Relační - < > <= >= is as
- Rovnosti - == !=
- Logické bitové - & | ^
- Logické spojky - && ||
- Podmínečné - ?:
- Přiřazovací - = += -= *= /= %= &= |= ^= <<= >>=

Operátory – Primární operátory

- (x) - závorky, implicitně určují prioritu.
- x.y - operátor tečka realizuje přístup k členům tříd nebo struktur. *x* identifikuje element ke kterému přistupujeme a *y* jeho člena.
- f(x) - operátor kulatých závorek obsahuje seznam parametrů pro vyvolání metody.
- a[x] - operátor hranatých závorek slouží k indexování pole.
- x++ a x-- - inkrementace respektive dekrementace.
- new - slouží k vytvoření instance referenčního typu.
- checked a unchecked - tyto operátory zapínají respektive vypínají kontrolu výsledků matematických operací.
- typeof - slouží k *reflexi*. Vrátí instanci třídy System.Type.
- sizeof - slouží k zjištění velikosti hodnotových typů. Tato operace je považována za nebezpečnou a proto musí být tento operátor umístěn v bloku *unsafe*.

Operátory – Primární operátory



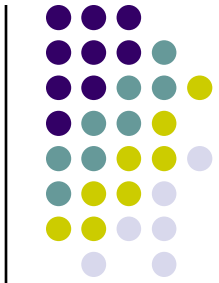
- Definování bloku *unsafe* se explicitně zříkáme bezpečnostních kontrol prostředí .NET
- Blok je definován pomocí klíčového slova *unsafe* a složenými závorkami.
- Je nutné překladači povolit *unsafe* bloky (Project-Properties-Configuration Properties-Build-Allow Unsafe Code Blocks).

```
unsafe
{
  Console.WriteLine("Size of int:
        {0}",sizeof(int));
}
```

Operátory – Unární operátory

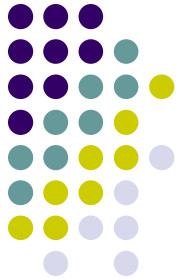
- + a - - operátory unárního plus a unárního mínus. Jsou definovány pro většinu číselných typů.
- ! - operátor logické negace. Je definován pro typ bool.
- ~ - bitový doplněk. Je definován pro typy int, uint, long, ulong.
- ++x, --x - inkrementace a dekrementace (v prefixovém tvaru)
- (T)x - operátor přetypování. V případě chyby je vygenerovaná výjimka *InvalidCastException*.

Operátory – Aritmetické operátory



- Jazyk C# obsahuje běžné aritmetické operátory: +, -, *, /, %.
- Tyto operátory lze kombinovat s operátorem přiřazení: +=, -=, ...
- Výsledkem může být:
  - nějaké číslo;
  - záporná nula;
  - kladné nebo záporné nekonečno;
  - NaN (Not a Number).

Operátory – Další operátory (1)



- Operátory posuvu - <<,>> slouží k realizaci bitového posuvu na celých číslech.
  - Aritmetický posun - při posuvu doprava je na pozici nejvyššího bitu doplňována 0 pro kladná a 1 pro záporná čísla.
  - Logický posun - při posuvu doleva jsou z leva doplněny nuly

```
int b=1<<8;
Console.WriteLine(b);
```

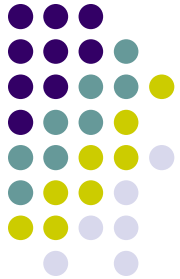
  - Výstup programu bude:

```
256
```

- Relační operátory - ==, !=, <, >, <=, >=. Při porovnávání stringů je porovnávána hodnota. Pokud chceme porovnat reference, musíme alespoň jeden řetězec přetypovat.

```
string a="hello";
string b="hello";
string c=String.Copy(a);
Console.WriteLine(a==c); //True
Console.WriteLine((object)a==(object)b); //True,same constant.
Console.WriteLine((object)a==(object)c); //Flase
```

Operátory – Další operátory (2)

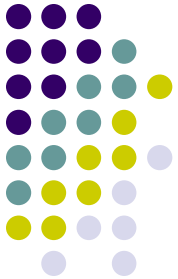


- Logické bitové operátory - AND &, OR |, XOR ^.

```
byte a=5;  //0000 0101
byte b=10; //0000 1010
byte c=a|B;//0000 1111 = 15
```

- Operátor přiřazení - je zprava asociativní.
  - U hodnotových typů jsou přímo kopírovány hodnoty.
  - Referenční typy pouze zkopíruji hodnotu reference. Chceme-li vytvořit opravdovou kopii referenčního typu musíme tuto operaci implementovat sami.
  - Standardní cesta je pomocí rozhraní ICloneable. To má k dispozici metodu Clone.

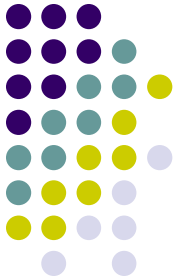
Operátory – Operátor is



- Pomocí operátoru *is* jsme schopni zjistit zda lze daný objekt přetypovat na donou třídu nebo zda-li implementuje konkrétní rozhraní.
  - syntaxe: `expression is type`
  - Výsledkem je hodnota typu bool.

```
class A {}
interface IA {}
interface IB : IA {}
class B :A , IB {}
class RunApp
{
  public static void Main()
  {
     B b=new B();
     Console.WriteLine(b is IA); //True
     Console.WriteLine(b is A);  //True
  }
}
```

Operátory – Operátor as



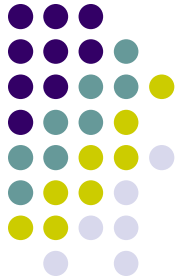
- Operátor as provádí přetypování na daný typ.
- Pokud přetypování není možné, vrací *null*.
- Syntaxe: `object = expression as type`
- Implementace operátoru as pomocí is by vypadala takto:

```
if (expression is type) object=(type)(expression);
else object=null;
```

- Příklad:

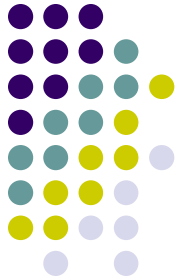
```
class A {}
class B : A {}
class RunApp
{
  public static void Main()
  {
    B b = new B();
    A a = b as A;
  }
}
```

Operátory – Přetěžování operátorů (1)



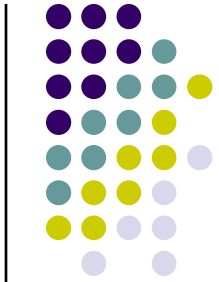
- Jazyk C# umožňuje přetěžování operátorů.
- Syntaxe:
  - `public static returnType operator op (object1 [,object2]) {...}`
  - Definice operátoru je vždy veřejná a statická.
  - Návratový typ (`returnType`) může být libovolný typ.
  - Klíčové slovo *operator* určuje že jde o přetížení operátoru *op*.
- Počet předávaných parametrů (object1, object2) závisí na typu přetěžovaného operátoru. Pro unární operátor bude pouze jeden parametr. Pro binární budou dva.
- První parametr by měl být typu, jehož operátor přetěžujeme.

Operátory – Přetěžování operátorů (2)



```
class Test
{
  protected string text;
  public Test(string text)
  {
    this.text=text;
  }
  public string Text
  {
    get
    {
      return this.text;
    }
    set
    {
      this.text=value;
    }
  }
  public static Test operator + (Test t1,Test t2)
  {
    return new Test(t1.Text+","+t2.Text);
  }
}
```

Operátory – Přetěžování operátorů (3)

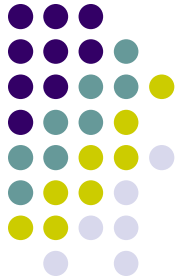


```
class RunApp
{
  public static void Main()
  {
    Test a=new Test("A");
    Test b=new Test("B");
    Test c=a+b;
    Console.WriteLine(c.Text);
    Console.WriteLine((a+b).Text);
    a+=b;
    Console.WriteLine(a.Text);
  }
}
```

- **Výstup programu bude:**

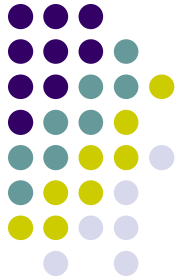
```
A,B
A,B
A,B
```

Operátory – Přetěžování operátorů (1)



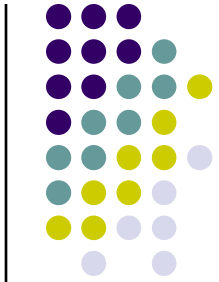
- Přetížitelné operátory jsou:
  - unární: +, -, !, ~, ++, --, true, false;
  - binární: +, -, *, /, %, |, ^, <<, >>, ==, !=, >, <, >=, <=.
- Nelze přetěžovat operátor [], ale lze vytvářet indexery.
- Operátor přiřazení = nelze přetížit (podobný efekt může mít uživatelsky definovaná konverze).
- Při přetížení operátoru + se implicitně přetíží i operátor +=.
- Jazyk C# umožňuje přetížit operátory *true* a *false*.
  - Musí být vždy definovány oba, nebo žádný.
  - Umožňuje, aby se třída nebo struktura chovala jako logická proměnná.

Operátory – Přetěžování operátorů (1)



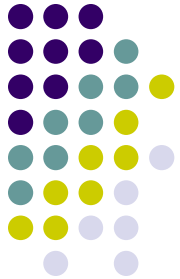
- Přetížitelné operátory jsou:
  - unární: +, -, !, ~, ++, --, true, false;
  - binární: +, -, *, /, %, |, ^, <<, >>, ==, !=, >, <, >=, <=.
- Nelze přetěžovat operátor [], ale lze vytvářet indexery.
- Operátor přiřazení = nelze přetížit (podobný efekt může mít uživatelsky definovaná konverze).
- Při přetížení operátoru + se implicitně přetíží i operátor +=.
- Jazyk C# umožňuje přetížit operátory *true* a *false*.
  - Musí být vždy definovány oba, nebo žádný.
  - Umožňuje, aby se třída nebo struktura chovala jako logická proměnná.

Operátory – Přetěžování operátorů (1)



```
class Test
{
  protected string text;
  public string Text
  {
    get{return this.text;}
    set{this.text=value;}
  }
  public static bool operator true (Test a)
  {
    if (a.Text=="hello") return true; else return false;
  }
  public static bool operator false (Test a)
  {
    if (a.Text!="hello") return true; else return false;
  }
  public static void Main()
  {
    Test a=new Test();
    a.Text="hello";
    if (a) Console.WriteLine("succes");
  }
}
```

Operátory – Uživatelsky definované konverze (1)

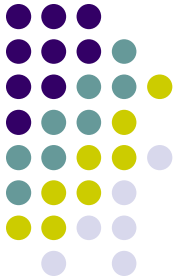


- Slouží ke konverzi třídy nebo struktury do jiné třídy, struktury nebo základního typu.
- Jsou dva druhy uživatelsky definovaných konverzí:
  - implicitní - jsou prováděny aniž by uživatel musel cokoliv zadávat;
  - explicitní - uživatel musí zadat, že chce provést konverzi.
- Uživatelsky definované konverze se deklarují podobně jako při přetěžování operátorů.
- Syntaxe:

```
public static implicit operator type(object)
public static explicit operator type(object)
```

- Typ, pro kterou je konverze definovaná, se musí objevit buď v parametru nebo musí konverze tento typ vracet.
- Uživatelsky definovaná konverze musí být veřejná a statická.

Operátory – Uživatelsky definované konverze (2)



```
class Test
{
  protected string text;
  public string Text
  {
    get
    {
      return this.text;
    }
    set
    {
      this.text=value;
    }
  }
  public static implicit operator Test(string a)
  {
    Test t=new Test();
    t.Text=a;
    return t;
  }
  public static implicit operator string(Test a)
  {
    return a.Text;
  }
}
```

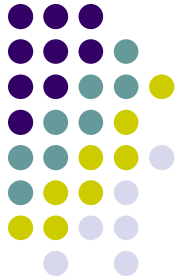
Operátory – Uživatelsky definované konverze (1)

```
class RunApp
{
  public static void Main()
  {
    Test a=new Test();
    a.Text="hello";
    Console.WriteLine(a.Text);
    string b=a; //implicit conversion
    Console.WriteLine(b);
    Test c="world"; //implicit conversion
    Console.WriteLine(c.Text);
  }
}
```

- Výstup programu bude

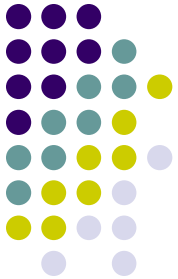
```
hello
hello
world
```

Operátory – Uživatelsky definované konverze (1)



```
class Test
{
  protected string text;
  public string Text
  {
    get
    {return this.text;}
    set
    {this.text=value;}
  }
  public static explicit operator Test(string a)
  {
    Test t=new Test();
    t.Text=a;
    return t;
  }
  public static void Main()
  {
    Test a=(Test)"Hello"; //explicit conversion
    string b=(string)a;   //error
  }
}
```

Operátory – Operátory přidané ve verzi 2.0



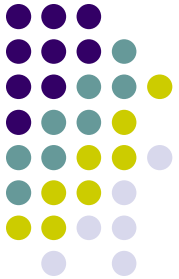
- Operátor ??
  - null splývající operátor (null coalescing operator)
  - Výsledek výrazu A ?? B je A, pokud je A non-nullable typu. Jinak je výsledek B.

    int? z = x ?? y;

    nt i = z ?? -1;
- Operátor ::
  - Tento operátor slouží pro přístup ke globálním jménům.

```
int System, Console;
//this wont work
System.Console.WriteLine("doesnot work!");
//this will work
global::System.Console.WriteLine("works!");
```

Příkazy (1)

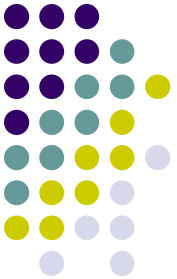


- Příkaz if

```
if (logic expression) statement1 [else statement2]
```

  - Výraz, na jehož základě je realizováno větvení, musí být logická hodnota.
  - Větev else není povinná.
- Příkaz switch
  -
```
switch(expression)
{
  case constant1: [statements1; break;]
  case constant2: [statements2; break;]
  ...
  [default: statementsN; break;]
}
```
  - Testovaný výraz v příkazu switch může být bud: celočíselný typ, char, enum nebo string.
  - Jednotlivé skupiny příkazů musí být ukončeny příkazem break.
  - Varianta default (pokud nebyla nalezena žádná shoda) není povinná.
  - Lze kombinovat s příkazem goto.

Příkazy (2)



```
string b="have a nice day";
switch (b)
{
  case "hello":
  case "good day":
    Console.WriteLine("welcome");
    break;
  case "have a nice day":
    Console.WriteLine("how nice!");
    goto case "bye";
  case "bye":
    Console.WriteLine("bye");
    Break;
  default:
    Console.WriteLine("What?");
    break;
}
```

Příkazy (3)

- Příkaz for

```
for(initialization; logic expression;
  iterators) statement
```

  - Inicializace je sekvence výrazu nebo přiřazení (oddělených čárkou). Provede se na začátku cyklu.
  - Logický výraz se testuje před každým průchodem cyklu. Řídí cyklus.
  - Třetí výraz je vykonán po vyhodnocen vždy po průchodu tělem cyklu.
- Příkaz while

```
while (logic expression) statement
```

- Příkaz do-while

```
do statement while (logic expression)
```

Příkazy (4)

- Příkaz foreach

```
foreach(type identifier in expression)
  statement
```

  - Cyklus speciálně navřený pro průchod prvky pole nebo kolekce.
  - `type` - typ elementu pole nebo kolekce.
  - `identifier` - jméno, toto jméno lze používat v těle cyklu pro zpracovávány prvek pole
  - `expression` - musí obsahovat referenci na pole nebo kolekci.

Příkazy (5)

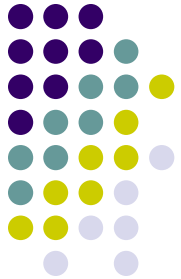
```
string[] array=new string[]{"hello","good day","merry
christmas"};
foreach(string s in array) Console.Write(s+", ");

SortedList list=new SortedList();
list.Add("hello","hello world");
list.Add("good","good day");
list.Add("merry","merry christmas");
foreach(DictionaryEntry element in list)
Console.WriteLine(element.Value);
```

- Výstup programu bude:

```
hello, good day, merry christmas,
hello world
good day
merry christmas
```

Příkazy (6)



- Příkaz break

  ```
  break;
  ```

  - Musí být použit v těle cyklu nebo v příkaz switch.
- Příkaz continue

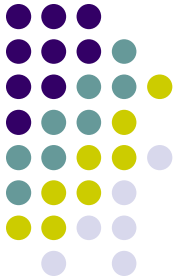
  ```
  continue;
  ```

  - Musí být použit v těle cyklu.
- Příkaz return

  ```
  return [expression];
  ```

  - Příkaz return musí být v každé metodě s návratovou hodnotou různou od void.
  - Dochází-li ve funkci v větvení, pak každá větev musí vracet hodnotu.
  - Hodnotu musí vracet i metoda get v indexerech a vlastnostech.
  - Je li návratuvý typ void, lze použít příkaz return bez argumentu.

Příkazy (7)



- **Příkaz goto**
  - **Příkaz lze kombinovat s příkazem switch.**

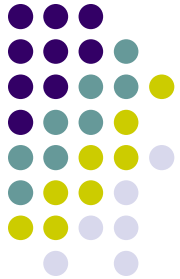
```
goto case constant;
goto default;
```

- **Příkaz skoku na konkrétní návěští.**

```
goto label;
:label
```

```
goto label;
Console.WriteLine("hidden");
//warning, ureachable code
label:
  Console.WriteLine("visible");
```

Výjimky (1)

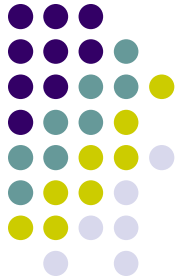


- Chyby, ke kterým dojde pří běhu programu, jsou ošetřovány pomocí výjimek.
- Následující příklad skončí výjimkou.

```
class SomeClass
{
  static void Main(string[] args)
  {
    int[] array = new int[3];
    array[3]=5;
     //throw IndexOutOfRangeException
  }
}
```

- Výjimka je reprezentovaná třídou *System.Exception*, nebo třídou z ní odvozenou.

Výjimky (1)

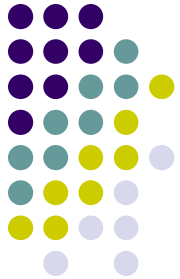


- Chyby, ke kterým dojde pří běhu programu, jsou ošetřovány pomocí výjimek.
- Následující příklad skončí výjimkou.

```
class SomeClass
{
  static void Main(string[] args)
  {
    int[] array = new int[3];
    array[3]=5;
     //throw IndexOutOfRangeException
  }
}
```

- Výjimka je reprezentovaná třídou *System.Exception*, nebo třídou z ní odvozenou.

Výjimky (2)



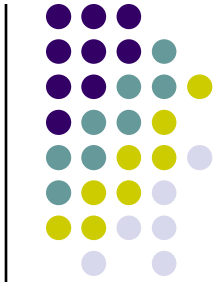
- Výjimku můžeme ošetřit a zpracovat. Syntaxe je:

```
try
{...}
catch (Exception e)
{...}
```

- Předchozí příklad můžeme modifikovat:

```
class SomeClass
{
  static void Main(string[] args)
  {
    int[] array = new int[3];
    try
    {
      array[3]=5; //throw IndexOutOfRangeException
    }
    catch (Exception e)
    {
    }
  }
}
```

Výjimky (3)



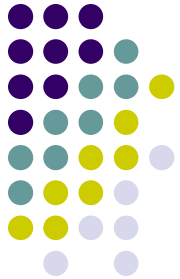
- Někdy můžeme chtít ošetřit víc typů výjimek. Můžeme definovat víc bloků *catch*.

```
try
{...}
catch (IndexOutOfRangeException e)
{...}
catch (Exception e)
{...}
```

- Někdy potřebujeme, aby se kus programu provedl ať už k výjimce došlo nebo ne. K tomu slouží blok *finally*.

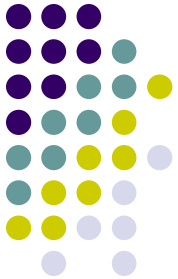
```
try
{
  array[3]=5; //throw IndexOutOfRangeException
}
catch (Exception e){}
finally
{
  Console.WriteLine("Always visible");
}
```

Výjimky – Předdefinované výjimky



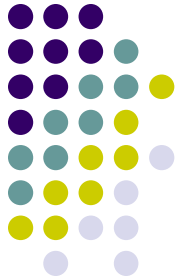
- **Exception** - Základní třída pro všechny výjimky.
- **SystemException** - Základní třída pro všechny druhy výjimek generovaných za běhu aplikace.
- **IndexOutOfRangeException** - Přístup k prvku mimi rozsah pole.
- **NullReferenceException -** Generovaná při pokusu pracovat s neinicializovanou referencí.
- **ArgumentException** - Základní třída pro všechny výjimky způsobené chybnými argumenty.
- **ArgumentNullException** - Parametr s *null* hodnotou nebyl očekáván.
- **InteropException** - Výjimka generovaná mimi CLR .NET Framework.

Výjimky – Vytvoření vlastního typu výjimky



- Všechny typy výjimek musí být odvozeny ze třídy **System.Exception**.
- Třída **Exception** má čtyři konstruktory.
  - `public Exception();`
    - Implicitní konstruktor inicializuje členská data na předdefinované hodnoty.
  - `public Exception(string);`
    - Nastavuje popis výjimky jako textový řetězec.
  - `public Exception(SerializationInfo, StreamingContext);`
    - Inicializuje člena třídy tzv. serializovanými daty.
  - `public Exception(string, Exception);`
    - Dává nám možnost vytvořit událost obsahující data jiné události. Takto můžeme tvořit celou hierarchii do sebe zanořených událostí.

Výjimky – Vytvoření vlastního typu výjimky



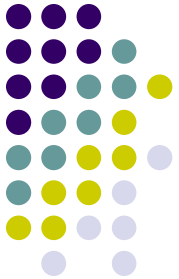
- Výjimky vyhazují některé operace nebo třídy z knihoven .NET Framework.
- Někdy můžeme chtít sami vyhodit výjimku. K tomu slouží příkaz **throw**.

```
throw expression;
```

  - Výraz musí být reference na instanci třídy Exception, nebo třídy z ní odvozené.
- V těle bloku catch můžeme vyvolat původní výjimku. Tedy tu kterou jsme v bloku catch dostali jako argument. Opět použijeme příkaz throw, tentokrát bez argumentu.

```
throw;
```

Výjimky – Příklad

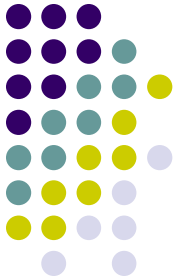


```
using System;
namestace Test
{
  class MyException : Exception
  {
    public MyException(string text,Exception exception) : base
(text,exception) {}
  }
  class RunApp
  {
    public static void Main()
    {
      throw new MyException("My error message",new
Exception("System error message"));
    }
  }
}
```

- Výstup programu bude:

```
Unhandled Exception: Test.MyException: My error message --->
System.Exception: System error message
   --- End of inner exception stack trace ---
   at Test.RunApp.Main(String[] args) in c:\test\RunApp.cs:line 12
Press any key to continue
```

Delegáti

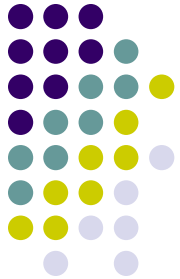


- Delegát je podobný ukazateli na funkci z jazyka C++.
- Oproti ukazatelům na funkce je typově bezpečný.
- Delegáti se používají jako:
  - call-back metody;
  - metody pro událostní programování.
- Delegáta je potřeba deklarovat.

```
modifiers delegate type delegatsName
   (parameter1,...);
```

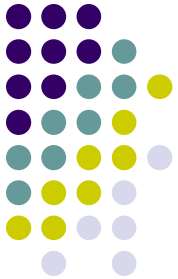
  - Delegát může být veřejný, privátní nebo chráněný.
  - Že jde o deklaraci delegáta určuje klíčové slovo delegate.
  - Jinak jde o běžnou deklaraci metody.
  - Před vlastním použitím je potřeba delegáta instanciovat. Tím ho svážeme s nějakou konkrétní metodou (metodami).

Delegáti – Příklad



```
public delegate void SomePrefix();
class Text
{
  string text;
  public Text(string text)
  {
    this.text=text;
  }
  public void WriteIt(SomePrefix prefix)
  {
    prefix(); //jako běžná metoda
    Console.WriteLine(text);
  }
}
class PrefixBuilder
{
  public static void SimplePrefix()
  {
    Console.Write("## ");
  }
  public void NicePrefix()
  {
    Console.Write(">-how nice-> ");
  }
}
```

Delegáti – Příklad



```
class RunApp
{
  static void Main()
  {
    Text text=new Text("Hello");

    SomePrefix simplePrefix=new
SomePrefix(PrefixBuilder.SimplePrefix);

    PrefixBuilder prefixBuilder=new PrefixBuilder();
    SomePrefix nicePrefix=new
SomePrefix(prefixBuilder.NicePrefix);

    text.WriteIt(simplePrefix);
    text.WriteIt(nicePrefix);
  }
}
```

- Výstup programu bude:

```
## Hello
>-how nice-> Hello
```

Delegáti – Kompozice delegátu

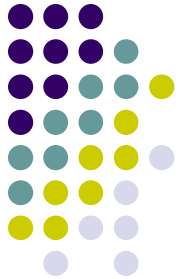
- Delegát nemusí být interně spojen pouze s jedinou metodou.
- Pomocí operátoru + a - jsem schopni delegáty "libovolně" skládat a slučovat.

```
class RunApp
{
  static void Main(string[] args)
  {
    Text text=new Text("Hello");
    SomePrefix simplePrefix=new
SomePrefix(PrefixBuilder.SimplePrefix);
    PrefixBuilder prefixBuilder=new PrefixBuilder();
    SomePrefix nicePrefix=new
SomePrefix(prefixBuilder.NicePrefix);
    SomePrefix greatPrefix=simplePrefix + nicePrefix +
simplePrefix;
    text.WriteIt(greatPrefix);
    greatPrefix-=nicePrefix;
    text.WriteIt(greatPrefix);
  }
}
```

- Výstup programu bude:

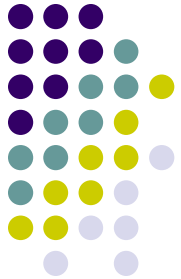
```
## >-how nice-> ## Hello
## ## Hello
```

Delegáti – Anonymní metody (1)



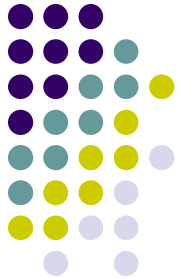
- Ve verzi 2.0 přibyla nová vlastnost *anonymní metody*.
- Anonymní metody umožňují psát kód delegátů přímo "in-line".
- Vytváří se klíčovým slovem **delegate,** za kterým následuje in-line definice anonymní funkce.
- Dvě hlavní vlastnosti anonymních metod:
  - vypuštění seznamu parametrů při definici anonymní funkce, pokud tento seznam nevyužíváme.
    - Seznam typových parametrů je považován za kompatibilní se všemi delegáty kromě těch, kteří mají *out* parametry.
    - Parametry definující typ delegáta je nutné vždy dodat, a to ve chvíli, kdy je delegát volán.
  - využití lokálních proměnných, které jsou definovány v těle metody, v níž je anonymní metoda umístěna.
    - Takovým proměnným a parametrům říkáme **vnější (outer) proměnné.** Navíc při odkazování se na tyto vnější proměnné je nazýváme **zachycené** (captured).

Delegáti – Anonymní metody (2)



- **Seznam parametrů** delegáta je kompatibilní s anonymní metodou pokud:
  - Anonymní metoda nedefinuje seznam parametrů a delegát nemá žádné *out* parametry.
  - Signatura anonymní metody je shodná s delegátem.
- **Návratový typ** delegáta je kompatibilní s anonymní metodou pokud platí alespoň jedno z těchto tvrzení:
  - Návratový typ delegáta je **void** a anonymní metoda neobsahuje žádné příkazy *return* anebo obsahuje pouze *return;*.
  - Návratový typ delegáta není **void** a všechny výrazy v anonymní metodě vázané s příkazem *return* mohou být implicitně přetypovány na návratový typ delegáta.

Delegáti – Anonymní metody (3)

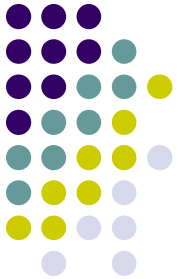


```
delegate double Function(double x);

static double[] Apply(double[] a, Function f)
{
    double[] result = new double[a.Length];
    for (int i = 0; i < a.Length; i++)
    {
        result[i] = f(a[i]);
    }
    return result;
}

double[] a = {0.0, 0.5, 1.0};
a=Apply(a,delegate(double x){return x * x;});
```

Delegáti – Anonymní metody (4)



```
delegate double Function(double x);
static double[] Apply(double[] a, Function f) {
     double[] result = new double[a.Length];
     for (int i = 0; i < a.Length; i++)
            result[i] = f(a[i]);
     return result;
}
```

- Anonymní metoda, která používá vnější proměnné.

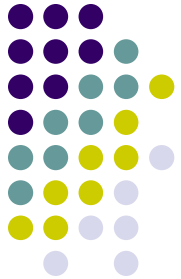
```
static double[] MultiplyAllBy(double[] a, double factor)
{
     return Apply(a, delegate(double x) { return
     x * factor; });
}
```

- Vypuštění parametrů v definici anonymní funkce:

```
Apply(a, delegate{return 0.0;});
```

  - Za klíčovým slovem delegate není ani složená závorka. Ta by definovala, že seznam parametrů je prázdný,

Delegáti – Konverze skupinových metod



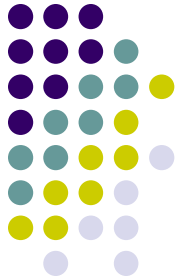
- Anonymní metoda může být implicitně přetypována na kompatibilní delegátský typ.
- C# 2.0 dovoluje stejný způsob konverze i pro **skupinu metod**.
- Dovoluje explicitním konkretizacím delegátů, aby byly ve všech případech vynechány.
- Místo:

```
addButton.Click += new EventHandler(AddClick);
Apply(a, new Function(Math.Sin));
```

- Můžeme použít:

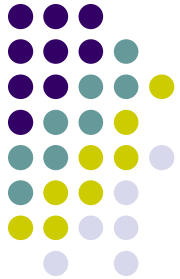
```
addButton.Click += AddClick;
Apply(a, Math.Sin);
```

Delegáti – Události (1)



- Zpracování událostí je v zásadě proces, kdy jeden objekt dokáže upozornit další objekty na to, že došlo k nějaké změně (události). Systém událostí v jazyce C# interně využívá delegátů. Na tyto delegáty jsou kladeny tyto podmínky:
  - delegát musí mít dva parametry a oba jsou objekty;
  - první udává kdo je zdrojem události;
  - druhý obsahuje informace spojené s konkrétní událostí. Tento objekt musí rozšiřovat třídu *EventArgs*.
- Využití delegátu a událostí demonstruje následující příklad.

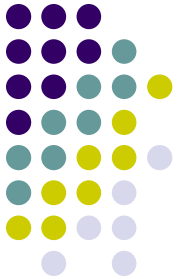
Delegáti – Události (2)



- Pokud událost generuje nějaké argumenty, které potřebují posluchači pro korektní zpracování událostí, může je předat prostřednictvím druhého parametru.
- Ten musí rozšiřovat třídu EventArgs.
- V našem případě třída bude mít navíc text, který obsahuje informace o vygenerované události.

```
class InfoEventArgs : EventArgs
{
    private string info;
    public InfoEventArgs(string info)
    {
        this.info=info;
    }
    public string Info
    {
        get {return info;}
    }
}
```

Delegáti – Události (Zdroj událostí)



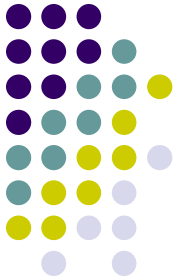
```
class Producer
{
  string name;
  public Producer(string name)
  {
    this.name=name;
  }
  public string Name
  {
    get {return name;}
  }

  public  delegate void WantToKnow(Producer source,InfoEventArgs args);

  public event WantToKnow ItemProduced;

  public void Produce(string productName)
  {
    Console.WriteLine("Production of "+productName+" started.");
    InfoEventArgs info=new InfoEventArgs(productName);
    Console.WriteLine("Production of "+productName+" ended.");
    if (ItemProduced!=null) ItemProduced(this,info); //vyvolání události
  }
}
```

Delegáti – Události (3)



- Předcházející příklad ukazuje jak lze implementovat zdroj událostí.
- Nejprve je definice typ delegátu `WantToKnow`, ten má požadovaný formát.
  - Nevrací žádnou hodnotu, první parametr je zdroj událostí a druhý třída rozšiřující `EventArgs`.
    - Jde o naši předpřipravenou třídu `InfoEventArgs`.
- Následuje definice události. Událost je deklarovaná klíčovým `event`. Následuje typ delegáta, který může být posluchačem události a jméno události.
  - V našem příkladě je událost pojmenovaná `ItemProduced`.
- V případě, že potřebujeme vyvolat událost. Použijeme podobný způsob jako bychom „zavolali“ metodu. Jako název použijeme jméno události, parametry definuje typ delegáta, který je posluchačem události.

Delegáti – Události (Posluchač)

```
class Customer
{
  string name;
  public Customer(string name,Producer producer)
  {
     this.name=name;

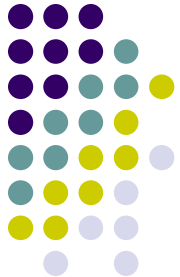
     //Registrace noveho delegata u udalosti
     producer.ItemProduced+=new
             Producer.WantToKnow(NewItemProduced);

  }
  //Obsluha udalosti
  public void NewItemProduced(Producer producer,InfoEventArgs info)
  {

    Console.WriteLine(this.name+": "+producer.Name+" produce
             item:"+info.Info);
  }

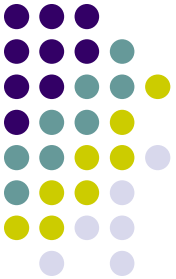
}
```

Delegáti – Události (4)



- Pokud chce nějaký objekt reagovat na generované události, musí se stát „posluchačem“ dané události.
  - To vlastně znamená, že musí vytvořit delegáta příslušného typu (v našem případě `WantToKnow`) a tohoto delegáta zaregistrovat u události.
    - V našem příkladě je vytvořen objekt typu `Producer.WantToKnow` obsahující referenci na metodu `NewItemProduced`.
    - Vytvořený delegát je přidán operátorem += k události producer.ItemProduced (jde o instanční položku objektu producer)

Delegáti – Události (Spuštění aplikace)

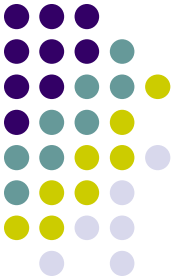


```
class RunApp {
  public static void Main() {
    Producer producer=new Producer("Haven inc.");
    Customer marek=new Customer("Marek",producer);
    Customer tom=new Customer("Tom",producer);
    producer.Produce("Ferrari");
    producer.Produce("pencil");
    producer.Produce("cake");
  }
}
```

- **Výstupem programu bude:**

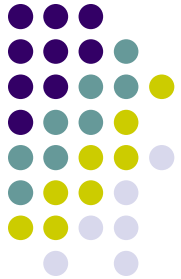
Production of Ferrari started.
Production of Ferrari ended.
Marek: Haven inc. produce item:Ferrari
Tom: Haven inc. produce item:Ferrari
Production of pencil started.
Production of pencil ended.
Marek: Haven inc. produce item:pencil
Tom: Haven inc. produce item:pencil
Production of cake started.
Production of cake ended.
Marek: Haven inc. produce item:cake
Tom: Haven inc. produce item:cake

Delegáti – Události (5)



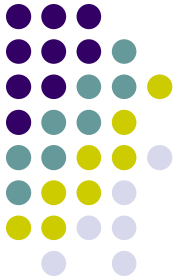
- V hlavní části programu pak nejprve vytvoříme instanci zdroje událostí, objekt `producer` typu `Producer`.
- Pak vytvoříme dva posluchače. Objekty `marek` a `tom` typu `Customer`.
  - Tyto objekty se zaregistrují u zdroje událostí.
- Následně jsou vygenerovány tři události pomocí volání metody `Produce`.

Generické datové typy – Proč používat



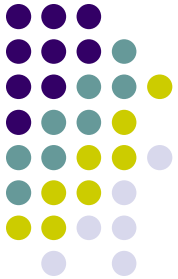
- Například při použití kolekcí dochází obvykle k přetypování na `object`.
  - Sice tak můžeme s daným objektem pracovat v různých kontextech, nutné je ovšem jeho přetypování.
    - Nutná typová kontrola za běhu aplikace.
    - Chyby, které by mohly být odhaleny již při překladu.
    - Zdrojový kód zaplaven typovými konverzemi.
- Řešením tohoto typu problému jsou generické datové typy.

Generické datové typy – Základní vlastnosti



- Mluvíme-li o generickém kódu, máme namysli kód, který využívá **parametrizované typy**.
  - Parametrizované typy jsou nahrazeny konkrétními typy v době použití kódu.
    - hodnotový typ, referenční typ, typový parametr.
  - **<T>** - třídy, struktury, rozhraní, metody a delegáty
  - Na jednotlivé parametrizované typy navíc můžeme klást různá omezení.
  - V rámci jedné definice můžeme použít několik parametrizovaných typů (<A,B,C>).

Generické datové typy – Jednoduchý příklad (1)

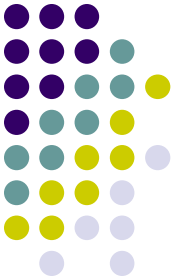


```
public class Stack
{
        object[] items;
        int count;
        public void Push(object item) {...}
        public object Pop() {...}
 }

Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop();

Stack stack2 = new Stack();
stack2.Push(new Customer()); // class for customer
string s = (string)stack2.Pop(); // bad explicit cast, but not error
```

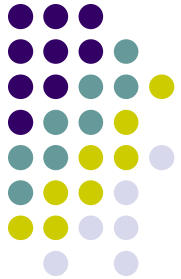
Generické datové typy – Jednoduchý příklad (2)



```
public class Stack<T>
{
    T[] items;
    int count;
    public void Push(T item) {...}
    public T Pop() {...}
 }

Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
string y = stack.Pop(); //type mismatch
```

Generické datové typy – Jednoduchý příklad (3)



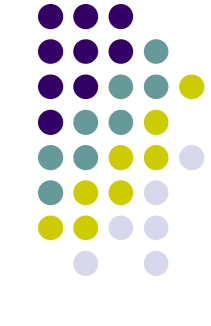
- Generické datové typy mohou mít více typových parametrů.

```
public class Dictionary<K, V>
{
  public void Add(K key, V value) {...}
  public V this[K key] {...}
}
```

  - Použití by pak vypadalo třeba takto:

```
Dictionary<string,Customer> dict = new
  Dictionary<string,Customer>();
dict.Add("Peter", new Customer());
Customer c = dict["Peter"];
```

Generické datové typy – Omezení (1)

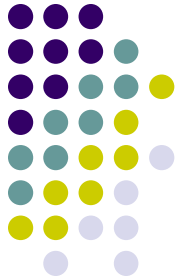


- Někdy potřebujeme na typované parametry klást jistá omezení.

```
public class Dictionary<K, V>
{
   public void Add(K key, V value)
   {
     …
     if (key.CompareTo(x) < 0) {...}
     // Error, no CompareTo method
     …
   }
}
```

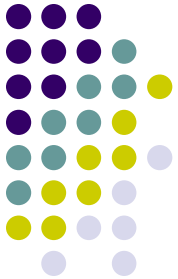
  - Tyto omezení můžeme specifikovat pomocí konstrukce `where`.

Generické datové typy – Omezení (2)



- *where T: struct* - parametrizovaný typ T musí být hodnotový typ.
- *where T : class* - T musí být referenční typ. Může jít o třídu, rozhraní, delegáta a nebo pole.
- *where T : new()* - T musí mít veřejný konstruktor bez paramtrů. Pokud toto omezení používáme v kombinaci s ostatními, musí být poslední.
- *where T : <base class name>* - T musí být odvozen ze specifikované třídy.
- *where T : <interface name>* - T musí implementovat dané rozhraní. Lze specifikovat několik těchto omezení najednou.
- *where T : U* - T musí být odvozen z dodaného parametrizovaného typu pro U.

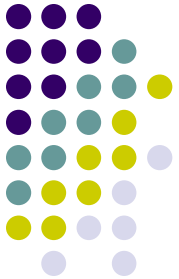
Generické datové typy – Omezení (3)



- Řešením předcházejícího problému může být, že budeme vyžadovat aby typový parametr K implementoval rozhraní `IComparable`.

```
public class Dictionary<K, V>
    where K: IComparable
{
    public void Add(K key, V value)
    {
            ...
            if (key.CompareTo(x) < 0) {...}
            ...
    }
}
```

Generické datové typy – Omezení (4)

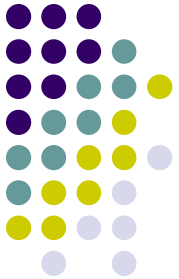


```
class EntityTable<K, E>
   where K: IComparable<K>, IPersistable
   where E: Entity, new()

class EmployeeList<T>
   where T : Employee, IEmployee, System.IComparable<T>,
   new()

public static void OpTest<T>(T s, T t) where T : class
{
   System.Console.WriteLine(s == t);
}
class List<T>
{
   void Add<U>(List<U> items) where U : T {/*...*/}
}
public class SampleClass<T, U, V> where T : V { }
```

Generické datové typy – Generické metody



- Předchozí definici třídy `Stack` bychom mohli rošířit například takto:

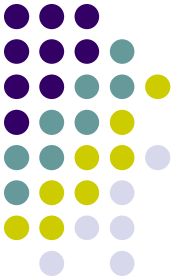
```
void PushMultiple<T>(Stack<T> stack, params T[]
   values)
{
    foreach (T value in values)
    stack.Push(value);
}

Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

  - Můžeme rovněž volat takto:

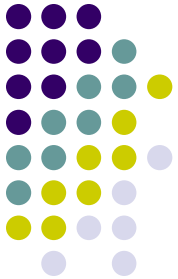
```
PushMultiple(stack, 1, 2, 3, 4);
```

Generické datové typy – Volání generických metod



```
class Test
{
      static void F<T>(int x, T y)
      {
            Console.WriteLine("one");
      }
      static void F<T>(T x, long y)
      {
            Console.WriteLine("two");
      }
      static void Main()
      {
            F<int>(5, 324); // Ok, prints "one"
            F<byte>(5, 324); // Ok, prints "two"
            F(5, 324); // Ok, prints "one"
            F<double>(5, 324); // Error, ambiguous
            F(5, 324L); // Error, ambiguous
      }
}
```

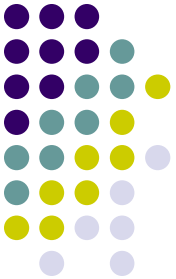
Generické datové typy – Signatura generických metod



- Omezení jsou u signatur generických metod ignorována.
- Významný je počet generických typových parametrů jako i seřazení pozic typových parametrů.

```
class A {}
class B {}
interface IX {
  T F1<T>(T[] a, int i); // Error
  void F1<U>(U[] a, int i); // Error
  void F2<T>(int x); // Ok
  void F2(int x); // Ok
  void F3<T>(T t) where T: A; // Error
  void F3<T>(T t) where T: B; // Error
}
```

Generické datové typy – Signatura generických metod

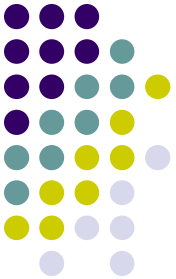


- Omezení jsou u signatur generických metod ignorována.
- Významný je počet generických typových parametrů jako i seřazení pozic typových parametrů.

```
class A {}
class B {}
interface IX {
  T F1<T>(T[] a, int i); // Error
  void F1<U>(U[] a, int i); // Error
  void F2<T>(int x); // Ok
  void F2(int x); // Ok
  void F3<T>(T t) where T: A; // Error
  void F3<T>(T t) where T: B; // Error
}
```

- Podobně je tomu i u jiných generických typů.

Generické datové typy – Signatura generických metod

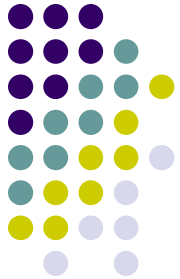


- Omezení jsou u signatur generických metod ignorována.
- Významný je počet generických typových parametrů jako i seřazení pozic typových parametrů.

```
class A {}
class B {}
interface IX {
  T F1<T>(T[] a, int i); // Error
  void F1<U>(U[] a, int i); // Error
  void F2<T>(int x); // Ok
  void F2(int x); // Ok
  void F3<T>(T t) where T: A; // Error
  void F3<T>(T t) where T: B; // Error
}
```

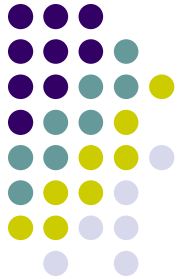
- Podobně je tomu i u jiných generických typů.

Generické datové typy – Vložený typ



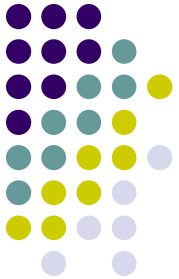
```
class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}
    }
    static void F(T t) {
      Outer<T>.Inner<string>.F(t, "abc"); // These two statements have
      Inner<string>.F(t, "abc"); // the same effect
      Outer<int>.Inner<string>.F(3, "abc"); // This type is different
      Outer.Inner<string>.F(t, "abc"); // Error, Outer needs type arg
    }
 }
```

Generické datové typy – Další vlastnosti



- Statické položky
  - Statické položky jsou sdíleny mezi instancemi stejného uzavřeného zkonstruovaného typu.
  - Statický konstruktor je vykonán právě jednou pro každý uzavřená zkonstruovaný typ.

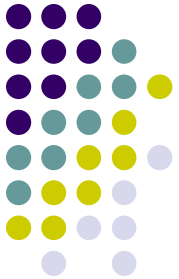
Generické datové typy – Implicitní hodnoty



- Implicitní hodnoty využívají klíčového slova default . Vrací implicitní hodnotu konkrétního typového parametru.
  - **null** pro odkazové typy
  - **0** pro číselné typy
  - **false** pro booleovské typy
  - **'\0'** znakové typy
  - **strukturu** inicializovanou implicitní hodnotou

```
public class C<T>
{
    private T value;
    public T M() {
        return (condition) ? value : default(T);
    }
}
```

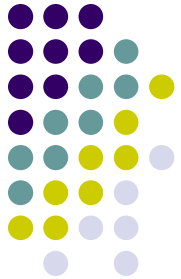
Generické datové typy – Negenerické typy



- Vlastnosti, události, indexery, operátory, konstruktory a destruktory nemohou sami mít typové parametry.
- Mohou se ovšem vyskytovat v generických typech a využívat typového parametru.

```
public class Dictionary<K, V>
{
    public void Add(K key, V value) {...}
    public V this[K key] {...}
}
```

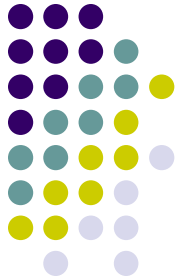
Atributy



- Platforma .NET definuje možnost asociovat libovolné informace (metadata) se zdrojovým kódem aplikace.
- Tyto metadata jsou realizována pomocí atributů.
- Tyto metadata jsou pak součástí assembly.
- Tyto metadata jsem v aplikaci schopni získat pomocí mechanismu reflexe.
- Příklad
  - Jak již bylo řečeno, součástí assembly jsou metadata určující verzi, výrobce, ...
  - Vytvoříme-li projekt pomocí Visual Studia.NET, vytvoří tento nástroj soubor AssemblyInfo.cs. Tento soubor obsahuje atributy vztahující se k celé assembly.
  - Tyto atributy mají následující formát:

    ```
    [assembly: AssemblyVersion("1.0.*")]
    ```
  - Chceme-li definovat atribut pro celou assembly, musíme na začátku uvést *assembly:*.
  - Následuje název atributu a jeho hodnota.

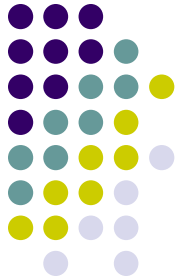
Atributy – AssemblyInfo.cs (1)



```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

Atributy – Příklady použití atributů (1)



- Atribut **Serializable**
  - Tento atribut použijeme pokud chceme, aby nějaká třída (její instance) byla serializována.

```
[Serializable]
class MyClass() ...
```

- Atribut **Obsolete**
  - Atribut určující, že daná metoda je již zastaralá a nebude v další verzi použita.

```
[Obsolete(message,bool)
```

    - message - vysvětlující text, který bude vypisovat překladač.
    - bool je true - použití metody je chyba při překladu.
    - bool je false - použití metody způsobí warning při překladu.

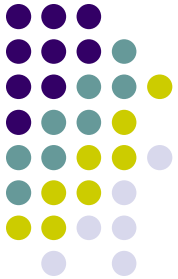
```
class Test
{
  [Obsolete("This method can not be used!",true)]
  public void SomeMethod() {}
}
```

Atributy – Realizace atributů (1)

- Atributy jsou realizovány jako normální třídy.
- Základní třídou pro všechny atributy je třída *System.Attribute*.
- Množinu atributů je možno libovolně rozšiřovat.

```
class MyAttribute : Attribute
{
  private string message;
  public MyAttribute(string message)
  {
    this.message=message;
  }
  public string Message
  {
    get
    {return this.message;}
  }
}
[MyAttribute("Simple class")]
class Example
{
  [MyAttribute("Stupid method")]
  public void SomeMethod() {}
}
```

Atributy – Realizace atributů (2)



```
class RunApp
{
  static void Main(string[] args)
  {
    Type example=typeof(Example);

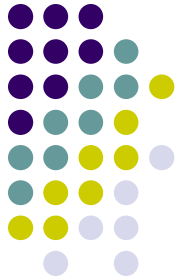
    foreach(MyAttribute myAttribute in
    example.GetCustomAttributes(false))
     Console.WriteLine("Class: "+myAttribute.Message);

    foreach(MemberInfo member in example.GetMembers())
      foreach(MyAttribute myAttribute in
        member.GetCustomAttributes(false))
        Console.WriteLine("Member: "+myAttribute.Message);
  }
}
```

- **Výstup programu bude:**

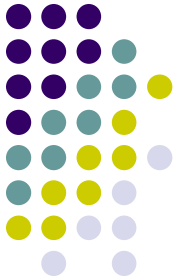
```
Class: Simple class
Member: Stupid method
```

Atributy – Poziční a pojmenované parametry (1)



- Existují dva typy parametrů atributů.
- Poziční parametry se zadávají jako formální parametry konstruktoru a jsou tudíž vždy povinné.
- Pojmenované parametry se zadávají pomocí jména a operátoru přiřazení.

Atributy – Poziční a pojmenované parametry (2)



```
class MyAttribute : Attribute
{
  private string message;
  public MyAttribute(string message)
  {
    this.message=message;
  }
  int number;
  public int Number
  {
    get {return number;}
    set {number=value;}
  }
  public string Message
  {
    get {return this.message;}
  }
}
```

- Atribut pak můžeme použít takto:

```
[MyAttribute("some message")]
[MyAttribute("some message",Number=3)]
```

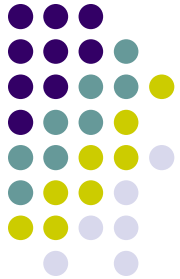
Atributy – Atributy atributů (1)

- Každému atributu můžeme nastavovat určité vlastnosti (metadata k metadatům v podobě atributů:-).

```
[AttributeUsage(destination,
  AllowMultiple=bool, Inherited=bool)]
```

  - destination - typ cíle, pro který je atribut použitelný.
  - Lze povolit násobnou aplikaci.
  - Povolit dědičnost atributu.
  - Cíl atributu je dán výčtovým typem *AttributeTargets*.
    - Assembly, Class, Constructor, Delegate, Enum, Event, Field, Interface, Method, Parameter, Property, RetrurnValue, Struct, All.
  - Cíle lze kombinovat pomocí logického operátoru OR.

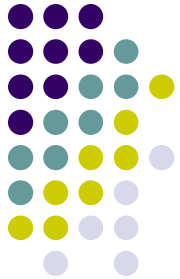
Atributy – Atributy atributů (2)



```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
AllowMultiple=true, Inherited=false)]
class MyAttribute : Attribute {
  private string message;
  public MyAttribute(string message)  {
    this.message=message;
  }
  int number;
  public int Number {
    get {return number;}
    set {number=value;}
  }
  public string Message  {
    get {return this.message;}
  }
}

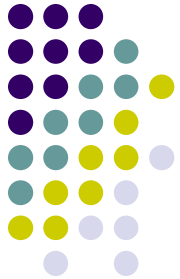
[MyAttribute("Simple class",Number=1)]
[MyAttribute("but very nice!")]
class Example {}
```

Spolupráce s existujícím kódem



- **Autoři platformy .NET a jazyka C# umožnili programátorům použít stávající programy. Používáme-li takovéto programy, zříkáme se výhod, které poskytuje CLR. Hovoříme pak o neřízeném kódu.**
  - **Spolupráce s komponentami modelu COM - schopnost prostředí .NET používat komponenty modelu COM a naopak komponentám modelu COM používat prvky prostředí .NET.**
  - **Spolupráce s běžnými knihovnami DLL - tyto služby umožňují programátorům v prostředí .NET používat knihovny DLL.**
  - **Nezabezpečený kód - umožňuje programátorům v jazyce C# používat například ukazatele. Takto vytvořený program není spravován CLR systému .NET.**

Spolupráce s existujícím kódem - COM interoperability (2)



- Použití existující COM komponenty v .NET aplikaci.
  - Nejprve je nutné COM komponentu registrovat v registry (nástroj regsvr32.exe).
  - Vytvoříme nový projekt v jazyce C#. V okně Solution Explorer aplikace Visual Studio vybereme položku Add Reference.
  - Najdeme příslušnou COM komponentu.
  - Reference na tuto komponentu byla přidána a lze ji normálně používat (tedy instanciovat a pak volat její metody,....).
  - Řekněme, že máme zaregistrovánu COM komponentu example.dll. Tato komponenta je přístupná přes rozhraní IExample

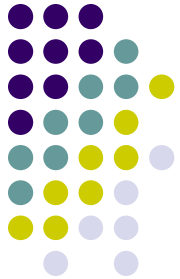
Spolupráce s existujícím kódem - COM interoperability (2)

```
interface IExample
{
  int getNumber();
}
```

- Reference na tuto komponentu pak bude: EXAMPLELib.Example.

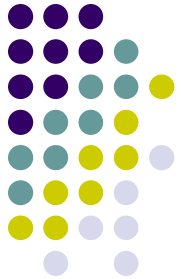
```
class Test
{
  public static void Main()
  {
    EXAMPLELib.Example example = new
EXAMPLELib.Example();
    Console.WriteLine(example.getMessage());
  }
}
```

Spolupráce s existujícím kódem - COM interoperability (3)



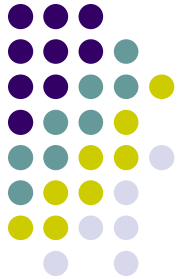
- **Použití .NET komponenty na místo COM komponenty.**
  - **Vytvoříme nový C# projekt typu Class Library.**
  - **Komponentu musíme zaregistrovat v Registry. K tomu složí nástroj regasm.exe.**
  - **Pro programy, které pracují s typovanými knihovnami lze vygenerovat potřebny .tlb soubor nástrojem tlbexp.exe**
  - **Pak lze tuto komponentu využívat stejně jako COM komponentu.**

Spolupráce s existujícím kódem – DLL knihovny



- Platform Invocation Services (PInvoke) - tyto služby umožňují řízenému kódu pracovat s knihovnami a funkcemi exportovanými z dynamických knihoven.
- Knihovna DLL je importovaná pomocí atributu DllImport.
- Importované funkce musí být označeny jako externí (klíčové slovo extern).

Spolupráce s existujícím kódem – DLL knihovny



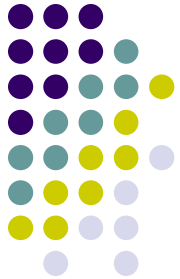
- Následující příklad ukazuje volání metody MessageBox z knihovny User32.dll.

```
using System;
using System.Runtime.InteropServices;

class RunApp
{
  [DllImport("user32.dll")]
  static extern int MessageBoxA(int hWnd, string
msg,string caption,int type);

  static void Main(string[] args)
  {
    MessageBoxA(0,"Hello world!","C# is calling!",0);
  }
}
```

Spolupráce s existujícím kódem – DLL knihovny



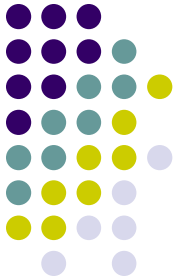
- Importovanou funkci lze pro použití v jazyce C# přejmenovat. Slouží k tomu pojmenovaný parametr EntryPoint atributu DllImport.

```
using System;
using System.Runtime.InteropServices;

class RunApp
{
  [DllImport("user32.dll", EntryPoint="MessageBox")]
  static extern int NiceWindow(int hWnd, string
msg,string caption,int type);

  static void Main(string[] args)
  {
    NiceWindow(0,"Hello world!","C# is calling!",0);
  }
}
```

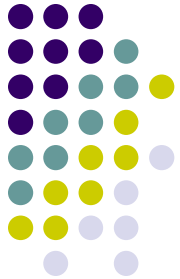
Nezabezpečený kód (1)



- Používáme-li nebezpečný kód, explicitně se zříkáme vlastností CLR.
- Blok je definován pomocí klíčového slova `unsafe` a složenými závorkami.
- Klíčové slovo unsafe lze použít také jako modifikátor u metody případně tříd.
- Je nutné překladači povolit unsafe bloky (Project-Properties-Configuration Properties-Build-Allow Unsafe Code Blocks).

```
class RunApp
{
  public static unsafe void Swap(int *x, int*y)
  {
    int tmp=*x; *x=*y; *y=tmp;
  }
  static void Main(string[] args)
  {
    int a=1; int b=4;
    unsafe
    {
      Swap(&a,&b);
    }
    Console.WriteLine("{0},{1}",a,b); //output: 4,1
  }
}
```

Nezabezpečený kód (2)



- Používáme-li přímý přístup do paměti, mohlo by dojít ke srážce s algoritmem garbage collection.
- Klíčové slovo fixed je nástrojem určeným pro znehybnění kusu paměti.

```
fixed (type* pointer = expression) statement
```

```
unsafe class Class1 {
  public static void Fill(int *array,int size)   {
    while(size-- > 0)*array++ = size+1;
  }
  public static void Main() {
    int[] array=new int[10];
    fixed(int* pointerToArray = array) {
      Fill(pointerToArray,10);
    }
    foreach(int field in array) Console.WriteLine(field);
  }
}
```