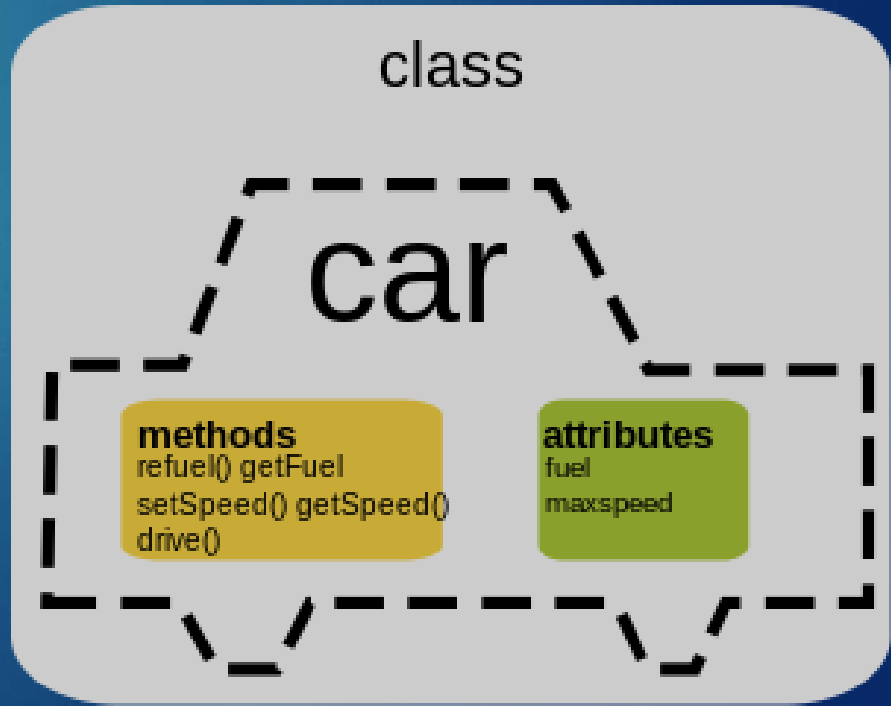
Objektovo orientované programovanie – základné pojmy

KOLEKTÍV AUTOROV

KEGA 038STU-4/2018 - KONVERGENCIA AUTOMATIZÁCIE A POKROČILÝCH IKT

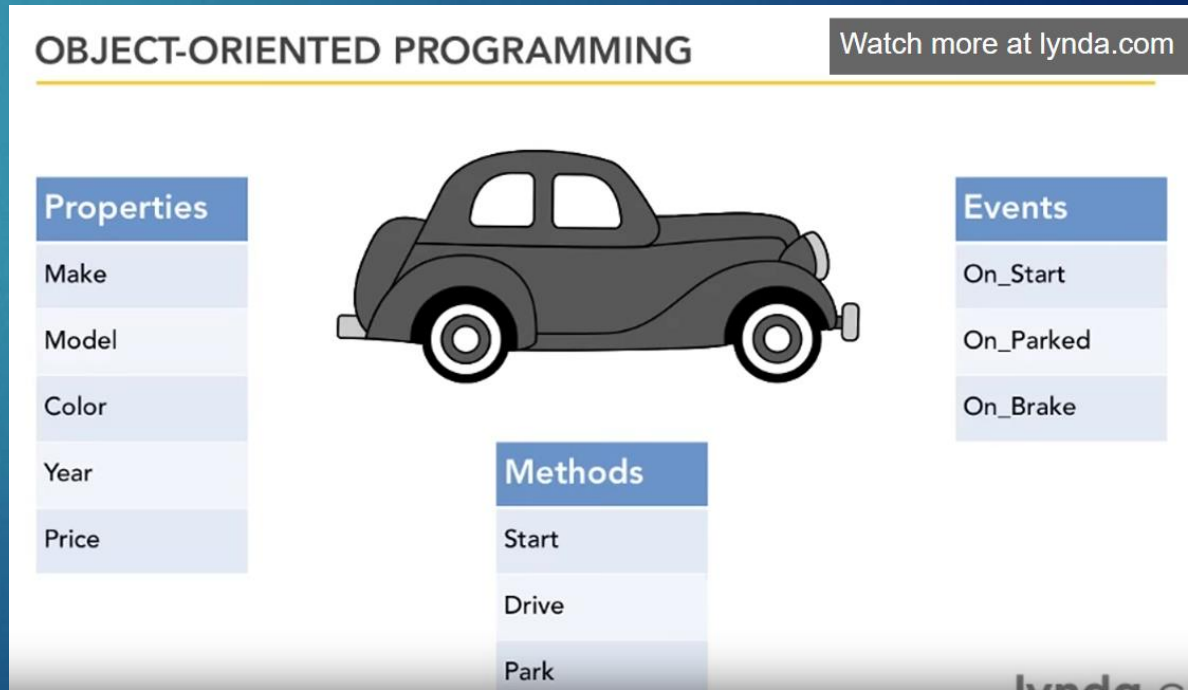
Objektovo orientované programovanie

- **Objektové programovanie** alebo **objektovo orientované programovanie** (z angl. Object-oriented programming, skratka **OOP**) je metodika vývoja softvéru založená na používaní dátových štruktúr nazývaných **objekty** a ich interakcie na vývoj aplikácií
- Princípy objektového programovania boli rozpracované už v 70. rokoch 20. storočia, no širšie sa vo vývoji softvéru začalo uplatňovať až koncom 20. storočia
- Objekty majú svoje vlastnosti, metódy a udalosti, pomocou ktorých objekt vykonáva určité činnosti, na ktoré bol naprogramovaný
- Z obsahového hľadiska vlastnosti typu trieda sú vo svojej podstate položky typu záznam. Metódy a udalosti sú svojim charakterom funkcie a procedúry



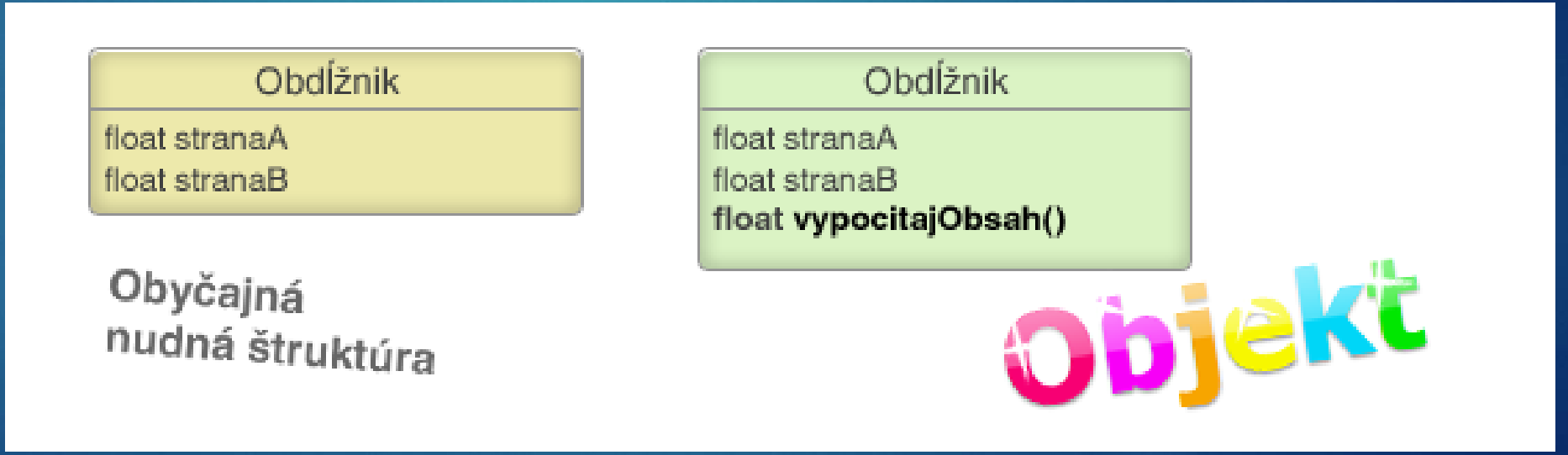
Objektovo orientované programovanie

- Udalosťou sa nazýva každá zmena stavových veličín, napr. Click, DoubleClick, stlačenie klávesy na klávesnici, impulz z časovača, zmena veľkosti okna, zatvorenie okna
- Základom objektového programovania je dátový typ **trieda**
- Dátový typ trieda je odvodený dátový typ a vychádza z dátového typu štruktúra (v jazyku Pascal, Delphi - záznam (record), C, C++ - štruktúra (struct))
- **Objekt - premenná typu trieda**
- Existuje mnoho programovacích jazykov používajúcich princíp OOP, napr.: Visual Basic, C++, C Sharp, Java, Python, PHP a mnoho iných
- OOP dramaticky zlepšuje prehľadnosť kódu, uľahčuje jeho písanie a zvyšuje jeho znovupoužiteľnosť



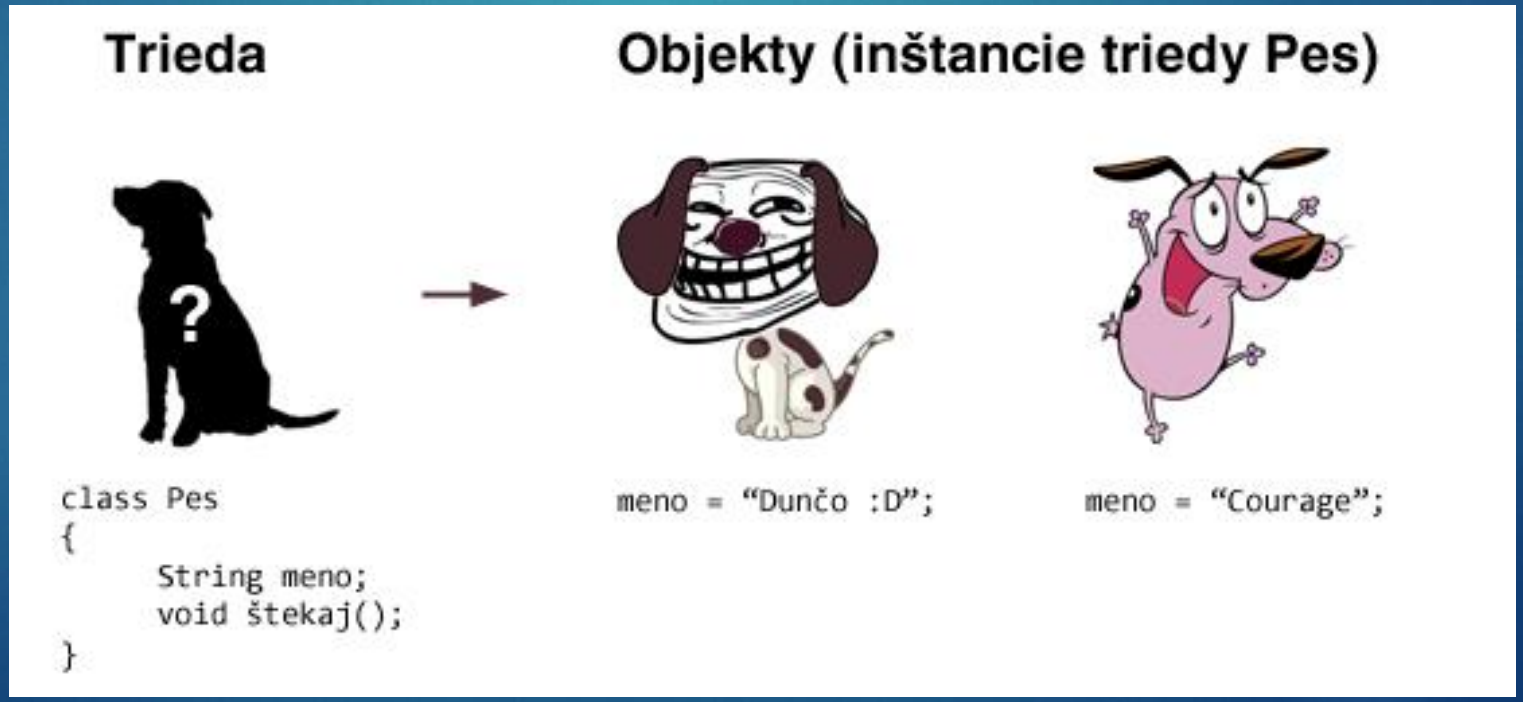
Objekt

- Objekty sú v podstate iba štruktúry obsahujúce rôzne premenné a s nimi súvisiace funkcie
- Premenné vo vnútri objektov zvyknú byť nazývané **members** (dátové členy) alebo **fields** (políčka) alebo aj **properties** (vlastnosti) a funkcie sa volajú **methods** (metódy)



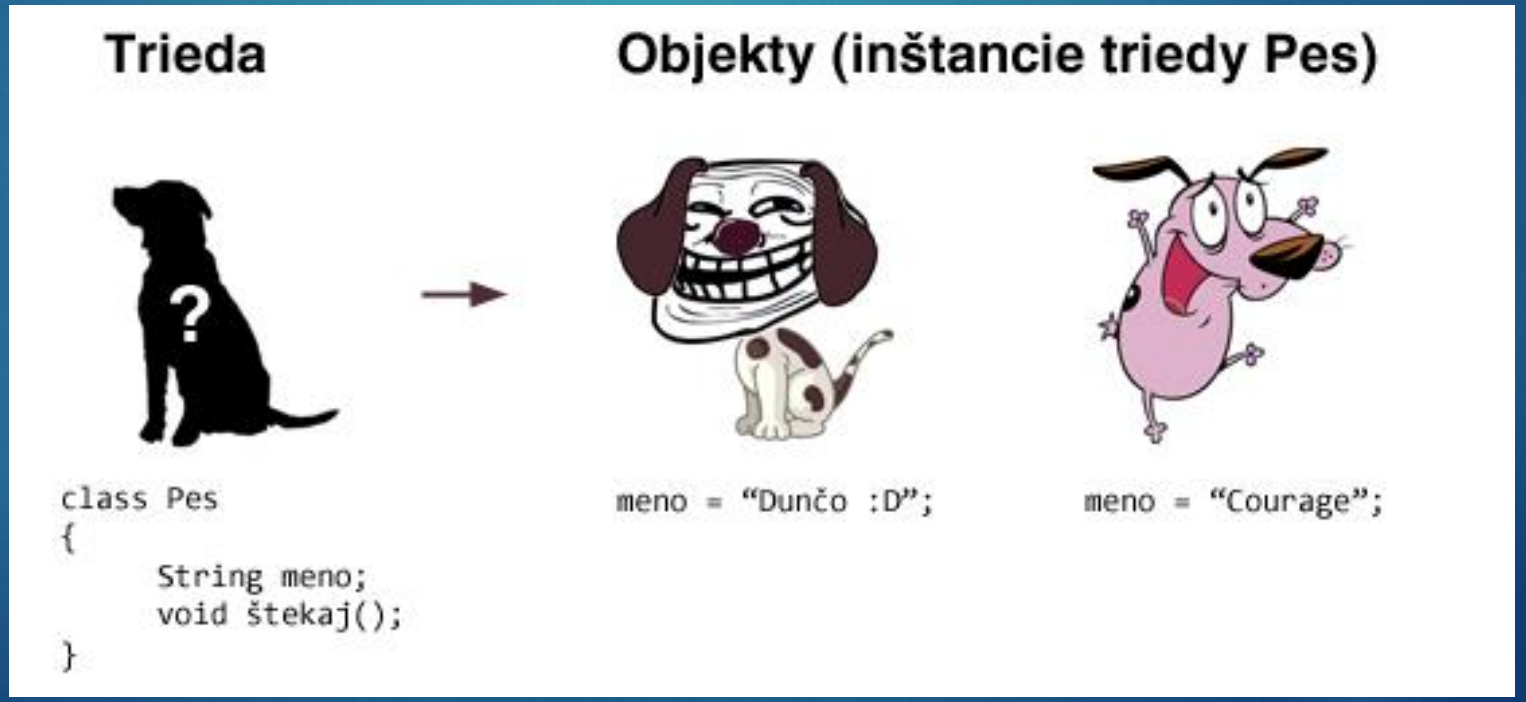
Trieda

- Napriek tomu, že hovoríme o objektovo orientovanom programovaní sa v samotných programoch málokedy stretnete s výrazom "**object**“
- Všade na vás pozerá kľúčové slovo "**class**“, po slovensky **trieda**
- Triedy sú vlastne šablónami, podľa ktorých sa vytvárajú objekty



Trieda

- Ako vidíte na obrázku, objekty sú konkrétne **výskyty** (**inštancie**) nejakej triedy
- Objekty sa navzájom líšia dátami, ktoré nesú (toto sa občas zvykne nazývať "stav objektu" - state)



Ukážka pseudokódu, konštruktor

- V nasledujúcom (pseudo)kóde si môžete všimnúť špeciálne metódy s rovnakým názvom ako trieda
- Sú to *konštruktory*, ktorých pomenovanie vychádza z toho, že sa volajú vždy keď sa vytvára/konštruuje nový objekt a slúžia na jeho inicializáciu
- Vytváranie objektu ("inštanciáciu") je vidno nižšie, v riadku s kľúčovým slovom *new*
- Definujeme 2 konštruktory, aby sme obdĺžniky mohli vytvárať buď bez parametrov (vtedy budú mať strany dĺžku 0) alebo s dvoma parametrami

```
class Obdlznik {
        float a, b; // toto su datove cleny

        // a tu nasleduju metody:
        Obdlznik() {
                a = b = 0;
        }
        Obdlznik(vyska, sirka) {
                a = vyska; b = sirka;
        }
        float vypocitajObsah() {
                return a * b;
        }
}

Obdlznik velkyObdlznik;
velkyObdlznik = new Obdlznik (15, 25);
println("Obsah velkeho obdlznika je: " + velkyObdlznik.vypocitajObsah());

velkyObdlznik.a = 1000; // chceme este vacsi a obdlznikovitejsi! ...zaruceny sposob zvacsenia
vasho obdlznika ;)
println("Obsah velkeho obdlznika je: " + velkyObdlznik.vypocitajObsah());
```

Dedičnosť (Inheritance)

- Jednou zo základných výhod OOP je **dedičnosť (**angl. **inheritance)**
- Ak ste chceli v štandardnom, procedurálnom programovaní zmeniť časť správania v určitých prípadoch, kód bolo treba znečisťovať všelijakými podmienkami
- Dedičnosť toto eliminuje tak, že môžete od existujúcej triedy odvodiť ďalšiu, pričom získa všetky vlastnosti pôvodnej triedy a vy môžete poprepisovať a popridávať, čo treba

Dedičnosť (Inheritance)

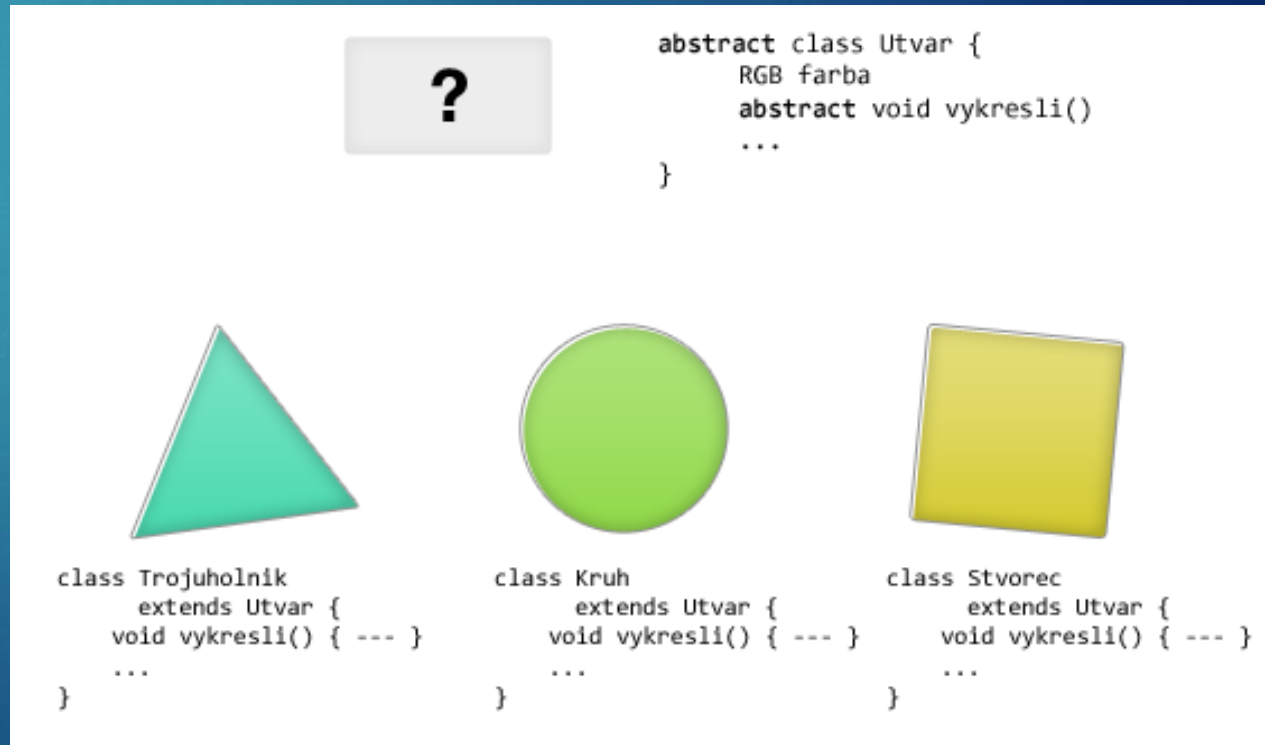
- Turbo-tlačidlo **zdedilo** metódu **onLeftClick()** a pridalo metódy, ktoré by sa mohli volať pri pravom kliku, strednom kliku, atď.
- Navyše prepísalo vykresľovaciu metódu obyčajného tlačidla vlastnou metódou, keďže sa neuspokojí len tak s hocijakým vzhľadom
- Jednoducho je náročnejšie, ale stále vychádza z rovnakého základu a správa sa podobne ako naše obyčajné tlačidlo, takže ho "zdedíme"

Dedičnosť (Inheritance)

- Prepisovanie metód (v našom prípade to bola metóda *kresli()*) sa v originále volá "**method overriding**", čo je slovo mierne podobné s overloadingom
- **Overloading** (**preťažovanie**) je ale niečo iné - existencia viacero funkcií(metód) s *rovnakým názvom a rôznymi parametrami* (príklad ste videli s konštruktormi obdĺžnika), takže pozor

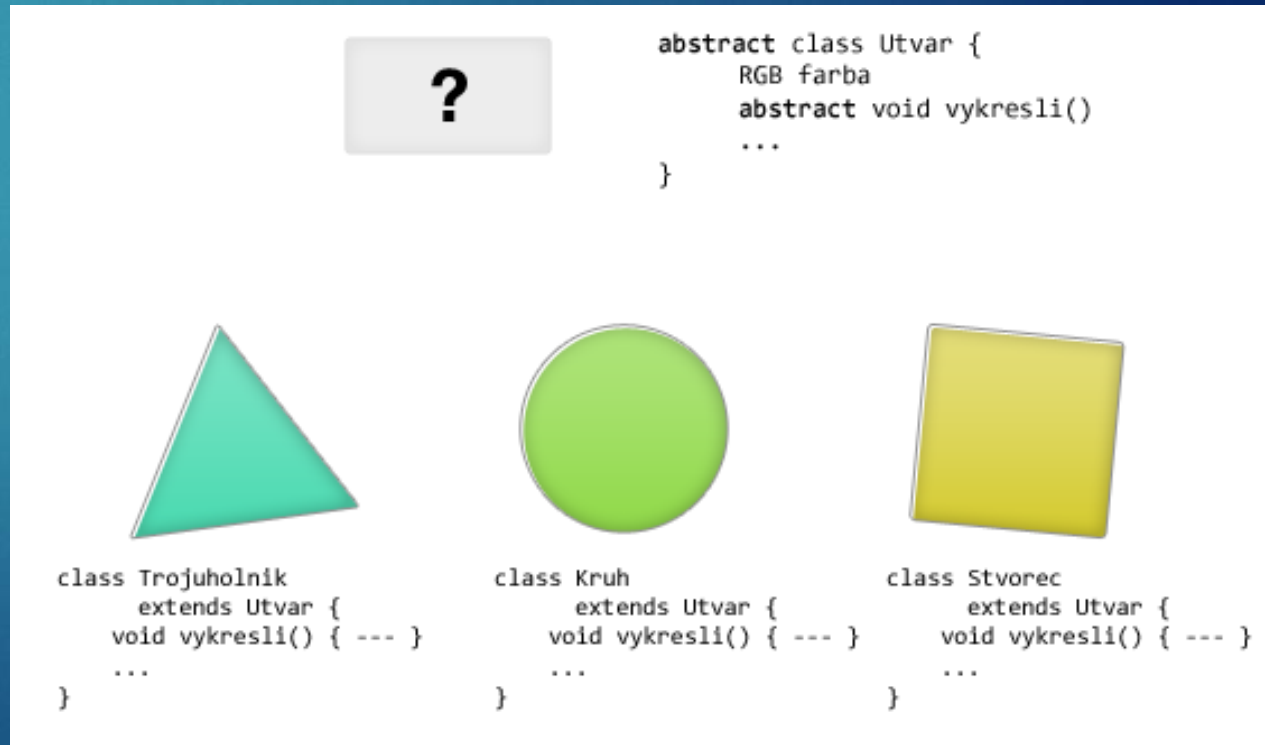
Abstrakcia

- Pod abstrakciou sa vo všeobecnosti chápe zameranie sa na **kľúčové vlastnosti** nejakého prvku reálneho sveta (alebo aj nereálneho)
- V OOP to zužitkujeme to hlavne pri abstraktných triedach, čo ukážeme na príklade s útvarmi, ktoré budeme vykresľovať
- Abstraktná trieda *Utvar* určuje, že všetky od nej odvodené triedy sa budú dať vykresliť a budú mať nejakú farbu
- Abstraktná trieda sa sama o sebe nedá vytvoriť, čo dáva zmysel, lebo taký "útvar" nemá žiadnu konkrétnu podobu



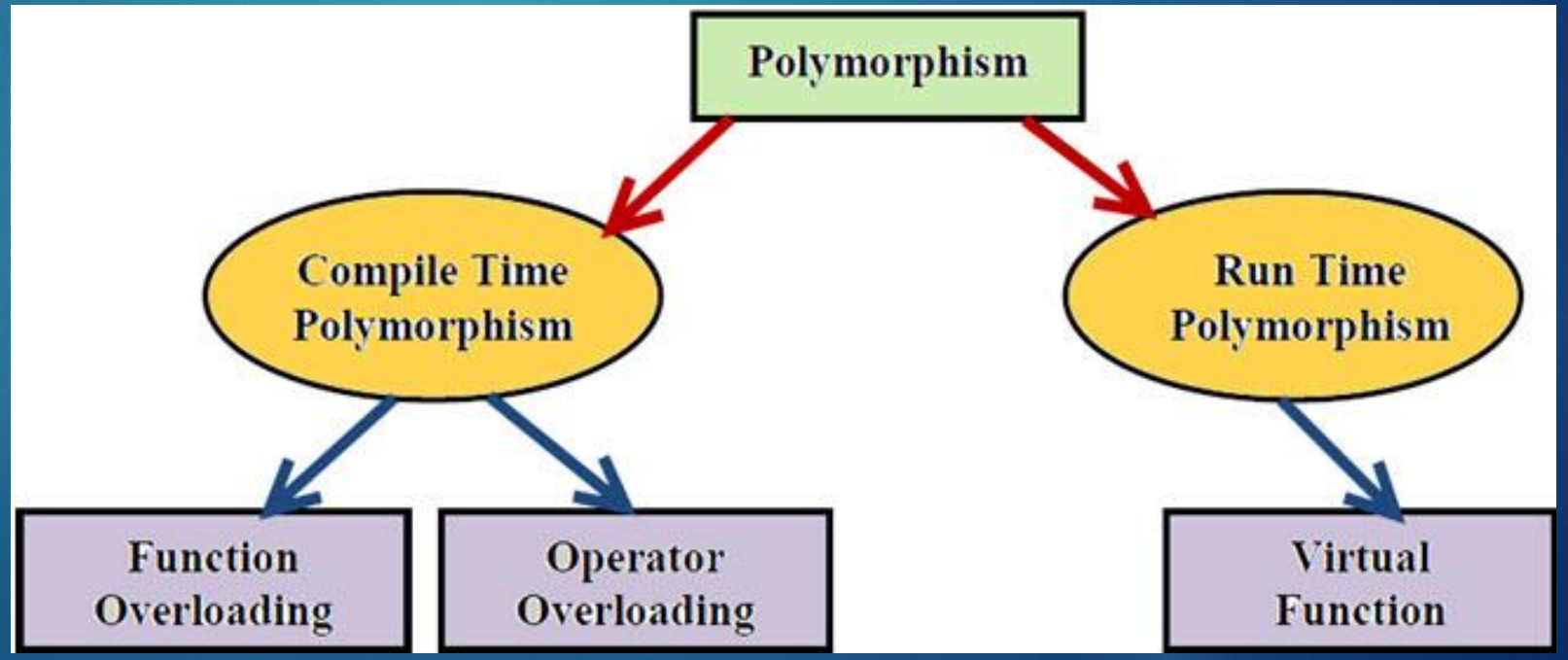
Abstrakcia

- Vytvoriť sa dajú len zdedené triedy, ktoré v tomto prípade implementujú vykresľovaciu funkciu
- Triedy môžu byť aj polo-abstraktné v zmysle, že môžete implementovať niektoré metódy a ostatné nechať neimplementované
- Podobnou záležitosťou ako abstraktné triedy sú v niektorých jazykoch (aj C#) **rozhrania** ("**interface**")
- Tie obsahujú iba metódy a žiadna z nich nie je implementovaná



Polymorfizmus

- V OOP ho možno vidieť v rôznych konceptoch
- Polymorfizmus je mechanizmus, ktorý umožňuje objektom rôznych typov odpovedať na volanie rovnakej metódy rôznym spôsobom
- Polymorfizmus (viacznačnosť) je schopnosť objektu nadobúdať viacero foriem



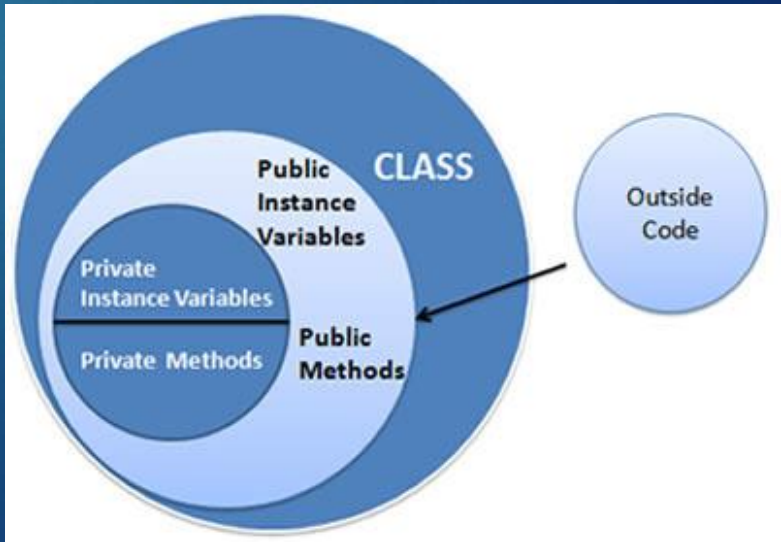
Zapuzdrenie (encapsulation)

- **Zapuzdrenie** skrýva vnútorné fungovanie triedy pred ostatnými triedami
- To je dobré ak chcete zabrániť používaniu vnútorných mechanizmov, ktoré by sa mohli zmeniť - napríklad vytvoríte triedu, ktorá sa pripája na server a využíva na to určité rozhranie - vy sa po istom čase rozhodnete, že vymeníte rozhranie za lepšie a rýchlejšie
- Ak by ste ponechali k nemu prístup všetkým programátorom, mohlo by sa stať, že niekto napíše kód, ktorý priamo používa špecifické funkcie tohoto rozhrania, takže pri jeho výmene by sa celý program rozpadol
- Vďaka zapuzdreniu bude obmedzený na vami sprístupnené funkcie, ktoré by ste prispôsobili zmene rozhrania a celý program funguje ďalej, ako by sa nič nezmenilo
- Takto vytvárate tzv. voľne viazané triedy, ktoré majú oveľa vyššiu znovupoužiteľnosť ako keby sa každá trieda vŕtala vo vnútorných záležitostiach iných tried

Zapuzdrenie (encapsulation)

- Možno ste sa už stretli so slovami *public, protected a private* – sú to modifikátory prístupu
- **public** dovoľuje prístup k metóde/údajom komukoľvek
- **protected** prvky sú dostupné iba zvnútra triedy a odvodených tried
- **private** čisto iba z triedy samotnej

- V súvislosti so zapuzdrením je často vidieť metódy *getNiečo()* a *setNiečo()*
- Get/set metódy majú význam v situáciách:
  - 1) keď sa hodnota pred uložením alebo pred vrátením musí nejako spracovať
  - 2) ak potrebujete zamedziť prepisovaniu alebo čítaniu hodnoty



Zapuzdrenie (encapsulation)

- Všimnite si špeciálnu premennú **this**
- Tá obsahuje vždy inštanciu aktuálneho objektu a v tomto prípade ju používame, aby sme vedeli odlíšiť parameter "meno" a dátového člena s rovnakým názvom
- Inak sa používa ak chcete inému objektu odkázať na "seba"

```
class Pes {
        private String meno; // ked dame psovi meno, uz ho nenaucime ine

        public Pes(String meno) {
                this.meno = meno;
        }
        public String getMeno() {
                return meno;
        }
}

Pes dunco = new Pes("Dunco");
dunco.meno = "Lolofon"; // toto by ste nemohli spravit
```

Použité zdroje

- http://wikipedia.org
- http://www.zajtra.sk/programovanie/165/objektovo-orientovane-programovanie-v-normalnej-ludskej-reci
- http://youtube.com

Objektovo orientované programovanie v C#

KOLEKTÍV AUTOROV

KEGA 038STU-4/2018 - KONVERGENCIA AUTOMATIZÁCIE A POKROČILÝCH IKT

Statické členy

- Statické členy nie sú takým pilierom OOP ako dedičnosť alebo polymorfizmus, ale sú užitočným prvkom
- K statickým členom a metódam môžete pristupovať aj keď nemáte vytvorenú žiadnu inštanciu, dokonca môžete vytvárať celé statické triedy

```
class Pes {
        private String meno; // ked dame psovi meno, uz ho nenaucime ine
        private static int pocet = 0;

        public Pes(String meno) {
                this.meno = meno;
                pocet++;
        }
        public String getMeno() {
                return meno;
        }
        public static int getPocet() {
                return pocet;
        }
}

Pes dunco = new Pes("Dunco"), e3 = new Pes("Ezechiel Treti");
Pes.getPocet(); // funguje toto
dunco.getPocet(); // aj toto.. ale dava to ovela mensi zmysel
```

Namespace

- Triedy sú v drvivej väčšine organizované do menných priestorov (**namespace**)

```
using Syncfusion;
using System;

using Crypto = System.Security.Cryptography;

namespace NamespaceDemo
{
    class Program
    {
        static void Main()
        {
            double hypotenuse = Calc.Pythagorean(2, 3);
            Console.WriteLine("Hypotenuse: " + hypotenuse);

            Crypto.AesManaged aes = new Crypto.AesManaged();

            Console.ReadKey();
        }
    }
}
```

Implementácia getterov a setterov

```
double result;

public double Result
{
    get { return result; }
    set { result = value; }
}
```

=

```
public double Result { get; set; }
```

Override a virtual

- „Overridnuť“ môžeme v C# len metódu, ktorá je označená ako **virtual**
- V Jave sú všetky metódy automaticky virtuálne
- V ProgrammerCalculator by sme sa k pôvodnej metóde dostali pomocou kľúčového slova **base** – napríklad **base.Add(num1, num2)**

```
using System;

public class Calculator
{
    public virtual double Add(double num1, double num2)
    {
        Console.WriteLine("Calculator Add called.");
        return num1 + num2;
    }
}

public class ProgrammerCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ProgrammerCalculator Add called.");
        return MyMathLib.Add(num1, num2);
```

Abstraktné triedy

- Nemôžeme vytvoriť objekt **abstraktnej triedy**, musíme tak vytvoriť triedu, ktorá ju bude dediť
- Ide o niečo podobné ako interface (bude vysvetlené ďalej)
- Abstraktná trieda môže obsahovať tiež **abstraktné metódy**, ktoré treba implementovať v odvodených triedach, tieto metódy sú tak implicitne virtuálne

```
public abstract class Calculator
{
    public abstract double Add(double num1, double num2);
}
```

Interface

- **Interface** je vlastne abstraktná trieda, avšak nemá implementované žiadne metódy
- Implementáciu treba napísať do odvodenej triedy
- Trieda môže implementovať viacero interfacov (abstraktnú resp. hocijakú triedu len jednu)
- Zvyknú sa pomenúvať tak, že prvé písmeno je `I`

```
public interface ICalculator
{
    double Add(double num1, double num2);
}
```

```
public class ScientificCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ProgrammerCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

Partial class

Two classes can be in the same namespace

Take a look at these two class files from a program called `PetFiler2`. They've got three classes: a `Dog` class, a `Cat` class, and a `Fish` class. Since they're all in the same `PetFiler2` namespace, statements in the `Dog.Bark()` method can call `Cat.Meow()` and `Fish.Swim()`. It doesn't matter how the various namespaces and classes are divided up between files. They still act the same when they're run.

SomeClasses.cs

```
namespace PetFiler2 {

  class Dog {

    public void Bark() {
      // statements go here
    }

  }

  partial class Cat {

    public void Meow() {
      // more statements
    }

  }
}
```

When a method is "public" it means every other class in the namespace can access its methods.

MoreClasses.cs

```
namespace PetFiler2 {

  class Fish {

    public void Swim() {
      // statements
    }

  }

  partial class Cat {

    public void Purr() {
      // statements
    }

  }
}
```

Since these classes are in the same namespace, they can all "see" each other—even though they're in different files. A class can span multiple files too, but you need to use the "partial" keyword when you declare it.

You can only split a class up into different files if you use the "partial" keyword. You probably won't do that in any of the code you write in this book, but the IDE used it to split your page up into two files so it could put the XAML code into MainPage.xaml and the C# code into MainPage.xaml.cs.

There's more to namespaces and class declarations, but you won't need them for the work you're doing right now. Flip to #3 in the "Leftovers" appendix to read more.

Generics

KOLEKTÍV AUTOROV

KEGA 038STU-4/2018 - KONVERGENCIA AUTOMATIZÁCIE A POKROČILÝCH IKT

Aký problém vieme riešiť s generickými typmi?

- Predstavme si, že chceme **implementovať zoznam čísel typu integer** – nižšie vidíme napríklad metódu Add
- Ak chceme implementovať podobný **zoznam kníh**, potrebujeme vytvoriť ďalšiu triedu => **neefektívne**

```
using System;

namespace Generics
{
    public class List
    {
        public void Add(int number)
        {
            throw new NotImplementedException();
        }

        public int this[int index]
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

```
using System;

namespace Generics
{
    public class BookList
    {
        public void Add(Book book)
        {
            throw new NotImplementedException();
        }

        public Book this[int index]
        {
            get { throw new NotImplementedException(); }
        }
    }
}
```

Neefektívne riešenie problému

- Ako možné riešenie problému sa ponúka nižšie uvedený variant
- Je však z hľadiska výkonu aplikácie veľmi neefektívny

```
public class ObjectList
{
    public void Add(object value)
    {

    }

    public object this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```

Riešenie problému pomocou generického typu

- Namiesto konkrétneho typu premennej/objektu napíšeme zátvorky <> a do nich napríklad písmeno `T` ako **template – šablóna**
- Teraz môžeme vytvoriť inštanciu zoznamu pre nami požadovaný typ premennej alebo objektu

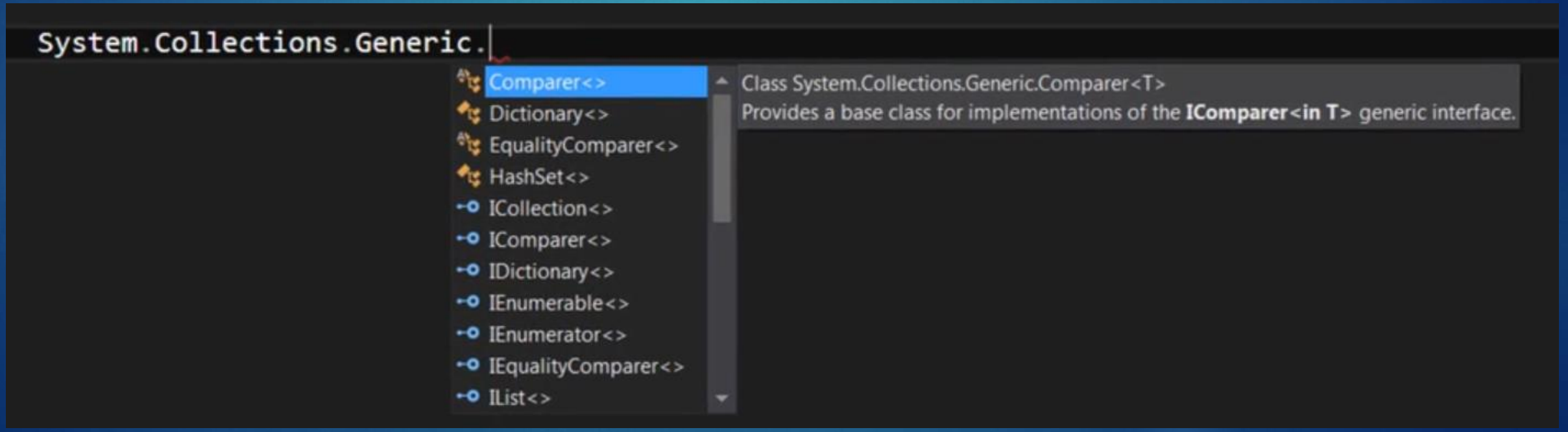
```
public class GenericList<T>
{
    public void Add(T value)
    {

    }

    public T this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var book = new Book { Isbn = "1111", Title = "C# Advanced" };

        //var numbers = new List();
        //numbers.Add(10);

        //var books = new BookList();
        //books.Add(book);

        var numbers = new GenericList<int>();
        numbers.Add(10);

        var books = new GenericList<Book>();
        books.Add(new Book());
    }
}
```

class Generics.Book

Použitie generického typu v praxi

- V praxi budete skôr používať preddefinované generické zoznamy ako vytvárať nové
- Nájdete ich na stránke: https://msdn.microsoft.com/en-us/library/system.collections.generic(v=vs.110).aspx



Použité zdroje

- http://wikipedia.org
- http://www.zajtra.sk/programovanie/165/objektovo-orientovane-programovanie-v-normalnej-ludskej-reci
- Syncfusion e-books
- Programming with Mosh
- http://youtube.com
- https://www.youtube.com/watch?v=pTB0EiLXUC8 - Object-oriented Programming in 7 minutes | Mosh