23. Třídy, generické třídy, instance, skládání, statické metody a proměnné. Zapouzdření, konstruktory, konzistence objektu, zpřístupnění vnitřní implementace, modifikátory public a private. Polymorfismus, dědičnost, překrývání, abstraktní třídy a metody. (A7B36OMO)

Třídy

Třídy slouží jako "plánek" na vytváření objektů - jednotlivých instancí tříd. Třída popisuje, z jakých prvků se objekt skládá. Tyto prvky lze rozdělit na datové členy a metody, které obvykle pracují nad datovými členy daného objektu. Datové členy popisují stav objektu, zatímco metody popisují chování (state and behaviour).

Instance

Instance, objekty, třídní instance, třídní objekty a instance objektů popisují stejnou skutečnost. Jak název napovídá jsou to jednotlivé instance tříd. Instance vznikají zavoláním konstruktoru třídy, od které chceme instanci. Lepé tuto situaci popisuje následující přiklady.

Příklad popisující vztah tříd a instancí

Analogií může být následující situace. Návrhář (programátor) navrhne plánek auta - z čeho se auto skládá a co umí. Následně zavolá bandu dělníků (konstruktor), aby mu vytvořili auto podle plánku (instanci). Vytvořené auto žije již vlastním životem, pokud ho přebarvíme/změníme jeho stav, neovlivní to plánek (třídu). Výjímkou jsou statické proměnné, které jsou sdílené pro všechny instance.

Příklad v C#

Analogii auta se nyní pokusíme namodelovat pomocí programovacího jazyka C#. Volba programovacího jazyka není podstatná, princip je ve většině objektově orientovaných jazycích stejný. Poznámka pro rejpaly, následující program je pro pochopení značně zjednodušený. Správně by měl využít C# properties a strukturu Color.

Pokud tento program spustíme, vypíše následující.

car1 má barvu Red

car2 má barvu Red

car1 má barvu Blue

car2 má barvu Red

```
/// <summary>
/// Třída auta
/// </summary>
public class Car
{
	// Datové členy, obecně jich může být libovolné množství

	/// <summary>
	/// Datový člen, který popisuje, jakou barvu má existující auto
	/// </summary>
	private string m_color;

	/// <summary>
	/// Konstruktor
	/// </summary>
	public Car()
	{
		// Při výrobě mají všechny auta červenou barvu
		m_color = "Red";
	}

	// Následují metody

	/// <summary>
	/// Přebarví auto
	/// </summary>
	/// <param name="color"></param>
	public void ChangeColor(string color)
	{
		m_color = color;
	}

	/// <summary>
	/// Metoda, která nám řekne, jakou barvu má auto.
	/// </summary>
	/// <returns></returns>
	public string GetColor()
	{
		return m_color;
	}
}

class Program
{
	static void Main(string[] args)
	{
		// Vytvoříme dvě instance auta,
		// new Car() zavolá konstruktor a obdržíme instanci třídy
		Car car1 = new Car();
		Car car2 = new Car();

		// Vypíšeme barvu obou aut.
		Console.WriteLine("car1 má barvu " + car1.GetColor());
		Console.WriteLine("car2 má barvu " + car2.GetColor());

		// Změníme barvu jednoho auta a opět vypíšeme
		car1.ChangeColor("Blue");
		Console.WriteLine("car1 má barvu " + car1.GetColor());
		Console.WriteLine("car2 má barvu " + car2.GetColor());
	}
}
```

Generické třídy

Generické třídy umožňují parametrizovat některý datový člen. Pokud v době psaní třídy není znám datový člen nebo ho nepotřebuju znát, můžu ho parametrizovat. Výhodou tohoto přístupu je znovupoužitelnost kódu a v případě C++ i optimalizace kódu během kompilace. Typickým příkladem jsou kolekce, které mohou udržovat jakýkoliv typ a zároveň být strongly typed (specifikovat přesně jaký typ objektu obsahují).

Pro úplnost uvádím příklad, jak napsat generickou třídu v C#. V tomto příkladě odstraníme typovou závislost barvy z předchozího příkladu. Jednou ji budeme reprezentovat jako string po druhé strukturou Color. Pokud bych nepoužil generickou třídu, musel bych ji psát dvakrát.

```
/// <summary>
/// Třída s parametrem typu barvy (jak budeme barvu reprezentovat).
/// </summary>
/// <typeparam name="TColor"></typeparam>
class Car<TColor>
{
	private TColor m_color;

	public Car(TColor color)
	{
		// V tomto příkladě nám někdo musí sdělit defaultní barvu, protože v době psaní
		// třídy ji nemůžeme určit.
		m_color = color;
	}

	public void ChangeColor(TColor color)
	{
		m_color = color;
	}

	public TColor GetColor()
	{
		return m_color;
	}
}

class Program
{
	static void Main(string[] args)
	{
		// barvu budeme ukládát reprezentovat stringem
		Car<string> car1 = new Car<string>("Red");

		// barvu budeme reprezentovat strukturou Color, která je součástí C#/.NET
		Car<Color> car2 = new Car<Color>(Color.Red);
	}
}
```

Skládání

Třídy obsahují datové členy, které jsou různých typů od int, string, float až po uživatelem definované typu. Objekty tedy mohou obsahovat jiné objekty, tomuto přístupu se říká skládání.

Statické metody a proměnné

Třídy mohou mimo jiné obsahovat i statické proměnné. Tyto proměnné nepřísluší žádnému objektu a existuji pouze jedenkrát pro celou třídu. Dokonce není ani nutné vytvářet instance této třídy, pro přístup k těmto prvkům. Statické metody mají přístup pouze ke statickým proměnným a lze je volat aniž by byla vytvořená instance třídy. Většina programovacích jazyků využívá klíčového slova static před deklarací.

Příklad ze kterého je vidět, že statická proměnná existuje pouze jedenkrát pro celou třídu.

```
public class MyClass
{
    /// <summary>
    /// Statická proměnná
    /// </summary>
    private static int m_staticCounter;
    private int m_nonStaticCounter;

    /// <summary>
    /// Statická metoda
    /// </summary>
    public static void IncrementStatic()
    {
        m_staticCounter++;
    }

    public void IncrementNonStaticCounter()
    {
        m_nonStaticCounter++;
    }

    public void PrintCounters(string identifier)
    {
        Console.WriteLine(identifier + " static counter: " + m_staticCounter);
        Console.WriteLine(identifier + " nonStatic counter: " + m_nonStaticCounter);
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyClass.IncrementStatic();
        MyClass c = new MyClass();
        MyClass c2 = new MyClass();
        c.PrintCounters("c");
        c2.PrintCounters("c2");
        Console.WriteLine();
        c.IncrementNonStaticCounter();
        MyClass.IncrementStatic();
        c.PrintCounters("c");
        c2.PrintCounters("c2");
    }
}
```

Výpíše

c static counter: 1
c nonStatic counter: 0

c2 static counter: 1
c2 nonStatic counter: 0

c static counter: 2
c nonStatic counter: 1
c2 static counter: 2
c2 nonStatic counter: 0

Zapouzdření

Zapouzdření je způsob propojení atributů a funkcí prostřednictvím objektu. Jedinou možností přístupu k atributům a funkcím objektu je vytvoření instance objektu. Zapouzdření umožňuje programátorovi umístit atributy a procedury do třídy a následně stanovat pravidla, která budou sloužit ke kontrole přístupu.

Například v jednom z předchozích příkladů bychom mohli umožnit změnu barvy auta pouze při splnění nejaké podmínky. Tuto podmínku, bychom napsali do metody ChangeColor.

Konstruktor

Konstruktor je funkce/metoda ve třídě, která má stejný název jako třída a nevrací žádnou hodnotu (Teoreticky vrací instanci dané třídy. Tento návratový typ je ale předem znám, proto se nepíše). Používá se k inicializaci objektu.

Konzistene objektu

- Souvisí se zapouzdřením (dobré zapouzdření napomáhá konzistenci)
- Přístup k objektu pouze přes dané rozhraní (metody), zbytek by měl být private
- Public metody měnící private proměnné musí zajistit jejich správnou úpravu tak, aby instance byla v konzistentním stavu (např. setter znemožní nastavit v určitých případech do proměnné null apod.)
- Lidsky řečeno, konzistence objektu znamená, že objekt by měl být stabilní za každého stavu (neměl by vyhazovat runtime exceptions)

Zpřístupnění vnitřní implementace

- Viz konzistence objektu a zapouzdření
- Objekt by měl být jakási tajemná „krabička“, která vyjadřuje co umí přes veřejné rozhraní (veřejné rozhraní by teda mělo být psáno tak, aby byl jasný účel objektu), ale JAK toho dosahuje by mělo být skryto.
- Ke skrytí vnitřní implementace slouží klíčové slovo private případně protected

Modifikátory public, private, protected

U jednotlivých členů třídy (atributy i metody) můžeme použít modifikátory přístupu. Modifikátor je nutné aplikovat na každého člena zvlášť.

Mezi typické modifikátory patří:

- public – člen označený tímto modifikátorem je dostupný bez omezení
- private – člen je přístupný pouze členům stejné třídy

- protected – přistupovat k takovému členu můžeme uvnitř vlastní třídy a ve všech zděděných třídách

Poznámka: Popis těchto modifikátorů je platný pro Javu a C# apod. jazyky. V C++ je přístup ke zděděným členům zkomplikován modifikátorem dědění.

Příklad ukazující modifikátory přístupu

```
public class BaseCar
{
    public int m_weight;
    protected Color m_color;
    private int m_length;
}

public class ExtendedCar : BaseCar
{
    public ExtendedCar()
    {
        m_weight = 1200;
        m_color = Color.Red;
        // Nelze napsat
        //m_length = 250;
    }
}

class Program
{
    static void Main(string[] args)
    {
        BaseCar car = new BaseCar();
        car.m_weight = 1300;
        // Nelze napsat
        // car.m_color = Color.Red;
        // car.m_length = 250;
    }
}
```

Polymorfismus

Polymorfismus v programování znamená, že metody se stejnou signaturou mohou provádět mnoho různých činností. Při implementaci se využívá dědění a virtuálních metod.

Příklad

Výstupem je:

HAFHAF

MNAU

```
public abstract class Animal
{
	public abstract String MakeASound();
}

public class Dog : Animal
{
	public override string MakeASound()
	{
		return "HAFHAF";
	}
}

public class Cat : Animal
{
	public override string MakeASound()
	{
		return "MNAU";
	}
}

class Program
{
	static void Main(string[] args)
	{
		Animal[] animals = new Animal[] { new Dog(), new Cat() };
		foreach (Animal animal in animals)
		{
			Console.WriteLine(animal.MakeASound());
		}
	}
}
```

Dědičnost

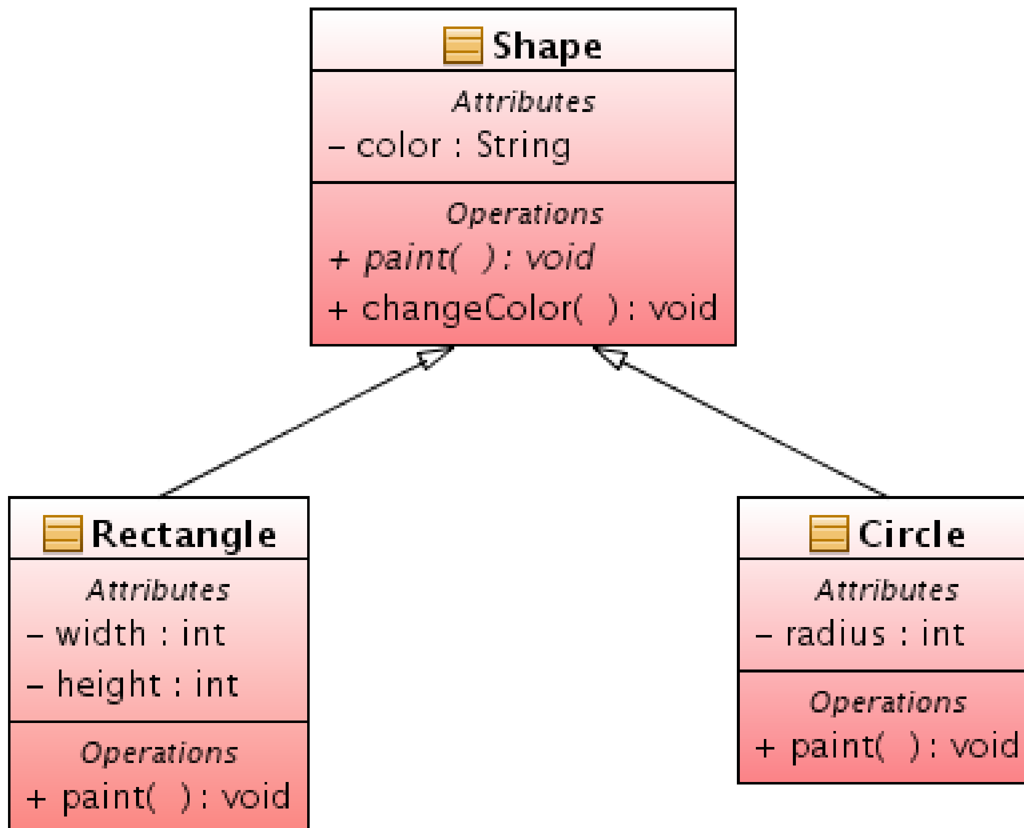
Dědičnost (inheritance) je důležitou vlastností objektově orientovaného programování, protože představuje způsob, kterým jeden objekt získá atributy a chování jiného objektu. Vzniká tak hierarchie tříd, přičemž nejobecnější objekty jsou v nižších stupních hierarchie postupně upřesňovány (a tím také rozmělňovány) do mnoha konkrétnějších tříd. V jazycích Java a C# je povoleno dědit pouze od jedné třídy, zatímco C++ umožnuje dědit více tříd najednou. Zděděný typ je také možné přiřadit do proměnné typu, který je v hierarchii tříd výše.

Příklady

V předchozím příkladu s kočkou a psem jsme použili dědění. Na vrcholu hierarchie byl typ Animal. Typ Dog a Cat pro svoji implementaci použily dědění od třídy Animal. Ve foreach cyklu jsme jednotlivé instance tříd mohli přiřazovat do proměnné typu Animal.

Poznámka: Všechny třídy v jazyce C# i Java dědí z obecného typu Object. Který je na vrcholu hierarchie všech tříd.

V následujícím příkladu třídy Rectangle a Circle dědí od obecné třídy Shape. Rectangle a Circle automaticky zdědí proměnné a metody nadřazeného objektu. Metoda paint() je ve třídě Shape abstraktní, konkrétní implementace je doplněna až na úrovni obou potomků.



Překrývání (overriding)

Zděděné třídy mohou překrýt chování některých metod z bázové třídy. Podmínkou je, aby metody v bázové třídě byly virtuální (Java má metody implicitně virtuální) a měly modifikátor public nebo protected. Některé jazyky pro overriding používají klíčové slovo override. S překrýváním souvisí polymorfismus. Viz příklad na polymorfismus.

Poznámka: Neplést si překrývání s přetížením (overloading) - metody se stejným jménem ale jiným počtem nebo typem argumentů (signatura metody)

Abstraktní třídy a metody

Abstraktní metoda

- Metoda, která je pouze deklarována, ale nemá žádnou implementaci
- Implementace se typicky doplňuje až na úrovni potomka

Abstraktní třída

- Třída, od které nelze vytvářet instance
- Obsahuje obvykle alespoň jednu abstraktní metodu (není podmínkou)

Abstraktní třída vs Interface

C# a Java podporují také tzv. interface. Hlavní rozdíly mezi abstraktní třídou a interfacem popisuje následující tabulka.

Vlastnost	Interface	Abstraktní třída
Mnohonásobná dědičnost	Třída může dědit více interfaců	Třída může zdědit pouze jednu abstraktní třídu
Defaultní implementace	Ne	Ano
Třídní atributy a konstanty	Ano	Ne
Modifikátor přistupu	Všechno public	Může specifikovat přístup

Poznámka: V praxi se velice často interface a abstraktní třída kombinují. Navrhne se interface, abstraktní třída částečně implementuje interface, zbytek doplní pomocí abstraktních metod. Z abstraktní třídy se dále dědí a vytváří konkrétní implementace.

Závěr

Základem objektově orientovaného programování je dědičnost, abstrakce, polymorfismus a zapouzdření.

Zdroje

- http://statnice.stm-wiki.cz/doku.php?id=spolecne:spol23
- Jim Keogh, Mario Giannini, OOP bez předchozích znalostí Průvodce pro samouky, ISBN 80-251-0973-9