Polymorfismus

- Porovnání jazyků z hlediska polymorfismu
- Jazyky C, C++, C#
- Jazyk Java
- PHP a jiné
- Na závěr souhrn vlastností jednotlivých jazyků

Jazyk C

- Jazyk C ve větší míře nepodporuje polymorfismus vůbec
- Absence tříd, pouze struktury
- Pouze přetěžování funkcí – vytváří dojem koerce datových typů
- Silně typovaný jazyk

Jazyk C++

- Implemetuje polymorfismus již velmi intenzivně
- Vyvinut na základě zkušeností s jazykem Simula, C, ADA
- První komerční implementace již roku 1983
- Multiparadigmatický jazyk
- Dědičnost, vícenásobná dědičnost, šablony, přetěžování (i operátorů)

Jazyk C++

- Dědičnost, vícenásobná dědičnost

```
class Employee {
public:
	Employee(const char* szName);
	~Employee();

	unsigned short getAge()    const      { return m_sAge; }
	void setAge(unsigned short sAge)   { m_sAge = sAge; }

private:
	char* m_szName;
	unsigned short m_sAge;
};

class Waiter : public Employee {
public:
	.....
	.....
};
```

Modifikátory dědění: public

```
class Base {
	public: int m_Public;
		Base(int A);
	private: int m_Private;
	protected: int m_Protected;
};

class Derived : public Base {
	public:
	Derived(int B) : Base(B)
	{
		m_Public = 1;		// Tohle lze
		m_Protected = 2;	// Tohle lze
		m_Private     = 3;	// Vyvolá chybu
	}
};
int main()
{
	Derived d;
	d.m_Public = 1;		// Tohle lze
	d.m_Protected = 2;	// Vyvolá chybu
	d.m_Private     = 3;	// Vyvolá chybu
}
```

Modifikátor dědění: public

modifikátor Base	modifikátor Derived	přístup z Derived	přístup z venku
public	**public**	Ano	Ano
protected	**protected**	Ano	Ne
private	**private**	Ne	Ne

Modifikátory dědění: private

```
class Base {
	public: int m_Public;
		Base(int A);
	private: int m_Private;
	protected: int m_Protected;
};

class Derived : private Base {
	public:
	Derived(int B) : Base(B)
	{
		m_Public = 1;		// Tohle lze
		m_Protected = 2;	// Tohle lze
		m_Private     = 3;	// Vyvolá chybu
	}
};
int main()
{
	Derived d;
	d.m_Public = 1;		// Vyvolá chybu
	d.m_Protected = 2;	// Vyvolá chybu
	d.m_Private     = 3;	// Vyvolá chybu
}
```

Modifikátor dědění: private

modifikátor Base	modifikátor Derived	přístup z Derived	přístup z venku
public	**private**	Ano	Ne
protected	**private**	Ano	Ne
private	**private**	Ne	Ne

Modifikátory dědění: protected

```
class Base {
        public: int m_Public;
                Base(int A);
        private: int m_Private;
        protected: int m_Protected;
};

class Derived : protected Base {
        public:
        Derived(int B) : Base(B)
        {
                m_Public = 1;           // Tohle lze
                m_Protected = 2;        // Tohle lze
                m_Private     = 3;      // Vyvolá chybu
        }
};
int main()
{
        Derived d;
        d.m_Public = 1;                 // Vyvolá chybu
        d.m_Protected = 2;              // Vyvolá chybu
        d.m_Private     = 3;            // Vyvolá chybu
}
```

Modifikátor dědění: protected

modifikátor Base	modifikátor Derived	přístup z Derived	přístup z venku
public	**protected**	Ano	Ne
protected	**protected**	Ano	Ne
private	**private**	Ne	Ne

- Jaký je rozdíl oproti dědědí typu *private*?
  Třída, která dědí Derived má stále uvnitř k vlastnostem přístup!

Vícenásobná dědičnost

Class Employee { … };

Class Singer : public Employee { … };
Class Waiter : public Employee { … };

Co se nyní stane když chceme vytvořit zpívajícího číšníka?

Class SingingWaiter : public Singer, public Waiter { … };

SingingWaiter podědí vlastnosti tříd Singer a Waiter, ale jak to bude se třídou Employee?

SingingWaiter bude obsahovat dvě kopie třídy Employee! Každá ze rodičovských tříd ji totiž vnitřně inicializuje.

Vícenásobná dědičnost

Tomuto lze zamezit použitím tzv. virtuální základní třídy:

```
class Singer        :     public virtual Employee { … };
class Waiter        :     virtual public Employee { … };  // Na pořadí nezáleží

class SingingWaiter : public Singer, public Waiter { … };
```

SingingWaiter už nyní obsahuje jen jednu třídu Employee, kterou třídy Singer a Waiter sdílí.

Nejednoznačnosti v případě stejného názvu vlastnosti nebo metody v rodičovských třídách se řeší pomocí operátoru rozlišení.

```
void SingingWaiter::DoSomething()
{
      Singer::DoThings()
      Waiter::DoThings()
}
```

Spřátelené třídy (funkce)

Někdy existuje potřeba, aby jedna třída, která není s druhou třídou společensky spjatá měla přístup k jejím soukromým vlastnostem nebo metodám.

```
//class TvOvladac; - není třeba dopředně deklarovat!
class Televize {
private:
	unsigned int m_uCisloProgramu;
public:
	Televize();

	friend class TvOvladac;
};

class TvOvladac {
Public:
	TvOvladac();
	void PrepniProgram(Televize &Tv, unsigned int uProgram)
		{ Tv.m_uCisloProgramu = uProgram; }
	…
};
```

Virtuální metody

Jedním z jevů polymorfismu v přírodě je schopnost potomků vylepšovat nebo měnit vlastnosti svých předků. Tento aspekt polymorfismu je v jazyce C++ (a mnoha jiných) realizován pomocí tzv. virtuálních metod.

```
Class ZakladniTrida {
    Public:
        Trida();

        virtual void Metoda();
};

Class OdvozenaTrida : publoc ZakladniTrida {
    Public:
        OdvozenaTrida();
        void Metoda();
};
```

Klíčové slovo *virtual* také vynucuje použití tzv. dynamické vazby namísto statické. Umožňuje také skrývání metod vynecháním jejich parametrů v potomkovi.

Abstraktní třídy

Někdy je potřeba definovat jakéhosi kostlivce, který sice nemůže existovat sám o sobě, ale všechny ostatní existence schopné třídy jsou od něj odvozené. Jedná se o tzv. rozhraní popisující pouze jak by měly objekty daného druhu vypadat. Praktickou ukázkou můžou být tzv. Streamy:

```
Class Stream {
Public:
    Stream() {}
    Virtual Int Read(void* pData,unsigned int uSize) = 0;
    Virtual Int Write(void* pData,unsigned int uSize) = 0;
    Virtual Int Tell() = 0; // Použití čistě virtuálních metod => třída Stream je abstraktní
    …
};
Class FileStream : public Stream { … }
Class NetworkStream : public Stream { … }
Class GzipStream : public Stream { … }
```

Samotná abstraktní třída Stream neříká jak se mají data z daného zařízení číst,ale říká jak mají přístupové třídy vypadat.

Šablony v C++

- Šablony jsou přímou realizací myšlenky parametrického polymorfismu
- Jedná se o snahu oddělit algoritmus a datové typy pro větší znovupoužitelnost algoritmů
- V C++ je implementována pomocí klíčového slova template<>
- Poprvé implementováno v jazyku ADA
- Masivní použití šablonových tříd v knihovně STL

Jednoduché šablony

```
template <class T>
class Conteiner {
private:
	const int m_Size = 500;
	T* m_Con[m_Size];
…

};
```

Můžeme tak pracovat s anonymním typem T. Je možné také vytvářet multišablony.

- Často se však stává, že potřebujeme trošku specifikovat implementaci daného algoritmu. Např. třídící algoritmus je odlišný pro číselné prvky a řetězcové prvky.
- K tomuto se používá tzv. specializace (zúžení) šablony
- Od standardu C++11 lze také použít *defaultní specializaci*

Specializace šablon

Mějme šablonu kontejneru, jehož kód je sice obecný, ale pro určitý datový typ se musí kód odlišit.

```
template <class T> class MyArray {
    public:
        void Sort();
        ....
};

template <> class MyArray<char*> {
    public:
        void Sort();
        ...
};
MyArray<int> A;
MyArray<char*> B;
```

Ve druhém případě kompilátor pozná, že má použít specializovanou implementaci a tudíž i jinou třídící metodu.
U multišablon lze provést i tzv. částečnou specializaci.

Jazyk C#

- V otázce polymorfismu jazyk C# místy vylepšuje vlastnosti jazyka C++
- Dědičnost a způsob dědění je stejný jako v jazyce C++
- Jazyk C++ je vhodnější pro programátory, kteří chtějí sáhnout i nízkoúrovňové věci
- Garbage-collecting
- Nepodporuje templatovou specializaci tříd
- Nepodporuje vícenásobné dědění (jen pro rozhraní)

Vlastnosti (GET,SET)

V jazyce C# není nutné implementovat pro danou soukromou vlastnost *get* a *set* metody stylem jaký známe z C++. Můžeme to vidět na následující ukázce:

C++

```
class MyClass {
private:   int m_nMyValue;
public:    int getMyValue() const { return m_nMyValue; }
           void setMyValue(const int nVal) {
                  if (nVal > 10) m_nMyValue = nVal;
                  else m_nMyValue = 0;
      }
};
```

C#

```
class MyClass {
public int MyValue { get;
                     set {
                            if (value > 10) MyValue = value;
                            else  MyValue = 0;
                     }
};
```

Interface

Jazyk C# odděluje pojem *rozhraní* a *abstraktní třídy.*

Interface obsahuje pouze kostru, ale nemůže implementovat žádný kód – je možné dědit vícenásobně od několika *interface*

Abstraktní třída může obsahovat a implementovat kód, avšak stejně jako u *interface* nelze vytvořit její instanci, jen od ní dědit.

```
public interface ISteerable { SteeringWheel wheel { get; set; };  }
public interface IBrakable { BrakePedal brake {get; set; } }
public class Vehicle : ISteerable, IBrakable
{
    public SteeringWheel wheel { get; set; }
    public BrakePedal brake { get; set; }

    public Vehicle() { wheel = new SteeringWheel(); brake = new BrakePedal(); }
}
```

Boxing/Unboxing

Díky této vlastnosti můžeme v C# nahlížet na všechno tak, jakoby bylo odvozené od superobecné třídy object.

- *Boxing* je (implicitní) možnost odkazovat se na specifickou třídu nebo typ pomocí typu *object*
- *Unboxing* je (explicitní) možnost získat z obecného typu *object* libovolný specifický typ.

```
Int A = 123;
object B = A;

int C = (int)B;
```

• V reálné případě však boxing a unboxing umožňuje překlenout typickou mezeru všech jazyků mezi *typem* a *referencí na daný typ*.

Polymorfické modifikátory v C#

- Jazyk C# nemá čistě virtuální metodu, pouze abstraktní třídu. Pokud má být třída abstraktní je třeba jí deklarovat s klíčovým slovem *abstract*.
- Oproti C++ umožňuje C# deklarovat třídu nebo vlastnost, která je terminální – tj. již není možné od ní dále dědit/zdědit jí. K tomu se používá klíčové slovo *sealed*.
- Pokud následní předefinuje virtuální metodu, bylo v jazyce C# kvůli přehlednosti zavedeno klíčové slovo *override*, které musí následník uvést v deklaraci metody. Je tak zřejmé, že tato deklarace přepisuje danou virtuální metodu předka a nedefinuje další.
- V C# také existují dva druhy konstant: *const* a *readonly*. Zatímco *const* definuje konstantu, která i po zdědění zůstává pořád stejná, *readonly* definuje sice konstantu, ale tu si může třída na kterékoliv úrovni dědění zvolit – *readonly* tak může být sice konstanta, ale jiná pro každého následníka.

Polymorfické modifikátory v C#

Oproti C++ přibyl v C# ještě navíc přístupový modifikátor *internal.* Ten umožňuje, aby k vlastnosti nebo metodě mohlo být přistoupeno jen v daném „assembly „– tzn. máme-li třídu, která je součástí frameworku v DLL knihovně.

Vlastnosti a metody v této třídě označené jako *internal* budou viditelné jen uvnitř DLL knihovny, ale ne mimo ní. Program používající framework neuvidí tyto metody a vlastnosti. Klíčové slovo *internal* lze použít i pro třídy – celá třída tak nemusí být viditelná vně frameworku.

Podpora částečných deklarací

Jazyk C# umožňuje „roztrhat“ deklaraci třídy na více míst v kódu pomocí klíčového slova *partial*. Tato vlastnost snadno zhorší čtivost kódu, ale na řadě míst může pomoci.

SouborA.cs

```
partial class Trida {
    private int vlastnostA { get; set; }
    public void MetodaA() { … }
    public Trida();
}
```

SouborB.cs

```
partial class Trida {
    public void MetodaB() { … }
    publiv int vlastnostB() { get; set }
    …
}
```

Jazyk Java

- Java je ve svých jazykových vlastnostech překonávána jazykem C#
- Naopak výhodou Javy oproti většině jazykům je jeho použitelnost na skoro všech myslitelných platformách. V tom se snad Javě může vyrovnat pouze jazyk C nebo C++
- Java podobně jako C# nepodporuje vícenásobnou dědičnost
- Klíčové slovo *super* stejně jako *base* v C# nebo operátor rozlišení v C++

Další rozdíly Javy, C# a C++

- Java podporuje generické typy (šablony), ale nepodporuje jejich specializaci (tzn. ani defaultní)
- Oproti C# nepodporuje částečné deklarace tříd.
- Nepodporuje vlastnosti podobně jako C# - nutné implementovat get a set metody
- Generika v C# a Javě nejsou Turingovsky kompletní (na rozdíl od C++)
- C# a Java používají (package nebo namespace) k logickému členění kódu.

Další podobnosti

- Java stejně jako C# podporuje terminální třídu pomocí klíčového slova *final*
- V jazyce Java je unboxing implicitní na rozdíl od C# kde je explicitní
- V Javě neexistuje slovo virtual, ale pouze jediné abstract. Použití je stejné jako v C#
- Dědění se vynucuje pomocí klíčového slova extends

Jazyk PHP

- Jazyk PHP od verze 3 (lépe od verze 4) podporuje OOP a s ním i základní polymorfické vlastnosti
- Celá implementace objektů byla přepsána ve verzi 5 především s ohledem na výkon
- Podporuje abstraktní třídy, terminální třídy i virtuální metody
- Virtuální metody pouze na principu statické vazby (není tabulka virtuálních metod při běhu)
- Neexistují generika (jedná se o slabě typovaný jazyk

Polymorfismus v PHP

```
class Animal {
    var $name;
    function __construct($name) {
        $this->name = $name;
    }
}

class Dog extends Animal {
    function speak() {
        return "Woof, woof!";
    }
}

class Cat extends Animal {
    function speak() {
        return "Meow...";
    }
}

$animals = array(new Dog('Skip'), new Cat('Snowball'));

foreach($animals as $animal) {
    print $animal->name . " says: " . $animal->speak() . '<br>';
}
```

Trik pro přetížení metody v PHP

```
class myClass {
    public function overloadedMethod() {
        if ( func_num_args() > 1 ) {
            $param1 = func_get_arg(0);
            $param2 = func_get_arg(1);
            $this->_overloadedMethodImplementation2($param1,$param2)
        } else {
            $param1 = func_get_arg(0);
            $this->_overloadedMethodImplementation1($param1)
        }
    }

    protected function _overloadedMethodImplementation1($param1) {
        ...
    }

    protected function _overloadedMethodImplementation2($param1,$param2) {
        ...
    }
}
```

PHP

- PHP nepodporuje přetěžování metod

  Zavedení polymorfismu (ikdyž slabého) do jazyka PHP přineslo spoustu výhod. Díky němu mohla vzniknou velká řada frameworků.
- Jednoduchost jazyka umožnila vytvoření např. platformy HipHop

Jazyky, které se sem nevešly

- Ruby je jazyk používaný pro webové programování, na rozdíl od PHP je těžce objektový, polymorfismus na pokročilé úrovni
- Python podporuje polymorfismus. Generika jsou podporována pomocí *duck-typing,* ale nepodporuje přetěžování
- Perl nepodporuje přetěžování, odvozování tříd v omezené míře
- Pascal má pouze parametrický polymorfismus

Souhrn

- Ze studovaných jazyků je C# je v ohledech polymorfismu nejvyspělejší
- Jazyk Java má své přednosti jinde než v polymorfismu, avšak i ten je na vysoké úrovni
- Jazyk C++ má polymorfismus také na vysoké úrovni a s novým standardem C++11 se může rovnat jazyku C#, oproti oběma zmíněným má výhodu nativního běhu na většině platforem
- PHP je příkladem jazyka s nepříliš vyvinutým polymorfismem, avšak i tato nízká míra jeho implementace otevřela spoustu nových možností

Zdroje

- Stephen Prata: C++ Primer 6th Edition
- Andrei Alexandrescu: Advanced C++

- http://en.wikipedia.org/wiki/Polymorphism
- http://en.wikipedia.org/wiki/Comparison_of_Java_and_C++
- http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java
- http://www.stackoverflow.com
- http://www.cplusplus.com
- http://gcc.gnu.org
- http://msdn.microsoft.com