

Objektově orientované programování polymorfismus

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 1

A0B36PR2 – Programování 2

Část 1 – Organizace předmětu

Informace o předmětu

Přednášky

Cvičení, domácí úkoly a semestrální práce

Hodnocení předmětu a zkouška

Část 2 – Objektově orientované programování v Javě

Objektově orientované programování

Vztahy mezi objekty – dědičnost a polymorfismus

Položky třídy a instance

Konstruktor

Význam metody `main`

Neměnitelné objekty (Immutable objects)

Část 3 – Polymorfismus

Dědičnost

Polymorfismus

Příklad návrhu a využití polymorfismu

Dispatch

Double Dispatch

Část I

Organizace předmětu

Základní zdroje a webové stránky

A0B36PR2 - Programování 2

- Webové stránky předmětu

<https://cw.fel.cvut.cz/wiki/courses/a0b36pr2>

- Odevzdávání domácích úkolů

<https://cw.felk.cvut.cz/upload>

- Přednášející:

- doc. Ing. **Jan Faigl**, Ph.D.



Organizace a hodnocení předmětu

- A0B36PR2 – Programování 2
- Rozsah: 2p+2c
- Zakončení: Z,ZK

Z – zápočet, ZK – zkouška

- Kredity: 6

Po prvním roce studia je nutné získat ≥ 30 kreditů

- Prerekvizity: A0B36PR1 – Programování 1
 - Základy procedurálního a objektově orientovaného programování
 - Přehled o vlastnostech programovacího jazyka Java a virtuálního stroje JVM

Cíle předmětu

Programování 2

- **Prohloubit si** pohled na výpočetní prostředky jako „*počítačový vědec*“ a naučit se je efektivně používat *Computer scientist*
 - Formulovat problém a jeho řešení počítačovým programem
 - Získat povědomí jaké problémy lze výpočetně řešit
- **Získat zkušenost** s programováním *získání vlastní zkušenosti*
 - Programování v jazyku Java a jazyku C *cvičení, domácí úkoly a semestrální práce*
- **Prohloubit si** schopnost číst, psát a porozumět malých programům
- **Osvojit si** schopnost samostatně vytvořit větší programový celek *semestrální práce*
- **Získat** programovací návyky jak psát
 - srozumitelné a přehledné zdrojové kódy;
 - opakovaně použitelné programy.

Zdroje a literatura

- Přednášky – slidy, poznámky a především **vlastní zápisky**
- Cvičení – získání praktických dovedností řešením domácích úkolů a dalších úloh

programovat, programovat, programovat

- On-line kurzy programování

`search for programming in Java | programming in C`

- Knihy Java a jazyk C (C++)

Knihy – Java



Učebnice jazyka Java 5. v., *Pavel Herout* KOPP, 2010, ISBN 978-80-7232-398-2



Introduction to Java Programming, 9th Edition,
Y. Daniel Liang, Prentice Hall, 2012

<http://www.cs.armstrong.edu/liang/intro9e>



An Introduction to Object-Oriented Programming
with Java, 5th Edition, *C. Thomas Wu*,
McGraw-Hill, 2009

<http://it-ebooks.info/book/1908/a>



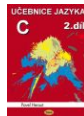
Knihy – Jazyk C



Učebnice jazyka C, VI. vydání, *Pavel Herout*, KOPP, 2010, ISBN 978-80-7232-406-4



Učebnice jazyka C – 2. díl, IV. vydání, *Pavel Herout*, KOPP, 2008, ISBN 978-80-7232-367-8



The C Programming Language, 2nd Edition (ANSI C), *Brian W. Kernighan, Dennis M. Ritchie*, Prentice Hall, 1988 (1st edition – 1978)



The C++ Programming Language, 4th Edition (C++11), *Bjarne Stroustrup*, Addison-Wesley, 2013, ISBN 978-0321563842



Přednášky – letní semestr (LS) akademického roku 2015/2016

- Harmonogram akademického roku 2015/2016

<http://www.fel.cvut.cz/cz/education/harmonogram1516.html>

- Přednášky:

- Dejvice, místnost T2:D3-209, čtvrtek, 9:15–10:45

- 14 výukových týdnů

14 přednášek

- Rektorské volno – středa 11. 5.

Cvičící

- Ing. Petr Váňa
(vedoucí cvičení)



- Ing. Petr Čížek



- Ing. Ondřej Hrstka



- Ing. Martin Balík, Ph.D.
(cvičení na Android)



- Ing. Martin Mudroch, Ph.D.



- Ing. Jakub Mrva



Počítačové laboratoře

- Síťové bootování a síťové domovské adresáře
- Vývoj v Javě: Owncloud – <https://owncloud.cesnet.cz>
 - Prostředí NetBeans 8.0, IntelliJ IDEA, Eclipse a Java verze 8.
 - Sestavení projektu nástrojem **maven** <http://maven.apache.org>
- Vývoj v C:
 - Prostředí NetBeans 8.0 (C/C++) a Eclipse–CDT
CLion – <https://www.jetbrains.com/clion>
 - C/C++ vývojová prostředí Code::Blocks a CodeLite
<http://www.codeblocks.org> a <http://codelite.org>
 - Překladače **gcc** a **clang**
 - Sestavení projektu nástrojem **make** (GNU make)
 - Textový editor – **vim**
<http://www.root.cz/clanky/textovy-editor-vim-jako-ide>
- Odevzdávání domácích úkolů – Upload system
<https://cw.felk.cvut.cz/upload>
- Semestrální práce – repozitář systému pro správu verzí Git
<https://gitlab.fel.cvut.cz>

Domácí úkoly a další úlohy

- Samostatná práce s cílem osvojit si praktické zkušenosti
- Odevzdání domácích úkolů prostřednictvím Upload system

<https://cw.felk.cvut.cz/upload>

- Nahrání (upload) archivů s nezbytnými zdrojovými soubory
- Ověření správnosti implementace automatickými testy

detekce plagiátů

- Úkoly jsou jednoduché a navrhované tak, aby byly stihnutelné
- Klíčem k úspěšnému dokončení předmětu je samostatná práce a osvojení si technik a znalostí

průběžná práce a řešení úkolů

- Pokud něčemu nerozumíte, ptejte se cvičících

pokud možno hned a neodkládejte na později

- Pokud vám přijde úkolů málo, ptejte se po dalších úlohách na procvičování.

Přehled domácích úkolů

- 5 domácích úkolů po 5 bodech

1. (7. týden) - UnitTest pro externí knihovnu (Java)

Odevzdání nejlépe do dalšího týdne

2. (9. týden) - Celulární automaty 1 (Jazyk C)

3. (10. týden) - Permutace (Jazyk C)

4. (11. týden) - Celulární automaty 2 (Jazyk C)

5. (12. týden) - Četnosti (Jazyk C)

- Podmínkou zápočtu je úspěšné odevzdání všech domácích úkolů

- Bodová ztráta za pozdní odevzdání úkolu

Maximální počet bodů za úkol klesá s každým týdnem pozdního odevzdání

Kontrola znalostí testem

- 1 test na přednášce se ziskem maximálně 10 bodů

1. (7. týden - 7.4.2016) – test (~ 60 minut)

Objektově orientované programování a jazyk Java

Čas je orientační a spíše odpovídá očekávané náročnosti testu

Semestrální práce

- Samostatná práce na větším programovém celku
- Volba tématu práce do 2. cvičení *Bodová penalizace -35 bodů.*
- Povinná konzultace semestrální práce do 5. týdne *Bodová penalizace -35 bodů.*
- Odevzdání semestrální práce do **1.5.2016** (10. týden)!
Termín odevzdání v repositáři, následně budou práce postupně hodnoceny
- Maximální počet bodů za semestrální práci 35 bodů
- Podmínkou zápočtu je alespoň 20 bodů ze semestrální práce
- Témata semestrální práce:
 - Témata na stránkách předmětu; vlastní téma možné *Java, ale po dohodě lze i jiný jazyk*
 - Objektový návrh, GUI, netriviální pokrytí unit testy
 - Použití logování (loggeru) a vláken
 - Projekt je v Maven a vývoj musí probíhat v Git (FEL GitLab)
 - Dokumentace v angličtině viz informace na cvičení

<https://cw.fel.cvut.cz/wiki/courses/a0b36pr2/semester-project/start>

Hodnocení předmětu

Zdroj bodů	Maximum bodů	Přípustné minimum bodů
Domácí úkoly (5×5 bodů)	25	15
Test na přednášce	10	0
Semestrální práce	35	20
Písemný zkuškový test	20	10
Implementační zkouška	10	0

- Pro zápočet je minimální počet bodů ze semestru **40**
Cvičící může udělit až 5 bonusových bodů (například za výbornou semestrální práci), nejvýše však do celkového součtu 70 bodů.
- Pro úspěšné absolvování předmětu je nutné získat **zápočet** a vykonat **zkoušku**
- Získání **zápočtu** je podmíněno odevzdáním všech domácích úkolů, úspěšným testem a odevzdáním semestrální práce

Nejpozději ve 14. výukovém týdnu

Klasifikace předmětu

Klasifikace	Bodové rozmezí	Hodnocení	Slovní hodnocení
A	> 90	1	výborně
B	81–90	1,5	velmi dobře
C	71–80	2	dobře
D	61–70	2,5	uspokojivě
E	51–60	3	dostatečně
F	<51	4	nedostatečně

- Minimální přípustné body:

15 (úkoly) + 20 (semestrální práce) + 10 (písemná zkouška) = 45 bodů

Část II

Objektově orientované programování v Javě (připomínka)

Objektově orientované programování (OOP)

OOP je přístup jak správně navrhout strukturu programu tak, aby výsledný program splňoval funkční požadavky a byl dobře udržovatelný.

- **Abstrakce** – koncepty (šablony) organizujeme do tříd, objekty jsou pak instance tříd
- **Zapouzdření** (encapsulation)
 - Objekty mají svůj stav skrytý, poskytují svému okolí **rozhraní**, komunikace s ostatními objekty zasíláním zpráv (volání metod)
- **Dědičnost** (inheritance)
 - Hierarchie tříd (konceptů) se společnými (obecnými) vlastnostmi, které se dále specializují
- **Polymorfismus** (mnohotvárnost)
 - Objekt se stejným rozhraním může zastoupit jiný objekt téhož rozhraní.

Třídy a objekty

Objekty - reprezentují základní entity OO systému za jeho běhu.

- Mají konkrétní vlastnosti a vykazují chování
- V každém okamžiku lze popsat jejich stav
- Objekty se v průběhu běhu programu liší svým vnitřním stavem, který se během vykonávání programu mění

Třídy - popisují možnou množinou objektů. Předloha pro tvorbu objektů třídy. Mají:

- Rozhraní - definuje části objektů dané třídy přístupné zvenčí
- Tělo - implementuje operace rozhraní
- Instanční proměnné - obsahují stav objektu dané třídy

Třídy a objekty - vlastnosti

- **Zapouzdření** (encapsulation) je množina služeb, které objekt nabízí navenek.
Odděluje rozhraní (**interface**) a jeho implementaci.
- **Stav** je určen daty objektu.
- **Chování** je určeno stavem objektu a jeho službami (metodami).
- **Identita** je odlišení od ostatních objektů (v prog. jazycích pojmenování proměnných reprezentující objekty určité třídy).

Třída

Popisuje množinu objektů – je jejich vzorem (předlohou) a definuje:

- **Rozhraní** – části, které jsou přístupné zvenčí
public, protected, private, package
- **Tělo** – implementace operací rozhraní (metod), které určují schopnosti objektů dané třídy
instanční vs statické (třídní) metody
- **Datové položky** – atributy základních i složitějších datových typů a struktur
kompozice objektů
 - Instanční proměnné – určují stav objektu dané třídy
 - Třídní (statické) proměnné – společné všem instancím dané třídy

Struktura objektu

- Objekt je kombinací dat a funkcí, které pracují nad těmito daty
- Objekt je tvořen
 - **Datovými strukturami** – atributy
 - Ovlivňují vlastnosti objektu
 - Jsou to proměnné různých datových typů
 - Data jsou zpravidla přístupná pouze v rámci daného objektu a zvnějšku jsou skryta před jinými objekty

Zapouzdření (encapsulation) / „getter a setter“
 - **Metodami** – funkce / procedury
 - Určují chování objektu
 - Definují operace nad daty objektu
 - Metody představují služby objektu, proto jsou často veřejné

Mohou být deklarovány jako privátní
- **Objekt** je instance třídy
 - V Javě lze vytvářet pouze dynamicky operátorem **new**
 - **Referenční proměnná**

Hodnota proměnné „odkazuje“ na místo v paměti, kde je objekt uložen

Princip zapouzdření (Encapsulation)

- „Utajení“ vnitřního stavu objektu
- Jiné objekty nemohou měnit stav objektu přímo a způsobit tak chybu
- Metody objektu umožňují objektu komunikovat se svým okolím, tvoří jeho **rozhraní**
- Proměnné (data) objektu nejsou z vnějšku objektu přístupné, pro přístup k nim lze využít pouze metody
- Zapouzdření umožňuje udržovat a spravovat každý objekt nezávisle na jiném objektu. Umožňuje **modularitu** zdrojových kódů.

Vztahy mezi objekty

- V OO systému interagují objekty mezi sebou prostřednictvím zasílání zpráv (messages) požadavků na provedení služeb poskytovaných objektem
 1. Po obdržení zprávy objekt vyvolá požadovanou metodu
 2. Případně zašle výsledek
- Objekt poskytující službu se často nazývá *server*
- Objekt žádající o službu se nazývá *klient*
- Mezi objekty je **relace–asociace**, volá-li objekt služby jiného objektu
- S relacemi mezi objekty souvisí viditelnost a vazby mezi objekty

Agregace / Kompozice

Vztah mezi objekty **agregace** reprezentuje vztah typu „je tvořeno/je součástí“ – **has-a**

Příklad

Je-li objekt **A** agregací **B** a **C**, pak objekty **B** a **C** jsou obecně obsaženy v **A**

Hlavním důsledkem je fakt, že **B** ani **C** nemohou přežít bez **A**

V tomto případě hovoříme o kompozici objektů

Příklad implementace

```
class GraphK { //kompozice
    private Edge[] edges;
}

class GraphA { //agregace
    private Edge[] edges;
    public GraphA(Edge[] edges) {
        this.edges = edges;
    }
}

class Edge {
    private Node v1;
    private Node v2;
}

class Node {
    private Data data;
}
```

Dědičnost – Inheritance

- Odvozená třída dědí metody a položky nadtřídy, ale také může přidávat položky nové
 - Můžeme rozšiřovat a specializovat schopnosti třídy
 - Můžeme modifikovat implementaci metod
- Objekt odvozené třídy může „vystupovat“ místo objektu nadtřídy
 - Můžeme například využít efektivnější implementace aniž bychom modifikovali celý program.
- Vztah dědičnosti je také označována jako relace typu **is-a**

Datové položky třídy a instance

■ Datové položky třídy

- Jsou společné všem instancím vytvořeným z jedné třídy
- Nejsou vázaný na konkrétní instanci
- Jsou společné všem instancím třídy
- V Javě jsou uvozeny klíčovým slovem **static**

■ Datové položky instance

- Tvoří vlastní sadu datových položek objektu
- Jsou to tzv. proměnné instance
- Jsou iniciovány při vytvoření instance

V konstruktoru při vytvoření instance voláním `new`

- Existují po celou dobu života instance
- Proměnné jedné instance jsou **nezávislé** na proměnných instance jiné

Metody třídy a instance

■ Metody třídy

- Nejsou volány pro konkrétní instance
- Představují zprávu zaslanou třídě jako celku
- Mohou pracovat pouze s proměnnými třídy

Nikoliv s proměnnými instance

- V Javě jsou uvozeny klíčovým slovem **static**
- Jsou to tzv. statické metody

■ Metody instance

- Jsou volány vždy pro konkrétní instanci třídy
- Představují zprávu zaslanou konkrétní instanci
- Pracují s proměnnými instance i s proměnnými třídy
- Lze volat pouze až po vytvoření konkrétní instance

Přístup ke členům třídy

- Podle principu zapouzdření jsou některé členy třídy označovány jako soukromé (privátní) a jiné jako veřejné.
- Programátor předepisuje k jakým položkám lze přistupovat a modifikovat je
- Přístup ke členům třídy je určen **modifikátorem přístupu**
 - **public:** – přístup z libovolné třídy
 - **private:** – přístup pouze ze třídy, ve které byly deklarovány
 - **protected:** – přístup ze třídy a z odvozených tříd
 - Bez uvedení modifikátoru je přístup povolen v rámci stejného balíčku **package**

Řízení přístup ke členům třídy

Modifikátor	Přístup			
	Třída	Balíček	Odvozená třída	„Svět“
public	✓	✓	✓	✓
protected	✓	✓	✓	X
<i>bez modifikátoru</i>	✓	✓	X	X
private	✓	X	X	X

<http://docs.oracle.com/javase/tutorial/java/java00/accesscontrol.html>

Vytvoření objektu – Konstruktor třídy

- Instance třídy (objekt) vzniká voláním operátoru **new** s argumentem jména třídy, který volá **konstruktor** třídy
- Konstruktor nemá návratový typ, jmenuje se stejně jako třída a můžeme jej přetížit pro různé typy a počty parametrů
- Jiný konstruktor třídy lze volat operátorem **this**

*Operátorem **super** lze volat konstruktor nadřazené třídy*

- Není-li konstruktor předepsán, je vygenerován konstruktor s prázdným seznamem parametrů
 - Je-li konstruktor deklarován, implicitní zaniká
- Přetížení konstruktoru pro různé typy a počty parametrů
- Konstruktor je zpravidla vždy **public** overloading
- Privátní (**private**) konstruktor použijeme například pro:
 - Třídy obsahující pouze statické metody nebo pouze konstanty
Zakážeme tak vytváření instancí.
 - Takzvané singletony (singletons)

Statická metoda `main`

- Deklarace hlavní funkce

```
public static void main(String[] args) { ... }
```

představuje „spouštěč“ programu

- Musí být statická, je volána dříve než se vytvoří objekt
- Třída nemusí obsahovat funkci `main`
 - Taková třída zavádí prostředky, které lze využít v jiných třídách
 - Jedná se tak o „knihovnu“ funkcí a procedur nebo datových položek (konstant)
- Kromě spuštění programu může funkce `main` obsahovat například testování funkčnosti objektu nebo ukázkou použití metod objektu

Neměnitelné objekty (Immutable objects)

■ Definice neměnitelného objektu

- Všechny datové položky jsou **final** a **private**
- V případě objektů jsou odkazy na neměnitelné objekty
- Neimplementujeme „settery“ pro modifikaci položek
- Zákaz přepisu metod v potomcích (**final** modifikátor u metod)

*Nebo jednoduše zákaz odvozování uvedením **final class***

<http://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>

- Objekty, které v průběhu života nemění svůj stav
- Modifikace objektu není možná a je nutné vytvořit objekt nový
- Mají výhodu v případě paralelního běhu více výpočetních toků

<http://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html>

Příklad – **final** nezaručuje neměnitelnost objektu 1/3

- Definujme **final** třídu zapouzdřující referenční **private final** proměnnou **values** typu odkaz na pole **int** hodnot
- Přístup k proměnné **values** je pouze přes *getter* **getArray()**

```
public final class ArrayWrapper {
    private final int[] values;

    public ArrayWrapper(int n) {
        values = new int[n];

        for (int i = 0; i < values.length; i++) {
            values[i] = (int) (Math.random() * n);
        }
    }

    public int[] getArray() {
        return values;
    }

    public String toString() {
        ...
    }
}
```

lec01/ArrayWrapper

Příklad – **final** nezaručuje neměnitelnost objektu 2/3

- Obsah objektu `ArrayWrapper` vypíšeme metodou `toString()`

```
public final class ArrayWrapper {
    ...
    public String toString() {
        ...
        StringBuilder sb = new StringBuilder(values.length >
0 ? new Integer(values[0]).toString() : "empty");
        for (int i = 1; i < values.length; i++) {
            sb.append(" ");
            sb.append(values[i]);
        }
        return sb.toString();
    }
}
```

lec01/ArrayWrapper

Příklad – **final** nezaručuje neměnitelnost objektu 3/3

```
final ArrayWrapper a = new ArrayWrapper(10);
```

```
System.out.println("Random final array '" + a + "'");
```

- Po vytvoření objektu `ArrayWrapper` je obsah pole inicializován v konstruktoru na náhodná čísla

```
Random final array '1 5 5 9 4 9 7 1 8 1'
```

- Přístup k políčkám pole máme přes *getter* `getArray()`

```
final int[] v = a.getArray();
```

```
for (int i = 0; i < v.length; i++) {
```

```
    v[i] = i;
```

```
}
```

```
System.out.println("Final object and final array can be  
still modified\n'" + a + "'");
```

- Původní „final” objekt tak můžeme změnit

```
Final object and final array can be still modified
```

```
'0 1 2 3 4 5 6 7 8 9'
```

`lec01/FinalArrayDemo`

Referenční datové položky neměnitelných objektů musí odkazovat na **neměnitelné objekty**.

Část III

Polymorfismus

Polymorfismus

- Polymorfismus – mnohoznačnost / mnohotvárnost

Vlastnost, která nám umožňuje pojmenovat nějakou konkrétní schopnost (metodu) identickým jménem, přičemž její implementace se může v jednotlivých třídách hierarchie tříd lišit.

- Základním způsobem realizace polymorfismu jsou

- **Dědičnost** (inheritance)
- Virtuální metody – dynamické vázání jména metody ke konkrétnímu objektu
- Rozhraní (*interface*) a abstraktní třídy (*abstract*)
- Překrývání metod (**override**)

Základní vlastnosti dědičnosti

- Dědičnost je mechanismus umožňující
 - Rozšiřovat datové položky tříd nebo je také modifikovat
 - Rozšiřovat nebo modifikovat metody tříd
 - Vytvářet hierarchie tříd
 - „Předávat“ datové položky a metody k rozšíření a úpravě
 - **Specializovat** („upřesňovat“) třídy *protected*
- Mezi hlavní výhody dědění patří:
 - Zásadním způsobem přispívá k znovupoužitelnosti programového kódu
 - **Dědičnost je základem polymorfismu** *Spolu s principem zapouzdření*

Inheritance - dědičnost

Založení definice a implementace jedné třídy na jiné existující třídě

Třída **B** dědí od třídy **A** pak:

- Třída **B** je **podtřídou (subclass)** nebo **odvozenou třídou (derived class)** třídy **A**
- Třída **A** je **nadtřídou (superclass)** nebo **základní třídou (base class)** třídy **B**

Podtřída **B** má obecně dvě části:

- Odvozená část je zděděna od **A**
- Nová **inkrementální část (incremental part)** obsahující definice a kód přidaný třídou **B**

Dědičnost (inheritance), pokračování

- Inheritance je také označována jako relace typu **is-a**
 - Objekt typu **B** je také instancí objektu typu **A**
- Vlastnosti z **A** zděděné v **B** je možné předefinovat:
 - Změna viditelnosti
 - Jiná implementace operací
- Inheritanční relace vytváří objektové hierarchie
 - Funkce podtříd lze soustředit do jejich nadtříd
 - Lze vytvářet abstraktní třídy, ze kterých je možné další třídy vytvářet **specializací**

Polymorfismus

- Polymorfismus (mnohotvárnost) se v OOD projevuje tak, že se můžeme stejným způsobem odvolávat na různé objekty
- Pracujeme s objektem, jehož skutečný obsah je dán okolnostmi až za běhu programu
- **Polymorfismus objektů** - Necht' třída **B** je podtřídou třídy **A**, pak objekt třídy **B** můžeme použít všude tam, kde je očekáván objekt třídy **A**
- **Polymorfismus metod** - Vyžaduje dynamické vázání, statický a dynamický typ třídy
 - Necht' třída **B** je podtřídou třídy **A** a redefinuje metodu $m()$
 - Proměnná x statického typu **B**, dynamický typ může být **A** nebo **B**
 - Jaká metoda se skutečně volá pro $x.m()$ závisí na dynamickém typu

Dědičnost, polymorfismus a virtuální metody

- Vytvoření dynamické vazby je zpravidla v OO programovacím jazyce realizováno virtuální metodou
- Redefinované metody, které jsou označené jako virtuální, mají dynamickou vazbu na konkrétní dynamický typ
- V Javě jsou všechny metody deklarovány jako virtuální; „výjimku“ tvoří
 - statické metody – volány se jménem třídy
 - skryté metody – pragmaticky na ně není přístup
 - metody deklarované s klíčovým slovem **final**

nedovoluje překrývat metody v potomcích

- metody deklarované ve třídě **final**

nedovoluje od třídy odvozovat další třídy

<http://docs.oracle.com/javase/tutorial/java/IandI/final.html>

*V konstruktoru bychom měli volat pouze **final** metody, tak bude objekt inicializován podle zamýšleného způsobu*

Příklad přepsání inicializační metody 1/2

- Ve konstruktoru třídy `BaseClass` voláme metodu `doInit()`

```
public class BaseClass {  
    public BaseClass() {  
        doInit();  
    }  
  
    public void doInit() {  
        System.out.println("Initialization of BaseClass");  
    }  
}
```

lec01/BaseClass

- `doInit()` přepíšeme v odvozené třídě `DerivedClass`

```
public class DerivedClass extends BaseClass {  
    public DerivedClass() {  
        super();  
    }  
  
    @Override  
    public void doInit() {  
        System.out.println("Init. of DerivedClass");  
    }  
}
```

lec01/DerivedClass

Příklad přepsání inicializační metody 2/2

- Po vytvoření objektu třídy `DerivedClass` voláme konstruktor nadřazené třídy `super`
- Vlivem dynamické vazby se však volá implementace `doInit()` třídy `DerivedClass` a nikoliv původní třídy `BaseClass`

```
public class ConstructorDemo {  
    public static void main(String[] args) {  
        System.out.println(  
            "Creating new instance of DerivedClass will  
            not call the initialization of the BaseClass due to  
            overridden doInit() in the DerivedClass");  
        DerivedClass obj = new DerivedClass();  
    }  
}
```

lec01/ConstructorDemo

- Po spuštění se proto vypíše „*Initialization of the DerivedClass*” a nikoliv řetězec uvedený v `BaseClass`
- Proto pokud je nutné zajistit správnou inicializaci nadřazené třídy voláme v konstruktoru pouze **final** metody.

Vytvoření dynamické vazby – dědičnost

- Děděním vytváříme vazbu mezi nadřazenou a odvozenou třídou
- Za běhu programu se můžeme na odvozenou třídu „dívat“ jako na nadřazenou třídu
 - Voláme metody identického jména za běhu je však určena konkrétní instance třídy a je vykonána příslušná implementace
- Vytvoření vazby můžeme provést:
 - Odvozením třídy od nadřazené třídy
 - Odvozením třídy od **abstraktní** třídy
 - Implementací rozhraní (**interface**)
- Příklad volání metody **doStep** objektu reprezentujícího hráče hrající nějakou konkrétní strategii:

```
Player player = new RandomPlayer();  
player.doStep();  
player = new BestPlayer();  
player.doStep();
```

Příklad odvození třídy

- Nadřazená třída

```
public class Player {  
    public void doStep() {  
        // do some default strategy  
    }  
}
```

- Odvozená třída

```
public class RandomPlayer extends Player {  
    public void doStep() {  
        // do a random strategy  
    }  
}
```

Abstraktní třída

- Deklarace třídy se uvozuje klíčovým slovem **abstract**
- Abstraktní třída umožňuje deklarovat abstraktní metody (opět klíčovým slovem **abstract**)
 - Abstraktní metody se mohou vyskytovat pouze v abstraktních třídách, jsou protikladem finálních metod, které nelze předefinovat.*
- Abstraktní metody nemají implementaci a je nutné ji definovat v odvozených třídách
 - Kontrola a podpora objektového návrhu na úrovni jazyka*
- Lze je využít například pro vytvoření společného předka hierarchie tříd, které mají mít společné vlastnosti (bez konkrétní implementace), případně doplněné o datové položky

Příklad odvození od abstrakní třídy

- Nadřazená abstrakní třída

```
public abstract class Player {  
    public abstract void doStep();  
}
```

- Odvozená třída

```
public class RandomPlayer extends Player {  
    @Override  
    public void doStep() {  
        // specific strategy  
    }  
}
```

- Explicitně uvádíme, že metodu přepisujeme
- Lze vytvořit referenční proměnnou abstrakní třídy, ale **vytvořit instanci abstrakní třídy nelze**

Rozhraní třídy – **interface**

- V případě potřeby „dědění“ vlastností více předků lze využít rozhraní **interface**

Řeší vícenásobnou dědičnost

- Rozhraní definuje množinu metod, které třída musí implementovat, pokud implementuje (**implemets**) dané rozhraní
Garantuje, že daná metoda je implementována, neřeší však jak
- Rozhraní poskytuje specifický „pohled“ na objekty dané třídy
Můžeme přetypovat na objekt příslušného rozhraní
- Třída může implementovat více rozhraní
Na rozdíl od dědění, u kterého může dědit pouze od jediného přímého předka
- Případnou „kolizi“ shodných jmen metod více rozhraní řeší programátor

Příklad implementace rozhraní

- Rozhraní

```
public interface Player {  
    public void doStep();  
}
```

- Třída implementující dané rozhraní

```
public class RandomPlayer implements Player {  
    @Override  
    public void doStep() {  
        // specific strategy  
    }  
}
```

Abstraktní třída nebo rozhraní

- **Abstraktní třída** je vhodná pro případy:
 - Odvozené třídy sdílejí implementaci
 - Odvozené třídy vyžadují přístup na položky, které nejsou **public**
- **Rozhraní** je výhodné pokud:
 - Očekáváme, že rozhraní bude implementováno v jiných, nesouvisejících třídách
 - Chceme specifikovat chování konkrétního datového typu (dané jménem rozhraní), bez ohledu na konkrétní implementaci chování
 - Chceme využít vícenásobnou dědičnost

<http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html>

Zadání úlohy – Rámec pro simulaci strategického rozhodování

- Vytvořte simulátor strategické hry (např. sázení–ruleta)
- K simulátoru se může připojit až 5 účastníků hry
- Jeden simulační krok hry lze vyvolat metodou `nextRound`
- Vytvořte tři ukázkové hráče demonstrující použití rámce
 - a Jeden hráč vždy sází na červenou (`PlayerRed`)
 - b Druhý hráč sází náhodně na čísla od 1 do 36 (`PlayerRandom`)
 - c Třetí hráč sází vždy na políčko s nejnižší hodnotou (`PlayerMin`)

Návrh základní struktury

- Rámec se skládá z
 - Herního světa—**World** definující políčka, na která lze vsadit
 - Sázky **Bet** dle pravidel světa
 - Účastníka (**Participant**) hry, který sází na políčka v herním světě
 - Vlastního simulátoru—**Simulator**, který obsahuje svět, hrající hráče a zároveň umožňuje připojení hráčů do hry
- Hráčů (**Player**) hrající strategií a, b nebo c

Kompozice / Agregace

Pro demonstraci použití rámce

Sázka

■ Sázka – **Bet** – na co hráč sází a kolik

*Jednou vyřčená sázka platí a je neměnná – **immutable object***

```
public class Bet {  
    private final String bet;  
    private final int amount;  
  
    public Bet(String bet, int amount) {  
        this.bet = bet;  
        this.amount = amount;  
    }  
  
    public String getBet() { return bet; }  
    public int getAmount() { return amount; }  
  
    @Override  
    public String toString() {  
        return "(" + bet + ", " + amount + ")";  
    }  
}
```

Pro jednoduchost uvažujeme sázku na políčko jako String

Herní svět

- Herní svět **World** definuje políčka a umožňuje účastníkům (**Participant**) položit sázku (**Bet**)

Pro jednoduchost uvažujeme pouze políčka s čísly.

```
public class World {
    private final int MIN_NUMBER = 1;
    private final int MAX_NUMBER = 36;

    public int getMinNumber() {
        return MIN_NUMBER;
    }

    public int getMaxNumber() {
        return MAX_NUMBER;
    }
}
```

- Zapouzdříme rozsah číselných políček

Účastník hry – Participant

- Účastník může být implementován v jiných třídách (někým jiným),
- proto volíme pro účastníka rozhraní **interface**
- S referenční proměnnou typu **Participant** můžeme „pracovat“ v simulátoru aniž bychom znali konkrétní implementaci
- Účastník má v této chvíli pouze jediné definované chování a to vsadit si (sázku **Bet**) – metoda **doStep** pro konkrétní svět

```
public interface Participant {  
    public Bet doStep(World world);  
}
```

- Předáváme referenční proměnnou **World**
 - Hráč se tak může informovat o aktuálním stavu světa

Simulační rámec — Simulator

- **Simulátor** obsahuje svět **World** *(agregace)*
- **Simulátor** obsahuje hráče, ale vytváříme nezávisle mimo simulátor. Hráči se připojují ke hře metodou **join**. *(agregace)*
- Konkrétní implementace hráče je nezávislá, proto agregujeme účastníka hry **Participant**

```
public class Simulator {  
    World world;  
    ArrayList participants;  
    final int MAX_PLAYERS = 5;  
    int round;  
  
    Simulator(World world) {  
        this.world = world;  
        participants = new ArrayList();  
        round = 0;  
    }  
  
    public void join(Participant player) { ... }  
  
    public void nextRound() { ... }  
}
```

Připojení účastníka hry — Simulator – join

- Účastníky hry uložíme v kontejneru `ArrayList`
- Kontrolujeme maximální počet účastníků hry
- a přidáváme pouze nenulového hráče a to pouze jednou (`indexOf`)

```
public void join(Participant player) {  
    if (participants.size() >= MAX_PLAYERS) {  
        throw new RuntimeException("Too many players in the  
            game");  
    }  
    if (player != null && participants.indexOf(player) ==  
        -1) {  
        participants.add(player);  
    }  
}
```

Připojení účastníka hry — Simulator – nextRound

- Rámec odehrání jednoho kola můžeme implementovat i bez známé implementace konkrétního hráče
- Polymorfismus zajistí dynamickou vazbu na konkrétní objekt a volání příslušné metody objektu, který je uložen v seznamu `participants`

```
public void nextRound() {  
    for(int i = 0; i < participants.size(); ++i) {  
        Participant player = (Participant)participants.get(i);  
        Bet bet = player.doStep(world);  
        System.out.println("Round " + round + " player #" + i  
            + "(" + player + ") bet: " + bet);  
    }  
    round++;  
}
```

ArrayList obsahuje referenční proměnné typu Object, proto musím explicitně přetypovat. Tomu se můžeme vyhnout využitím generických typů, viz 2. přednáška.

Hráč – Abstraktní třída **Player**

- Demo hráči mohou sdílet společný kód, např. pro vypsání svého jména,
- proto volíme abstraktní třídu

```
public abstract class Player implements Participant {  
    private final String name;  
  
    public Player(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```

*Jedná se o abstraktní třídu, proto nemusíme explicitně uvádět metodu implementující rozhraní **Participant**, která je automaticky abstraktní.*

- Implementace metody **doStep** je „vynucena“ v odvozených třídách pro dílčí strategie **RandomPlayer**, **RedPlayer** a **MinPlayer**

Ukázka hráčů – RedPlayer

■ RedPlayer

```
public class RedPlayer extends Player {
    public RedPlayer() {
        super("Red");
    }
    @Override
    public Bet doStep(World world) {
        return new Bet("red", 1); //always bet 1
    }
}
```

Ukázka hráčů – RedPlayer a RandomPlayer

■ RandomPlayer

```
public class RandomPlayer extends Player {
    Random rand;
    public RandomPlayer() {
        super("Random");
        rand = new Random();
    }
    @Override
    public Bet doStep(World world) {
        Integer bet = rand.nextInt(36)+1;
        return new Bet(bet.toString(), 1); //bet 1
    } }
```

Ukázka hráče – MinPlayer

■ MinPlayer

```
public class MinPlayer extends Player {  
    public MinPlayer() {  
        super("Min");  
    }  
    @Override  
    public Bet doStep(World world) {  
        Integer bet = world.getMinNumber();  
        return new Bet(bet.toString(), 1); //always bet 1  
    }  
}
```

V tomto případě hráč interaguje se světem

Ukázka použití

```
public class Demo {  
    public static void main(String[] args) {  
        Simulator sim = new Simulator(new World());  
        sim.join(new RandomPlayer());  
        sim.join(new RedPlayer());  
        sim.join(new MinPlayer());  
  
        for(int i = 0; i < 3; ++i) {  
            System.out.println("Round number: " + i);  
            sim.nextRound();  
        }  
    }  
}
```

lec01/Simulator

Polymorfismus a dynamická vazba

- Za běhu programu je vyhodnocen konkrétní objekt a podle toho je volána jeho příslušná metoda
- V příkladu je to metoda **doStep** rozhraní **Participant**
- Zvolený návrh nám umožňuje doplňovat další hráče s různými strategiemi aniž bychom museli modifikovat svět nebo simulátor
- Využitím polymorfismu získáváme modulární a relativně dobře rozšiřitelný (použitelný) rámec
- Uvedené technice se také říká **single dispatch**

Předáváme volání funkce dynamicky (za běhu programu) identifikovanému objektu

Single Dispatch

- Základním principem tohoto návrhového vzoru je dynamická vazba a vyhodnocení typu za běhu programu
- Voláním identické metody `player.doStep()` získáme pokaždé jinou sázku aniž bychom museli identifikovat příslušného hráče

Výhoda dynamické vazby – virtuální funkce

- Relativně komplexního chování jsme dosáhli interakcí více jednoduchých objektů
- Při vykonání kódu je použita dynamická vazba pouze u jednoho objektu
- Je-li volání funkce závislé na více za běhu detekovaných objektech, hovoříme o **multi dispatch**
- V případě dvou objektů se jedná o **double dispatch**

Příklad rozšíření – Přidání políčka s hodnotou nula

- Přidání políčka s hodnotou 0 realizujeme vytvořením nové třídy **WorldZero**, která rozšiřuje původní svět **World**

```
public class WorldZero extends World {  
    private final int MIN_NUMBER = 0;  
  
    public int getMinNumber() {  
        return MIN_NUMBER;  
    }  
}
```

- Nový svět stačí předat simulátoru v konstruktoru
`Simulator sim = new Simulator(new WorldZero());`

- Zbytek programu zůstává identický

Příklad: `lec01/Simulator`

- Jak definovat nový svět s novými vlastnostmi aniž bychom museli modifikovat kompletně celý program?

Řešení je použít návrhový vzor **double dispatch**

Double Dispatch

- Principem **double dispatch** je vyhodnocení dvou objektů za běhu programu a automatická volba volání odpovídající funkce
- Podobného efektu lze dosáhnout použitím **instanceof** pro detekci příslušného typu objektu a explicitním voláním příslušné třídy
- Vzor double dispatch je však elegantnější a jednodušší

Příklad nového světa s novými vlastnostmi

- Nejdříve musíme zajistit identifikaci objektu světa za běhu
- Do světa proto přidáme metodu, ze které budeme volat **doStep** konkrétního hráče

```
public class World {  
    ...  
    Bet doStep(Participant player) {  
        return player.doStep(this);  
    }  
}
```

Tak zajistíme identifikaci konkrétní implementace světa

- Metodu pojmenujeme například **doStep**
- Ve třídě **Simulator** upravíme volání `player.doStep(world)` na `world.doStep(player)`
- Tím zajistíme, že se nejdříve dynamicky identifikuje typ objektu referenční proměnné **world** a následně pak typ objektu v referenční proměnné **player**

Program nyní funguje jako předtím, navíc nám však umožňuje rozšířit simulátor o novou implementaci světa

Nový svět – WorldNew

```
public class WorldNew extends World {  
    private final String[] fields;  
  
    public WorldNew() {  
        super();  
        fields = new String[36 + 1 + 4];  
        fields[0] = "even";  
        fields[1] = "odd";  
        fields[2] = "red";  
        fields[3] = "black";  
        for (int i = 0; i <= 36; ++i) {  
            fields[i + 4] = Integer.toString(i);  
        }  
    }  
  
    Bet doStep(Participant player) { // we need to link  
        return player.doStep(this); // doStep with this  
    }  
  
    public String[] getFields() { //new method  
        return fields;  
    }  
}
```

Rozšíření účastníka a existujících hráčů

- Účastníka hry **Participant** musíme rozšířit o uvažování nového světa

```
public interface Participant {  
    public Bet doStep(World world);  
    public Bet doStep(WorldNew world);  
}
```

- Implementaci původních hráčů provedeme v abstraktní třídě **Player**

```
public abstract class Player implements Participant {  
    ...  
    public Bet doStep(WorldNew world) {  
        return doStep((World)world); //default behaviour  
    }  
}
```

*Chování původních hráčů v novém světě neřešíme, proto s výhodou modifikujeme pouze abstraktní třídu **Player**.*

Nový hráč pro nový svět – PlayerNew

```
import java.util.Random;

public class PlayerNew extends Player {
    Random rand;
    public PlayerNew() {
        super("New player");
        rand = new Random();
    }
    @Override
    public Bet doStep(World world) {
        // strategy for standard world
        return new Bet("black", 1); //always bet 1 gold
    }
    @Override
    public Bet doStep(WorldNew world) {
        // strategy for the new world
        // random choice even or odd
        return new Bet(world.getFields()[rand.nextInt(2)], 1);
    }
}
```

Nový hráč má jiné chování v původním a novém světě.

Použití nového hráče v novém světě – Demo

```
public class Demo {  
  
    public static void main(String[] args) {  
        Simulator sim = new Simulator(new WorldNew());  
        sim.join(new RandomPlayer());  
        sim.join(new RedPlayer());  
        sim.join(new MinPlayer());  
        sim.join(new PlayerNew());  
  
        for (int i = 0; i < 3; ++i) {  
            System.out.println("Round number: " + i);  
            sim.nextRound();  
        }  
    }  
}
```

lec01/SimulatorDD

- Pouze rozšíření světa nestačí, je nutné realizovat dynamickou vazbu
- Svět a hráče můžeme nyní rozšiřovat, aniž bychom museli zasahovat do simulačního rámeč třídy **Simulator**

Přetížení metod „overloading” a přepsání metod „overriding”

- Přetížení metody je volba konkrétní implementace na základě typu a počtu parametrů.
- Přetížení je statická vazba a děje se při kompilaci programu
- Volání přepsané metody je vazba dynamická a děje se za běhu programu.
- Identifikovat objekt můžeme také sami operátorem **instanceof**
- Double dispatch obsahuje volání funkce navíc, ale ta je velmi krátká a tak je zpravidla „inlinována” za běhu

Při načtení programu i za běhu jsou prováděny optimalizace a krátké funkce tak mohou být přímo vloženy do kódu. Odpadá tak režie související s voláním a uložením „program counter” / „instruction pointer”

Shrnutí přednášky

Diskutovaná témata

- Informace o předmětu
- Přehled objektově orientovaného programování v Javě
- Polymorfismus – příklad
- Využití polymorfismu a návrhový vzor Double dispatch
- Příště: Výjimky, výčtové typy a kontejnery v Javě

Opakování z PR1