Údajové typy

V Jave rozoznávame údajové typy dvojakého druhu – **primitívne** a **referenčné**.

Premennú deklarujeme uvedením názvu typu, za ktorým nasleduje názov premennej alebo zoznam viacerých premenných oddelený čiarkami. Operátor = slúži na priradenie hodnoty premennej a môže byť uvedený priamo za deklaráciou premennej, kedy celú konštrukciu nazývame definíciou premennej.

Príklad:

```
int x;     // deklarácia premennej x
int y, z;  // deklarácia premenných y, z
int v = 5; // deklarácia a priradenie hodnoty (definícia) premennej v
```

Primitívne údajové typy

Reprezentujú elementárne údaje, je ich obmedzené množstvo a nedajú sa používateľsky definovať.

Prehľad primitívnych údajových typov

typ	stručný opis	veľkosť	najmenšia hodnota	najväčšia hodnota
`byte`	celé číslo	8 bitov	–128	+127
`short`	celé číslo	16 bitov	–32 768	+32 767
`int`	celé číslo	32 bitov	-2^{31}	$+2^{31} - 1$
`long`	celé číslo	64 bitov	-2^{63}	$+2^{63} - 1$
`float`	reálne číslo	32 bitov	v súlade so štandardom IEEE 754 – vysvetlenie je nad rámec tohto dokumentu	
`double`	reálne číslo	64 bitov		
`boolean`	logická (booleovská) hodnota	*«nedefinované»* (teoreticky jeden bit, ale reálne to je viac)	`false`	`true`
`char`	znak Unicode	16 bitov	`'\u0000'` (alebo 0)	`'\uffff'` (alebo 65 535)

Celočíselné typy

Na údajový typ `char` sa tiež môžeme pozerať ako na číslo. V takom prípade však nadobúda len kladné hodnoty. Základný celočíselný typ je `int`, rozšírený `long` a zúžené sú `short` a `byte`.

Literál typu `long` – sa dá explicitne zapísať tak, že za číslo pridáme znak `l` alebo `L`, napríklad `123456L`.

Hodnoty celočíselných údajových typov (`byte`, `short`, `int` a `long`) môžu byť zapísané v desiatkovej, osmičkovej alebo šestnástkovej (hexadecimálnej) číselnej sústave.

Ak sa číselný literál začína číslicou `0`, znamená to, že je zapísaný v osmičkovej sústave. Ak sa číselný literál začína znakmi `0x`, znamená to, že je zapísaný v šestnástkovej číselnej sústave.

```
int x = 123;  // x = 123
int y = 0123; // y = 83
int z = 0x12; // z = 298
```

Literály

Použitie literálov je rôzne. Používajú sa všade tam, kde potrebujeme do zdrojového kódu zapísať hodnotu priamo. **Literály** sú priamou reprezentáciou hodnoty v zdrojovom kóde (bez ďalších výpočtov). Literál je reprezentácia trvalej hodnoty v zdrojovom kóde.

Literál je možné použiť pri ukladaní konkrétnej hodnoty do premennej primitívneho údajového typu. Z toho vyplýva, že pri inicializácii hodnoty primitívneho údajového typu sa nepoužíva operátor `new`, pretože primitívne typy sú špeciálne údajové typy zabudované do jazyka – nejde o objekty (inštancie) nejakej triedy (ale môže z neho vzniknúť – ide o tzv. autoboxing).

```
boolean výsledok = true;
char  veľkéC = 'C';
byte  b = 100;
short s = 10000;
int   i = 100000;
```

Reálne typy

Na vyjadrenie reálnych čísiel sa v Jave používajú dva typy – `float` a `double`. Sú to typy s tzv. pohyblivou rádovou čiarkou (angl. floating point types – v doslovnom preklade typy s „plávajúcou desatinnou čiarkou“).

Môžeme ich v zdrojovom kóde explicitne označiť pridaním písmena `f` alebo `F` resp. `d` alebo `D` na koniec číselného literálu. Desiatkový exponent vyjadrujeme s pomocou znaku `e` alebo `E` – príklad: `1.23e5 = 12300`.

Reálne typy môžu okrem platných číselných hodnôt nadobúdať aj tieto špeciálne hodnoty:

- `POSITIVE_INFINITY` – kladné nekonečno,
- `NEGATIVE_INFINITY` – záporné nekonečno
- a `NaN` – hodnota nie je číslo – **N**ot **a N**umber.

Znakové typy

`char` je dvojbajtový typ na reprezentáciu znaku podľa štandardu Unicode. Znakový literál vyjadrujeme s pomocou apostrofov: `'a'`. (To znamená, že jeden znak uzavretý do úvodzoviek (`"a"`) reprezentuje jednoznakový reťazec, a nie znak.)

Znakový reťazec už nie je primitívny údajový typ, reprezentujú ho inštancie triedy `java.lang.String`.

Logický typ

`boolean` – môže nadobúdať hodnoty: `true` (pravda) a `false` (lož).

Prázdny typ

`void` – znamená „bez hodnoty“ a ide o prázdny údajový typ. V Jave nemôžu byť definované premenné typu `void`, ale metódy môžu mať návratový typ `void`. Také metódy nevracajú žiadnu hodnotu. (V Jazyku Pascal by sme hovorili o procedúrach.)

Typová konverzia

Celočíselné typy môžeme priamo previesť iba na „presnejšie hodnoty“ – môžeme previesť údajový typ z nižšou presnosťou (alebo rozsahom) na údajový typ s vyššou presnosťou (alebo rozsahom), aby nemohla nastať strata hodnoty.

Ak chceme postupovať opačným smerom, tak je nevyhnutné použiť tzv. **pretypovanie**. Slúži na to operátor pretypovania, ktorý zapisujeme ako cieľový údajový typ uzavretý do okrúhlych zátvoriek. Treba

ho použiť napríklad pri prevode z typu `long` na `int`, z `double` na `float`, ale aj pri konverzii hodnôt z reálnych premenných na celočíselné.

```
int i = 5;
short s = 5;
float f = 12.3f;

i = s;  // bude preložené bez chyby
s = i;  // nebude preložené bez chyby – kompilátor to nedovolí,
        // lebo by sa pri priradení mohla stratiť časť hodnoty
s = (short)i;  // «nie»
f = i;         // «áno»
i = f;         // «nie»
i = (int)f;    // «áno»
```

V Jave nie je možné hodnoty typu `boolean` reprezentovať celým číslom, ako napríklad v jazykoch C a C ++, a tiež nie je možné voľne zamieňať celočíselné typy s typom `boolean`. Ak chceme vykonať napríklad konverziu medzi typmi `int` a `boolean` – musíme použiť určitú operáciu, napríklad nasledujúcu konštrukciu:

```
boolean b = (i != 0);  // 0 na false, zvyšok na true
int i = (b ? 1 : 0);   // true na 1, false na 0
```

Príklad literálu	Údajový typ		Príklad literálu	Údajový typ
178	int		8864L	long
37.66	double		37.66D	double
87.56F	float		26.57e	double
'c'	char		true	boolean

Referenčné údajové typy

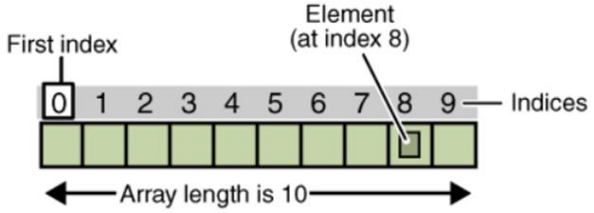
Referenčné údajové typy sú objekty a polia. Hodnota referenčnej premennej je odkaz (referencia) do pamäte na miesto, kde je objekt (alebo pole) uložený. V iných programovacích jazykoch sa používa na tento účel pointer (ukazovateľ pamäte), v Jave sa priamo s pamäťou nepracuje, namiesto pointerov sa používajú referenčné premenné.

`null` – je prázdna hodnota pre referenčné typy a znamená, že chýba odkaz na objekt resp. pole

Keďže referenčná premenná obsahuje iba odkaz na objekt, nie objekt samotný, priradením jej hodnoty inej premennej sa priradí opäť len referencia na pôvodný objekt, nie jeho samotné údaje tak, ako to bolo pri primitívnych údajových typoch.

Pole

Objekt poľa je kontajner, ktorý ukladá vopred pevne stanovený počet položiek rovnakého typu. To znamená, že dĺžka (veľkosť; počet prvkov) poľa je vopred určená a to v čase vytvorenia poľa.



Každá položka v poli sa nazýva element alebo prvok a každý prvok je dostupný prostredníctvom jeho číselného indexu (celočíselného, dokonca na indexovanie nie sú povolené hodnoty typu `long`, ale najviac `int`; nasledujúci pokus o inicializáciu poľa reťazcov by zlyhal: `String s[] = new String[10L];` pretože

`10L` je literál typu `long`). Číslovanie indexov sa začína od hodnoty `0`, to znamená, že deviaty element je dostupný pod indexom `8`.

Deklarácia premennej typu pole

```
int[] nejakéPole;       // deklarácia poľa celých čísiel typu int
byte[] nejakéPole;      //   tiež, ale typu byte
short[] nejakéPole;     //   tiež, ale typu short
long[] nejakéPole;      //   tiež, ale typu long
float[] nejakéPole;     // deklarácia poľa reálnych čísiel typu float
double[] nejakéPole;    //   tiež, ale typu double
boolean[] nejakéPole;   // deklarácia poľa logických hodnôt typu boolean
char[] nejakéPole;      // deklarácia poľa znakov
String[] nejakéPole;    // deklarácia poľa reťazcov
```

Tiež môžeme použiť hranaté zátvorky za názvom poľa, napríklad:

```
float poleReálnychČísiel[];
```

Vytvorenie, inicializácia poľa

Pole je v Jave vnímané tiež ako objekt, preto na inicializáciu poľa musíme použiť operátor `new`. Desaťprvkové celočíselné pole do premennej `nejakéPole` s pomocou neho inicializujeme takto:

```
nejakéPole = new int[10];
```

Zdroje

- «*neuvedené*»

Uzavretosť a dedičnosť

Dedičnosť

Pod dedičnosťou rozumieme odvodzovanie (v Jave používame kľúčové slovo `extends` – rozširuje) nových (objektových) tried od skôr deklarovaných tried. Odvodené triedy dedia všetky pôvodné atribúty a metódy, ktoré môžu modifikovať a pridávať k nim nové. Napríklad majme definovanú triedu `Zviera`, ktorá má rad atribútov a metód. Z nej odvoďme triedu `Cicavec` s ďalšími vlastnosťami typickými pre cicavca – napríklad čas dojčenia.

Dedičnosť dovoľuje vybudovanie hierarchie tried, ktoré sa postupne z generácie na generáciu rozširujú. Používa sa v prípadoch, keď sa chceme vyhnúť opakovanému písaniu rovnakého alebo podobného kódu. Je to dôležitý nástroj na vytváranie opakovane využiteľných programových modulov.

Programový modul (balíček; skupina tried) môže byť uzavretý alebo otvorený. Uzavretý – na jeho použitie nie je potrebné nič pridávať, používateľ nie je oprávnený modul (balíček, angl. `package`) modifikovať. Otvorený – používateľ môže nevhodné veci modifikovať a nové pridávať.

Dve základné výhody dedičnosti:

- má praktický význam na znovupoužiteľnosť programového kódu,
- je základom viacerých druhov polymorfizmu.

Pri dedení odvodenej triedy od rodičovskej môžeme to, čo bolo v základnej (rodičovskej) triede:

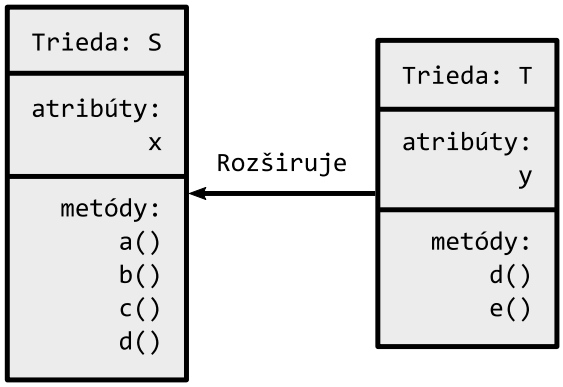
- dobré ponechať,

- chýbalo dodať,
- zlé (nevhodné alebo sa nám nepáčilo) zmeniť.

Realizácia dedičnosti v Jave

Musí jestvovať trieda, ktorá sa stane rodičom (nadradená trieda, angl. superclass). Meno tejto triedy sa uvedie v hlavičke triedy za kľúčovým slovom `extends`. V Jave je povolená len jednoduchá dedičnosť. To znamená, že každá trieda smie byť odvodená len od jednej rodičovskej triedy. Viacnásobnú dedičnosť môžeme dosiahnuť (napodobniť) prostredníctvom takzvaných rozhraní (angl. interfaces).

Spresnenie: Rozhrania sú v tomto dokumente spomínané aj pri abstrakcii, no ani tam, ani tu nie sú podrobne vysvetlené, ani k nim nie je uvedený žiadny príklad (príklady sú na iných miestach – hľadajte kľúčové slovo `implements`). Rozhranie deklarujeme podobne ako triedu, ale namiesto kľúčového slova `class` uvedieme `interface`. Do verzie Javy 8 bolo povolené do rozhrania uvádzať len deklarácie metód bez ich implementácie a definície konštánt. Rozhranie samé o sebe nie je použiteľné, spôsob jeho použitia tkvie v implementácii, čo je ekvivalent odvodzovania, ale namiesto kľúčového slova `extends` musíme uviesť `implements`. Za kľúčové slovo `implements` môžeme uviesť aj viacero rozhraní oddelených čiarkami. Jedna trieda Javy môže byť súčasne odvodená od jednej inej triedy a od viacerých rozhraní.



Objekt novej (odvodenej) triedy `T` obsahuje:

- atribúty `x` a `y` (jeden zdedený, jeden nový),
- zdedené metódy `a()`, `b()`, `c()`,
- novú metódu `d()`
- a prekrýva starú metódu `d()` rodičovskej triedy `S`.

Vlastnosti metód v odvodenej triede môžeme zmeniť dvojakým spôsobom:

- **Preťažením (overloading)** – použijeme rovnaké meno metódy, ale iné parametre, prípadne návratový typ.
- **Prekrytím (overriding, zasinenie – hiding)** – hlavička metódy je identická s hlavičkou metódy rodiča, ale metóda môže robiť niečo iné.

Pri využití dedičnosti počas konštrukcie musí byť umožnené volať konštruktor rodiča. Môžu nastať dva prípady:

- Rodič má konštruktor bez parametrov alebo implicitný konštruktor: potomok môže mať implicitný konštruktor a nemusí volať konštruktor rodiča.
- Rodič má konštruktor aspoň s jedným parametrom: konštruktor potomka musí jestvovať a prvým príkazom v jeho tele musí byť volanie konštruktora rodiča (príkaz `super()`).

```
class Rodič
```

```
{
    public int i;
    public Rodič(int novéI) { i = novéI; }
    // public Rodič() { i = 5; }
}

public class Potomok extends Rodič
{
    public Potomok()
    {
        super(8);
    }

    public static void main(String[] args)
    {
        Potomok potomok = new Potomok();
        System.out.println("Hodnota i: " + potomok.i);
    }
}
```

Po spustení:

```
Hodnota i: 8
```

Finálne metódy triedy

- Používajú sa vtedy, keď považujeme metódu za dokonalú a nechceme, aby bola v zdedených triedach prekrytá,
- táto metóda *nemôže* byť prekrytá, ale *môže* byť preťažená (pozri preťažovanie v kapitole o polymorfizme).

```
class Rodič
{
    public int i;
    public Rodič() { i = 1; }
    final int dajI() { return i; }  // Finálna metóda – nedá sa prekryť
}

public class Potomok extends Rodič
{
    // int dajI() { return i * 2; } // chyba
    public static void main(String[] args)
    {
        Potomok potomok = new Potomok();
        System.out.println("Hodnota je: " + potomok.dajI());
    }
}
```

Po spustení:

```
Hodnota je: 1
```

Abstraktné metódy triedy

- Ak chceme, aby bola rodičovská metóda určite prekrytá, uvedieme v jej hlavičke v rodičovskej triede kľúčové slovo (modifikátor) `abstract`.

Poznámky:

- Ak je jedna z metód triedy označená kľúčovým slovom (modifikátorom) `abstract`, tak musí byť tento modifikátor (`abstract`) použitý aj na celú triedu, ktorá sa tým stáva **abstraktnou triedou**,
- v zdedenej triede musíme implementovať (naprogramovať) všetky metódy označené ako abstraktné.

```
abstract class Rodič
{
    public int i; public Rodič() { i = 1; }
    abstract int dajI();
    final void nastavI(int novel) { i = novel; }
}

public class Potomok extends Rodič
{
    int dajI() { return i * 2; }
    void nastavI() { i = 5; } // preťaženie

    public static void main(String[] args)
    {
        // Rodič rodič = new Rodič(); // chyba
        Potomok potomok = new Potomok();
        potomok.nastavI(3);

        System.out.println("Hodnota je: " + potomok.dajI());
        potomok.nastavI(); // preťažená
        System.out.println("Hodnota je: " + potomok.dajI());
    }
}
```

Po spustení:

```
Hodnota je: 6
Hodnota je: 10
```

Zdroje

- https://cw.fel.cvut.cz/old/_media/courses/a0b36pr1/lectures/09/36pr1-09_tridyii.pdf
- http://docplayer.cz/21457756-Dedicnost-polymorfismus-interface-prace-se-soubory.html
- http://slideplayer.cz/slide/3198349/

Abstrakcia

Pod **abstrakciou** sa vo všeobecnosti chápe zameranie sa na kľúčové vlastnosti nejakého prvku reálneho sveta (alebo aj nereálneho).

Pri programovaní môže programátor (v programe, ktorý vytvára) abstrahovať od niektorých detailov práce jednotlivých objektov. Každý objekt pracuje ako čierna skrinka, ktorá dokáže vykonávať určené činnosti a komunikovať s okolím bez toho, aby vopred poznala spôsob, akým je táto funkcionalita implementovaná (tá sa totiž implementuje až dodatočne v odvodených triedach).

V OOP (v Jave) to zužitkujeme hlavne pri abstraktných triedach.


```
abstract class Útvar
{
    java.awt.Color farba;
    abstract void nakresli();
}
```

```
class Trojuholník extends Útvar
{
    void nakresli()
    {
        // ……
    }
}
```

```
class Kruh extends Útvar
{
    void nakresli()
    {
        // ……
    }
}
```

```
class Štvorec extends Útvar
{
    void nakresli()
    {
        // ……
    }
}
```

Abstraktná trieda `Útvar` určuje, že všetky od nej odvodené triedy sa budú dať nakresliť a budú mať nejakú farbu. Nemôžeme vytvoriť objekt (inštanciu) abstraktnej triedy. To dáva zmysel aj v súvislosti s uvedeným príkladom, lebo všeobecný „útvar“ nemá žiadnu konkrétnu podobu – „nevie“, ako má byť nakreslený.

Vytvoriť sa dajú len inštancie odvodených tried, ktoré v tomto prípade implementujú kresliacu metódu (`nakresli()`). Triedy môžu byť aj poloabstraktné v tom zmysle, že môžu implementovať niektoré metódy a ostatné nechať neimplementované. Podobnou záležitosťou ako abstraktné triedy, sú v jazyku Java aj rozhrania (angl. interfaces). Tie obsahujú iba deklarácia metódy, čiže žiadna z nich nie je implementovaná.

(**Poznámka:** Od verzie Java 8 sú povolené tzv. predvolené (`default`) implementácie metód rozhraní. Tie dovoľujú implementovať metódy aj v rozhraniach. Tento koncept však bol do jazyka pridaný ako dôsledok jeho rozširovania o možnosti funkcionálnej programovacej paradigmy.)

Zdroje

- «*neuvedené*»

Posielanie správ

1 Správy

Objekty na seba vzájomne pôsobia a komunikujú – vzájomne interagujú (angl. interact) prostredníctvom správ (angl. messages). Interakciu objektov si môžeme vysvetliť ako využitie princípu architektúry klient – server. Klient poslaním správy žiada server o nejakú službu. [7]

(**Poznámka:** Toto prirovnanie vyžaduje, aby čitateľ vopred chápal princíp architektúry klient – server. Najjednoduchšie (zjednodušene, ale rýchlo) sa táto architektúra dá vysvetliť na prehliadaní webových stránok. Keď zadáte webovú adresu do prehliadača a potvrdíte (tlačidlom alebo Enterom), odošlete tým *«správu»* z webového prehliadača – Vášho *«klienta»* na webový *«server»*. *«Server»* Vám odpovie odoslaním webovej stránky (v skutočnosti je toho viac, ale Vy uvidíte len tú stránku), čo je v podstate tiež *«správa»*, len trochu dlhšia.)

Pokračovanie: A server mu vyhovie alebo nevyhovie – podľa okolností, ako sú právomoci a podobne.

Server môže vyhovieť klientovi rôzne, môže vykonať určitú akciu, ktorú potvrdí odoslaním správy klientovi, môže len odpovedať na konkrétnu jednoduchú požiadavku správou (to však nie je časté, lebo väčšinou od servera požadujeme určitý výpočtový výkon, na to architektúra klient – server vznikla). Pri OOP však posielanie správ medzi objektmi môže mať celkom bežne za následok relatívne jednoduchú odozvu… (Nezabúdajme, že architektúru klient – server sme použili len ako prirovnanie a prirovnania nie sú nikdy úplne dokonalé.)

Jediný objekt nie je sám o sebe veľmi užitočný. Zvyčajne vystupuje ako komponent väčšieho programu alebo aplikácie, ktorá pozostáva z viacerých (mnohých) objektov. Programátori prostredníctvom interakcie objektov dosiahnu vysokú funkčnosť a viacúrovňové komplexné správanie. [7]

2 Odoslanie správ

V informatike je posielanie správ technikou na vyvolanie určitého správania v počítači (napríklad spustenie podprogramu). Spúšťajúci (volajúci) program (v Jave metóda) odošle správu ďalšiemu procesu a spolieha na neho a jeho podpornú infraštruktúru pri výbere a spustení určitého kódu.

Posielanie správ sa líši od konvenčného programovania, kde sa proces, podprogram alebo iná funkcia priamo zavolá menom. Posielanie správ je kľúčom k niektorým modelom súbežnosti a objektovo orientovaného programovania. Odovzdávanie správ môže byť implementované rôznymi mechanizmami vrátane kanálov.

Skladá sa zo **zapuzdrenia** a **distribúcie**.

Zapuzdrenie je myšlienka, že softvérové objekty by mali byť schopné zavolať (spustiť) služby na iných objektoch bez toho, aby vedeli alebo sa starali o to, ako sú tieto služby implementované. Zapuzdrenie môže znížiť množstvo kódovacej logiky a urobiť systémy udržateľnejšie.

Distribuované odovzdávanie správ poskytuje vývojárom vrstvu architektúry, ktorá poskytuje bežné služby na vytváranie systémov pozostávajúcich z podsystémov, ktoré fungujú na rozdielnych počítačoch, na rôznych miestach a v rôznych časoch. Keď distribuovaný objekt odosiela správu, vrstva správ môže zaobchádzať s problémami ako:

- Vyhľadanie aplikácie s pomocou rôznych operačných systémov a programovacích jazykov na rôznych miestach, odkiaľ pochádza správa (pravdepodobne myslené kam správa smeruje).
- Uloženie správy do frontu (skôr zásobníka, front mi znie hovorovo), ak v súčasnosti nie je spustený príslušný objekt na spracovanie správy a potom zavolanie správy (pravdepodobne odoslanie správy), keď je objekt dostupný. Tiež uloženie výsledku v prípade potreby, kým je odoslaný objekt pripravený na jeho prijatie. (Trošku ťažšie zrozumiteľné, keď bude čas, sa k tomu vrátim.)
- Kontrola rôznych transakčných požiadaviek na distribuované transakcie, napríklad ACID-testovanie údajov.

Zdroje

- «*neuvedené*»

Polymorfizmus

Úvod

Polymorfizmus znamená viactvarosť, pričom to, čo môže nadobúdať „viac tvarov“ (podôb, foriem) závisí od druhu polymorfizmu. V rôznych zdrojoch sa dajú nájsť rozmanité informácie o odlišných druhoch a rozdielnych rozdeleniach polymorfizmu, ale v tomto materiáli ponechávame len tie informácie, ktoré sa podarilo nájsť autorke, ktorá túto tému vypracúvala.

Je to jeden z hlavných konceptov OOP. Opisuje koncepciu, že objekty rôznych typov môžu byť prístupné prostredníctvom toho istého rozhrania. Každý typ môže poskytnúť svoje vlastné a nezávislé vykonávanie tohto rozhrania.

(To je len jeden z druhov polymorfizmu.)

Keďže všetky triedy v Jave sú predvolene odvodené od triedy `Object`, tak vlastne všetky objekty v Jave sú **polymorfné**, pretože prejdú minimálne dvomi kontrolami.

(Toto je nejaká miskoncepcia. Autor (myslím teraz na autora pôvodného zdroja, z ktorého študent/ka čerpal/a) zrejme prílišne generalizoval. To je ako keď hovorovo všetko pomenúvame „oné“. „Oné“ je všetko a nič. Načo je potom termín *polymorfná trieda*, ktorým by sme tie *polymorfné* chceli odlíšiť od *nepolymorfných*, keď sú vlastne všetky triedy polymorfné? Ja by som skôr povedal, že polymorfné sú všetky také, ktoré nie sú odvodené od predvolenej triedy `Object`. Tak mi to dáva lepší zmysel, no chcelo by to túto informáciu overiť priamo v dokumentácii Javy od Oracle, aby sme vedeli, čo na to hovoria tí najpovolanejší. No teraz už nie je čas... Neskôr.)

Ak nás zaujíma, či je objekt polymorfný, môžeme vykonať jednoduchý test. Ak je objekt pozitívne testovateľný viacerými (rôznymi) testami *„is-a“* (`instanceof`), tak je **polymorfný**.

(Opäť sa to vzťahuje len na jeden druh polymorfizmu.)

Príklad:

```
public interface Bylinožravec { }
public class Zviera { }
public class Jeleň extends Zviera implements Bylinožravec { }
```

Trieda `Jeleň` je považovaná za polymorfnú, pretože má viacnásobné dedičstvo.

Java podporuje ~~dva~~ typy polymorfizmu (ako som naznačil v úvode, ja viem o viacerých, ale v tejto fáze Vás nechcem „motať“):

- **statický** (metóda preťaženia – pozor, nie len, je to komplikovanejšie)
- a **dynamický**.

Statický polymorfizmus

Metóda preťaženia (angl. overloading) – umožnenie implementovať viaceré metódy v rámci tej istej triedy, ktoré používajú rovnaký názov, ale majú odlišnú množinu parametrov (podpis metódy). To predstavuje jednu zo statických foriem polymorfizmu.

Z dôvodu rôznych skupín parametrov má každá metóda iný „podpis“. (Podpis metódy je podrobnejšie vysvetlený nižšie – pri vysvetľovaní konceptu prepísania metódy.) To umožňuje kompilátoru identifikovať, ktorá metóda má byť volaná a viazať ju už počas prekladu programu. Tento prístup sa nazýva statická väzba (angl. static binding).

Skupiny parametrov sa musia líšiť aspoň v jednom z týchto troch kritérií:

- musia mať iný počet (parametrov), napríklad jedna verzia metódy prijíma dva a ďalšia tri parametre,
- (údajové) typy parametrov musia byť odlišné, napríklad jedna verzia metódy prijíma reťazec (`String`) a druhá celočíselný typ (`int`, `long`…),
- prijímané parametre sú uvedené v odlišnom poradí, napríklad jedna verzia metódy prijíma reťazec a celé číslo a ďalšia naopak celé číslo a reťazec. (Tento druh preťaženia sa neodporúča, pretože môže byť pre programátorov mätúci.)

Vo väčšine prípadoch predstavuje každá z týchto preťažených metód inú, ale veľmi podobnú funkčnosť.

Preťaženie metódy

Umožňuje definovať viac, než jednu metódu s rovnakým názvom, ak sa parametre metód odlišujú v počte, postupnosti a údajových typoch parametrov.

Dynamický polymorfizmus

Pri dynamickom polymorfizme sa výber volanej verzie metódy rieši dynamicky, počas činnosti programu (nie v čase kompilácie). Dynamický polymorfizmus je realizovaný viacerými rôznymi konceptmi. Prepísanie metódy je jedným z nich. (Ďalšie nebudeme v tejto fáze rozoberať.)

Prepísanie metódy (angl. overriding)

Prepísanie metódy je technikou implementácie dynamického polymorfizmu. Znamená to definovať v odvodenej triede takú metódu, ktorá má rovnaký názov a podpis ako metóda v základnej (rodičovskej, nadradenej) triede. To zanemená, že najprv definuje určitú metódu v základnej triede a potom definujeme rovnakú metódu v odvodenej triede. Počas činnosti programu, pri volaní takejto metódy jej menom, je spustená prepísaná metóda (metóda definovaná v odvodenej triede).

Na úplné pochopenie musíme definovať podpis metódy. Keď definujeme metódu, tak jej hlavička má tri časti: *návratový údajový typ*, *názov* a *zoznam parametrov s určením ich typov*. Napríklad:

```
int pridať(float a, boolean b);
```

Potom pre túto metódu je:

- `int` – návratový (údajový) typ metódy,
- `pridať` – názov metódy,
- `float a, boolean b` – zoznam parametrov s určením ich typov.

Teraz, ak definujeme takúto metódu v základnej triede a potom aj v odvodenej triede presne to isté, ale zachováme implementáciu inú, tak hovoríme, že metóda prepísania je prepojená metóda s ohľadom na tieto triedy.

(Pardon, ale toto už som nebol schopný pochopiť a ani opraviť. Táto kapitola bola totiž celá preložená cez Google Translate a jej oprava ma poriadne unavila. Ak bude čas, tak sa k tomu ešte vrátim…)

Zdroje

- «*neuvedené*»

Tvorba dokumentácie (Javadoc)

Javadoc je generátor dokumentácie vytvorený spoločnosťou Sun Microsystems pre jazyk Java na generovanie dokumentácie vo formáte HTML zo zdrojového kódu Javy. Formát „Poznámky k dokumentom“, ktorý používa Javadoc, je priemyselným štandardom na tvorbu dokumentácie tried Javy. Mnoho programátorských editorov zdrojových súborov pomáha programátorovi pri tvorbe dokumentácie Javadoc a zároveň používa informácie uvedené v komentároch Javadoc pri ponúkaní rýchlych pomocných informácií pre programátora (rýchly pomocník zobrazovaný počas písania kódu).

(**Poznámka:** Komentáre Javadoc sú stručne spomenuté aj v nasledujúcej kapitole.)

Zdroje

- «*neuvedené*»

Komentáre

Komentár sa vkladá pred alebo za časť zdrojového kódu, používanie komentára nie je povinné, umožňuje však pomoc pri pochopení činnosti programu. Komentár nemá opisovať, čo program robí, ale prečo to robí.

Komentáre nemajú vplyv na fungovanie výsledného algoritmu, sú skôr uvedené ako pomôcka na jednoduchšiu orientáciu v zdrojovom kóde.

- **Jednoriadkový komentár** sa začína znakmi `//` a končí sa na konci riadka. Prvý výskyt znakov `//` zmení zvyšok riadka na komentár.
- **Viacriadkový komentár** sa začína znakmi `/*` a končí sa znakmi `*/`. Všetko medzi nimi je súčasťou komentára.
- **Javadoc** komentár sa začína znakmi `/**` a končí znakmi `*/`. Všetko medzi nimi je súčasťou komentára, z ktorého sa zároveň môže vyrobiť Javadoc dokumentácia k programu v podobe automaticky generovanej webovej stránky.

  (**Poznámka:** Toto bolo veľmi stručné. K Javadoc komentárom treba povedať ešte aspoň to, že v rámci nich používame rôzne špeciálne reťazce na určenie parametrov metód `@param`, typov návratových hodnôt metód `@return`, zoznam výnimiek vrhaných metódou `@throws`, určenie odkazov na súvisiace metódy `@see` a podobne – je ich mnoho.)

Príklady:

```
// Toto je jednoriadkový komentár

/*
    Toto je
    viacriadkový
    komentár
*/

/**
Toto je Javadoc komentár.
*/
```

```
/**
 * Javadoc komentáre bývajú zarovnané s pomocou
 * hviezdičiek, ktoré sú rýdzo kozmetické. Na začatie
 * komentára je dôležitá len trojica znakov lomka
 * s dvomi hviezdičkami a na ukončenie komentára
 * je dôležitá len dvojica znakov hviezdička s lomkou.
 */
```

Zdroje

- «*neuvedené*»

Dôležitosť rozdelenia riešeného (programovaného) problému na menšie celky

Podstata tejto myšlienky je založená v starom známom *„rozdeľuj a panuj.“* Môžeme ju použiť prakticky všade. Rozdelíme celý problém na niekoľko menších problémov, a tak riešime malé, jednoduché a dobre známe problémy.

Prečo rozdeľovať program na menšie celky?

Najjednoduchšia a najrýchlejšia odpoveď by mohla byť asi takáto: *„Pomáha to pri organizácii.“* Napríklad je jednoduchšie mať všetko o používateľskom rozhraní na jednom mieste a ostatné (funkčné) záležitosti na inom mieste.

Ak to chceme povedať trocha odbornejšie, rozdelenie problému nie je len praktické, ale veľmi dôležité pri zvládaní zložitých problémov. Triedy, alebo iné typy modulov, nám dovoľujú „rozdeliť obavy.“ Vieme, kde máme hľadať konkrétnu vec – ak chceme urobiť zmeny v súvislosti konkrétnou záležitosťou, dokážeme ich zacieliť na jedno vopred známe miesto (triedu), a nemusíme hľadať rôzne miesta, kde sme túto vec použili. Program je preto ľahko **čitateľný.** Vykonáme zmenu a zároveň vieme, že sa zmena vykoná všade, kde je to potrebné. Rozdelenie riešeného problému je dôležité na **údržbu kódu.**

Ak celý kód vložíme do jedinej triedy, musíme myslieť na všetko naraz. V prípade malých projektov by takéto programovanie mohlo fungovať, ale pri veľkých projektoch je to v podstate nemožné. Preto takýto program rozdelíme na funkčné časti vo svojich vlastných triedach, aby sme „zapuzdrili“ (uzavreli) logiku. Ak pracujeme s určitou triedou, nepotrebujeme myslieť na zvyšok kódu, môžeme sa sústrediť len na jeden „maličký kúsoček,“ s ktorým práve pracujeme. Dôležitosť rozdelenia problému spočíva aj v **efektívnosti.**

Ďalšou výhodou je **znovupoužiteľnosť.** Pri programovaní spúšťa hlavná trieda vopred naprogramované metódy, ktoré môžu byť vhodné aj pre iné programy.

Dôležitou črtou je **testovateľnosť.** Je jednoduchšie otestovať, nájsť chybu a následne ju opraviť v relatívne malej časti programu, ako v programe s jedinou triedou a niekoľko sto či tisíc riadkami kódu.

Zdroje

- «*neuvedené*»

Vývoj založený na rozdeľovaní zodpovednosti (responsibility-driven design), párovanie (coupling), previazanie (cohesion) a refaktoring (refactoring)

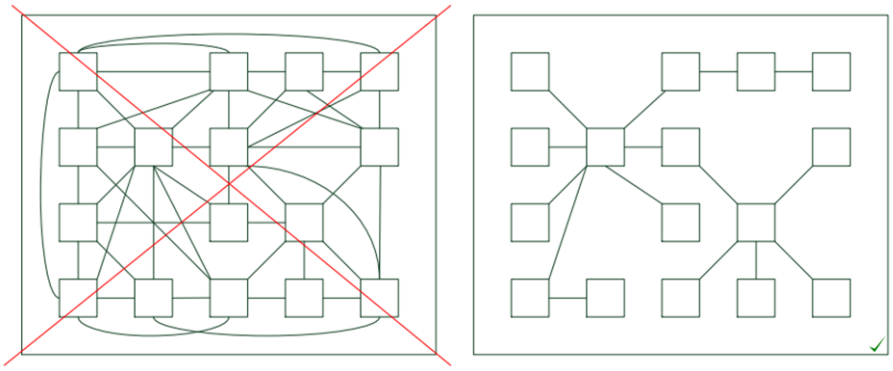
(Túto kapitolu by som musel celú prerobiť. Nenašiel som vhodné zdroje so slovenskou terminológiou na porovnanie, takže v niektorých termínoch neviem, či veriť nadpisu, alebo skôr prekladu nižšie, ktorý je pravdepodobne produkciou Google Translate. Na problematické časti upozorňujem farebne.)

Rozdeľovanie zodpovednosti

- Rozdelenie veľkej úlohy na menšie podúlohy (veľký systém na menšie časti),
- minimalizovať vzájomné previazanie jednotlivých častí (ideálne každú časť riešiť nezávisle na inej časti),
- maximalizovať súdržnosť časti systému (vždy sa zaoberať práve jednou časťou systému).

Vzájomná ~~previazanosť~~ (angl. coupling)

- Ide o tzv. párovanie.
- V OOP: sa previazanosť (angl. coupling) vzťahuje k tomu, že jeden prvok má vždy iný prvok, inými slovami, ako často dochádza k zmenám v triede A, tak isto dochádza k zmenám v triede B.
- Previazanosť je určená množstvom väzieb medzi softvérovými entitami (funkcie, triedy, metódy, balíky...).
- Previazanosť by mala byť minimalizovaná.
- Minimalizácia množstva väzieb medzi entitami znižuje ich vzájomnú závislosť.



Jestvujú dva typy previazanosti:

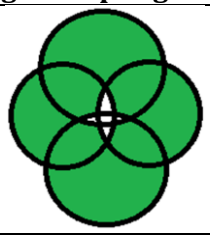
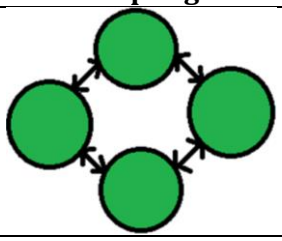
1. pevné viazanie (angl. tight coupling) –pri pevnom viazaní vo všeobecnosti platí, že sa menia obidve triedy spoločne.

```
// Program Javy na ukázanie konceptu pevného viazania (tight coupling)

class Námet
{
    Téma t = new Téma();
    public void čítajKnihu()
    {
        t.rozumiem();
    }
}

class Téma
{
    public void rozumiem()
    {
        System.out.println("Koncept pevného viazania (tight coupling)");
    }
}
```

2. voľné viazanie (angl. loose coupling) – párovanie je nezávislé.

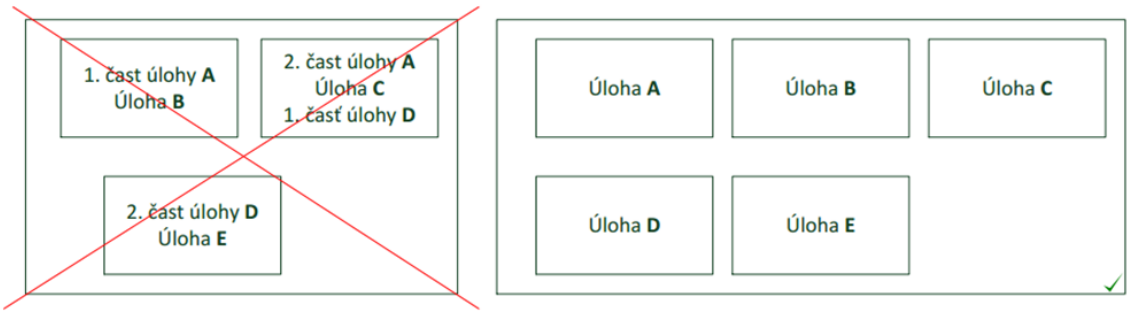
```
// Program Javy na ukázanie konceptu voľného viazania (loose coupling)
```

```
public interface Téma
{
    void rozumiem();
}

class Téma1 implements Téma
{
    public void rozumiem()
    {
        System.out.println("Mám to!");
    }
}

class Téma2 implements Téma
{
    public void rozumiem()
    {
        System.out.println("Rozumiem.");
    }
}

public class Námet
{
    public static void main(String[] args)
    {
        Téma t = new Téma1();
        t.rozumiem();
    }
}
```

1. tight coupling	**2. loose coupling**
	
viac vzájomnej závislosti	menej vzájomnej závislosti
viac koordinácie	menej koordinácie
viac informačných tokov	menej informačných tokov

~~Súdržnosť~~ (angl. cohesion)

- Tzv. previazanie.
- To, čo spolu vzájomne súvisí, je vhodné implementovať práve v jednej softvérovej entite (funkcia, trieda, metóda, balík).
- Nie je vhodné implementovať niekoľko vecí v jednej entite.
- Súdržnosť vyjadruje dodržanie tejto zásady.
- Je vhodné maximalizovať súdržnosť.
- Ak je úloha zložitejšia, treba rozdeliť jednotlivé podúlohy jednotlivým podsystémom (balíkom, triedam, metódam, funkciám).



- V OOP: vzťahuje sa na to, ako je navrhnutá jedna trieda.
- Súdržnosť je objektovo orientovaný princíp, ktorý je navyše spojený s tým, že sa učí, že trieda je navrhnutá s jediným, dobre zameraným účelom.

```
// Program Javy na ukázanie konceptu high cohesive behavior

class Násobiť
{
    int a = 5;
    int b = 5;
    public int násob(int a, int b)
    {
        this.a = a;
        this.b = b;
        return a * b;
    }
}

class Zobraz
{
    public static void main(String[] args)
    {
        Násobiť m = new Násobiť();
        System.out.println(m.násob(5, 5));
    }
}
```

Refaktoring (angl. refactoring)

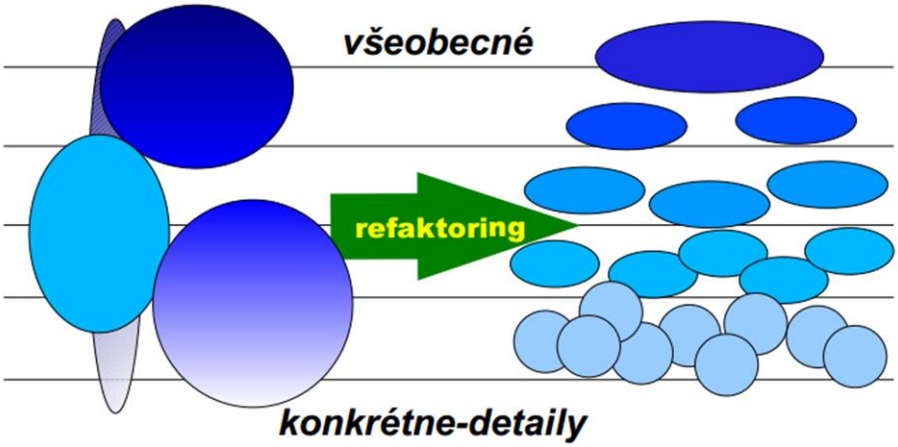
Termín **refaktoring** označuje takú zmenu štruktúry alebo návrhu programu, ktorá nespôsobí zmenu jeho správania.

Ciele refaktoringu:

1. čistejší kód, lepší návrh,

2. čitateľnejší kód, ľahšia ďalšia údržba,

- Proces reštrukturalizácie kódu postupuje po malých krôčikoch (po jednotlivých refaktoringoch).
- Predíde sa tak zmene správania, ktorá nie je žiadúca.
- Zamedzí sa tak aj „implementácii“ nových chýb.
- Technika na vylepšenie jestvujúceho kódu bez zmeny jeho správania.
- Označenie pre jednoduchý krok (transformáciu), akúsi minimálnu jednotku v sérii úprav vedúcich k požadovanému zlepšeniu, niektoré refaktoringy používajú iné – menšie a jednoduchšie.
- Po každej zmene (kroku refaktorovania) je projekt plne funkčný.



Aké chyby kódu má riešiť refaktoring?

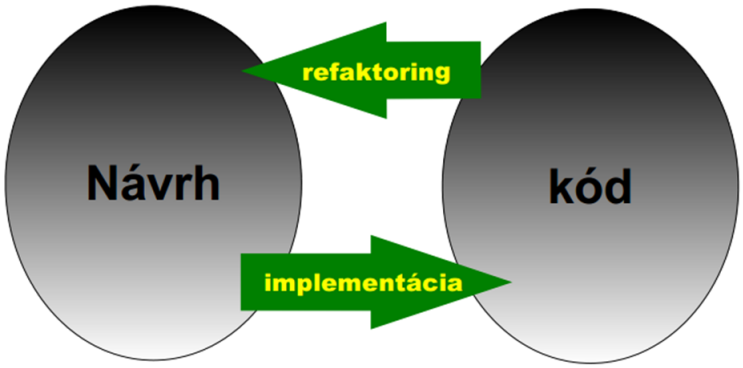
- Duplicitný kód.
- Dlhá metóda, veľká trieda.
- Dlhý zoznam parametrov.
- Zložité `switch`-e (hlavne založené na údajoch z inej triedy).
- Zbytočná prešpekulovaná všeobecnosť.
- Potreba napísať komentár v kóde (nie k metóde či triede).

Prečo refaktorovať?

Hlavnou motiváciou je:

- lepšia štruktúra (návrh) kódu,
- lepšia čitateľnosť, a teda ľahšia údržba,
- Postupnými krokmi refaktoringu je možné „opraviť“ aj dizajn,
- Refaktoring typicky vedie k čistejšiemu kódu,
- zväčša narastá počet tried a metód,
- zmenšuje sa však veľkosť tried (aj počet metód v triede),
- vznikajú tak menšie, jednoduchšie, znovu použiteľnejšie kusy kódu.

Návrh, implementácia a refaktoring



- Refaktoring je len systematický prístup k tomu, čo občas robíme tak či tak – reštrukturalizáciu kódu
- Tomu zodpovedajú názvy metód (typov/druhov/akcií/operácií?) refaktoringov:
  - vyňať (extract) metódu, triedu,
  - včleniť/vložiť (inline) metódu, triedu,
  - presunúť, premenovať, zmeniť signatúru.
- Refaktoring je systematický prístup k reštrukturalizácii kódu, nemení sa správanie programu, len štruktúra kódu.

Cieľom refaktoringu je:

- dodatočná „oprava“ návrhu, zvýšenie kvality kódu,
- zlepšenie čitateľnosti a zjednodušenie údržby.

Zdroje

- «*neuvedené*»

Programovacie paradigmy

Programovacia paradigma je spôsob konštrukcie alebo budovania charakteristických vlastností programu. Štruktúra a schopnosti programovacieho jazyka ovplyvňuje jej paradigma. Niektoré programovacie jazyky môžu byť určené len na využitie jednej paradigmy (Haskell) a niektoré môžu využívať viacero programovacích paradigiem (Python, Java). Počas relatívne krátkej histórie programovania vzniklo nesmierne množstvo programovacích paradigiem s rôznymi „odnožami“ (podmnožinami). Predstavíme si však len šesť z nich:

- imperatívna,
- funkcionálna,
- logická,
- objektová,
- symbolická,
- deklaratívna.

1 Imperatívna paradigma

Najjednoduchšie sa táto paradigma dá opísať frázou: *„First do this and next do that.“ – „Najprv urob toto a potom tamto.“* V tejto paradigme sa vykonávajú príkazy jednotlivo za sebou (akoby v prísne stanovenej postupnosti), pričom záleží na poradí zadania príkazov. V jazykoch, ktoré používajú imperatívnu paradigmu sa často dajú príkazy zoskupovať do celkov známych ako procedúry (čím už mierne zachádzame do procedurálnej paradigmy, ktorá tiež súvisí s paradigmou štruktúrovaného programovania, ktoré tu podrobnejšie nepreberáme).

Najznámejšie jazyky ktoré využívajú imperatívnu paradigmu sú Pascal (štruktúrované a procedurálne programovanie), Basic, C, Fortran, Algol.

2 Funkcionálna paradigma

Často opisovaná ako „omnoho čistejšia a prehľadnejšia“ v porovnaní s imperatívnou… Je prevažne orientovaná viac na matematickú stránku programovania a na definovanie a používanie funkcií. Funkcie sú znázorňované ako hodnoty, a otvára nové možnosti na programovanie.

Najznámejšie jazyky sú napríklad ~~PHP~~ (pozor, PHP je „známy jazyk“, ale nie pre túto paradigmu, aj keď ju môže používať, to by sme sem mohli zrovna zaradiť aj Javu, Pascal a iné) Haskell, Lisp, Scala, Python.

3 Logická paradigma

Je značne odlišná od troch najpoužívanejších paradigiem. Vyniká hlavne v schopnosti extrakcie údajov zo základných faktov. Má základ v axiómach (pravidlách), ktoré programátor píše (vkladá) a ktoré sú potom používané na riešenie zadaných problémov (hľadaní odpovedí na zadané otázky) programátorom (alebo iným používateľom). Programovanie sa po zadaní pravidiel stáva systematickým vyhľadávaním v skupinách „faktov“. Jeden z najpoužívanejších programovacích jazykov používajúcich logickú paradigmu je Prolog.

4 Objektová paradigma

Objektová paradigma je aktuálne jedna z najpopulárnejších paradigiem, hlavne z dôvodu jej systematického zoskupovania častí kódu, ktoré vzájomne úzko súvisia. Údaje sú ukladané v objektoch reprezentujúcich skutočné alebo abstraktné objekty sveta a implementujúce interakcie medzi nimi. V hlavných programovacích jazykoch tejto paradigmy sú objekty reprezentované takzvanými triedami, ktoré sú definované v určitej hierarchii (dedičnosti).

Známe objektovo orientované jazyky sú napríklad Smalltalk (jeden z pôvodných objektových jazykov), C++, C#, Java, Ruby alebo aj Python.

5 Symbolická paradigma (toto preskočte, nemal som čas sa tomu hlbšie venovať, aby som to správne skorigoval)

Je programovacia paradigma, ktorá dokáže ovplyvňovať svoje vlastné vzorce, ako keby boli obyčajnými údajmi. Časom tento jazyk dokáže sa učiť a efektívnejšie modifikovať, vďaka čomu je asi najvhodnejšiu paradigmou na umelé inteligencie.

Využíva ju napríklad Wolfram alebo Lisp.

6 Deklaratívna paradigma

Je programovacia paradigma založená na hľadaní riešenia problémov s pomocou určenia (deklarácie) toho, čo má program vykonať (nie toho, ako to má vykonať). Táto paradigma je často porovnávaná (konfrontovaná) s imperatívnou paradigmou, pričom sú zdôraznené rozdiely: imperatívny jazyk opisuje jednotlivé úkony s pomocou algoritmov a deklaratívny jazyk ponecháva, aby si algoritmy potrebné na dosiahnutie stanoveného cieľa vybral a zoradil programovací jazyk sám.

> (**Poznámka:** Na spresnenie, nejde o nejakú „umelú inteligenciu" schopnú samostatne vytvárať algoritmy, aj keď je pravdou, že jazyky používajúce deklaratívne paradigmy sa používajú aj v oblasti umelej inteligencie (podmnožinami tejto paradigmy sú funkcionálne a logické programovanie spomínané vyššie). Väčšinou však ide skôr o inteligentné použitie naprogramovaných algoritmov na nájdenie riešenia, pričom programátor iba deklaruje, čo sa má hľadať, nie ako sa to má hľadať.)

Využívajú ho napríklad SQL a XQuery.

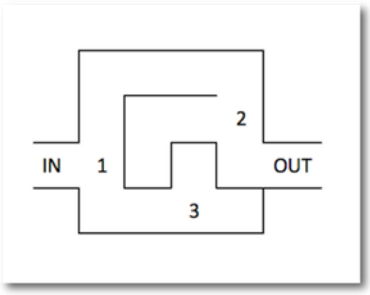
Zdroje

- «*neuvedené*»

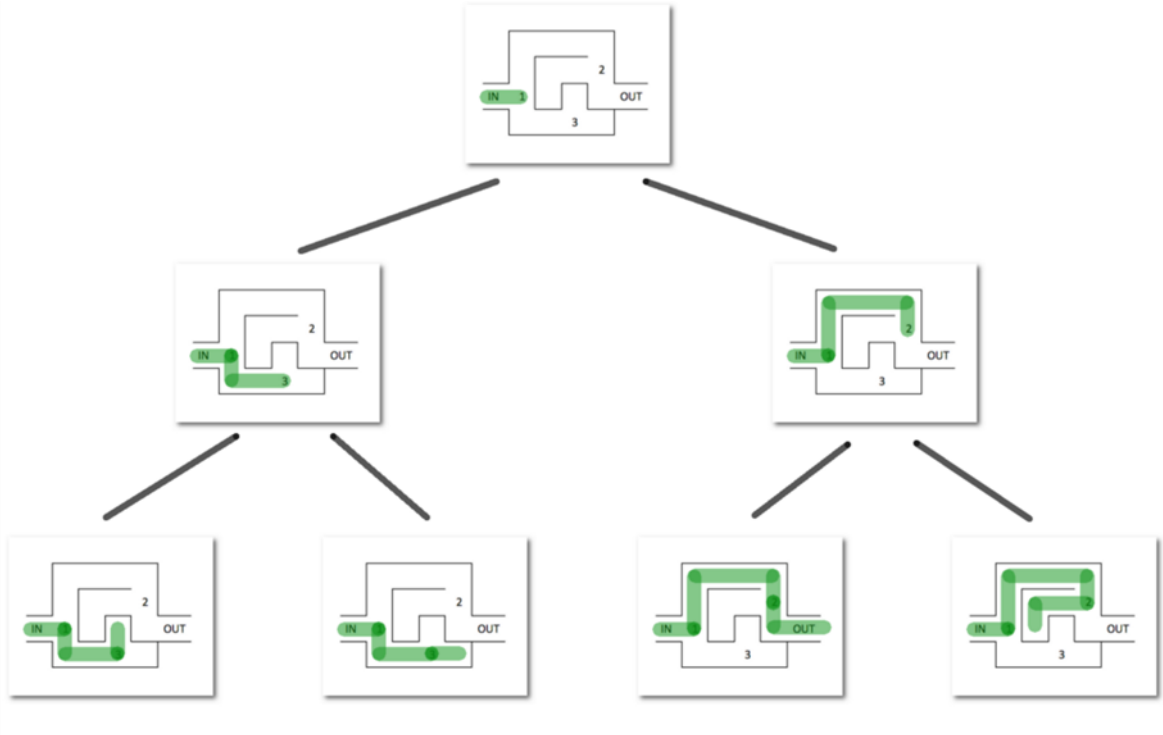
Backtracking

Backtracking je technika tvorby algoritmov, ktorá postupuje pri riešení úloh prechádzaním všetkých možných riešení. Pri backtrackingu sa postupuje tak, že program hľadá odpoveď na riešenú otázku vo vopred známom strome všetkých možných riešení, pričom, ak odpoveď na otázku nebola nájdená v jednej vetve, tak sa program vráti na predchádzajúcu a takto postupuje až do vyčerpania všetkých možností. Jednou z kľúčových súčastí algoritmov tvorených metódou backtrackingu je rekurzia. Backtrackingový algoritmus sa skončí najneskôr vtedy, keď sú vyčerpané všetky možnosti riešenia úlohy.

Najjednoduchšie sa to dá vysvetliť graficky. Pozrite sa na postup riešenia hľadania cesty von z tohto bludiska:



Funkcie sú označené ako 1, 2, a 3.



Program postupne stromovito prechádza cez všetky možné varianty riešenia (úrovne stromu). Ak sa vyčerpá jeden smer, vráti sa na predchádzajúcu úroveň a takto sa pokračuje, až kým sa nenájde riešenie alebo sa nevyčerpajú všetky možnosti na všetkých úrovniach.

Zdroje

- https://medium.com/@andreaiacono/backtracking-explained-7450d6ef9e1a
- http://people.cs.aau.dk/~normark/prog3-03/pdf/paradigms.pdf
- https://www.youtube.com/watch?v=3TBq__oKUzk

Základné riadiace štruktúry jazyka Java

Sekvencia (blok)

Je to najjednoduchšia „riadiaca“ štruktúra (alebo skôr programová štruktúra, pretože táto štruktúra v skutočnosti činnosť programu neriadi). Môžeme ju zjednodušene označiť ako „postupnosť príkazov“. Tvorí ju jeden alebo viac krokov (príkazov), ktoré sa vykonávajú postupne (chronologicky) za sebou – jeden príkaz v jednom čase. V Jave uzatvárame postupnosť príkazov (sekvenciu, blok) do zložených zátvoriek {} (niekedy nazývaných i „kučeravých“ alebo „vtáčích“ zátvoriek).

Príklad sekvencie (bloku; sčítame dve čísla a vypíšeme výsledok):

```
{
    int i = 20;
    int j = 40;
    int súčet = i + j;
    System.out.println("Súčet i a j je " + súčet + ".");
}
```

Výstup:

```
Súčet i a j je 60.
```

Vetvenia

Vetvenie je taká riadiaca štruktúra, ktorá podľa určitej podmienky (alebo podmienok) umožňuje počas činnosti programu (algoritmu) vyberať z viacerých možností/vetiev vykonávaných príkazov.

Podmienené spracovanie (`if`)

1. Ak je podmienka splnená, tak sa vykoná príkaz (resp. blok príkazov) v tele štruktúry.
2. Ak podmienka nie je splnená, tak sa blok príkazov nevykoná („preskočí“ sa).

```
if (podmienka)
{
    Blok príkazov – telo štruktúry;
}
```

Jednoduché vetvenie (`if-else`)

Z názvu vyplýva, že ide o jednoduchší zo spôsobov vetvenia. Na základe hodnoty pravdivostného (booleovského) výrazu – `true`/`false` sa vykoná jedna z dvoch vetiev. Ak je výraz pravdivý (`true` – hovoríme aj, že „podmienka je splnená/platná,“ pretože booleovský výraz nazývame aj podmienkou), tak sa vykonajú príkazy v takzvanej hlavnej vetve, inak sa vykonajú príkazy vo vedľajšej vetve. To znamená, že vždy sa vykoná nejaká vetva príkazov, ale vždy len jedna (z dvoch).

```
if (podmienka)
{
    Príkazy pre platnú podmienku;
}
else
{
    Príkazy pre neplatnú podmienku;
}
```

Viacnásobné vetvenie (`switch`)

Nie vždy je vetvenie `if-else` vhodné, niekedy je nevyhnutné vybrať jeden z viacerých úsekov (blokov, vetiev) kódu. Vtedy je vhodné použiť viacnásobné vetvenie `switch`.

Táto riadiaca štruktúra je riadená hodnotou premennej respektíve hodnotou výrazu s povoleným údajovým typom, čo môže byť celočíselná hodnota, znaková hodnota, hodnota vymenovacieho typu – `enum` a od verzie Javy 7 aj reťazcová hodnota. (Ak je namiesto výrazu uvedená len premenná, nazývame ju aj riadiacou premennou. V tom zmysle môžeme aj výraz nazvať riadiacim výrazom.)

Na základe hodnoty výrazu sa vyberie tá časť (vetva), ktorej hodnota je zhodná s hodnotou výrazu (prípadne túto hodnotu obsahuje zoznam hodnôt uvedený pri konkrétnej vetve a v Pascale môžeme použiť aj interval hodnôt, v ktorom sa má hodnota riadiaceho výrazu nachádzať).

V Jave musí byť každá vetva ukončená príkazom `break`, inak nastane takzvaný „prepad“ (angl. fall-through), čo znamená, že pri nájdení zhodnej hodnoty by sa nevykonali len príkazy zodpovedajúcej vetvy, ale aj nasledujúcej, či nasledujúcich vetiev.

Môžeme definovať aj predvolený blok príkazov – `default`, ktorý sa vykoná v prípade, že hodnota riadiaceho výrazu nie je zhodná ani s jednou z hodnôt uvedených pri vetvách tejto riadiacej štruktúry.

```
switch (testovaný výraz)
{
    case hodnota1:
        príkazy vykonané pri hodnote1 testovaného výrazu;
```

```
            break;

        case hodnota2:
            príkazy vykonané pri hodnote2 testovaného výrazu;
            break;

        default:
            príkazy vykonané pre všetky iné hodnoty výrazu;
    }
```

Cykly

Umožňujú opakovať činnosť alebo činnosti. Činnosť, ktorá sa opakuje je uvedená v príkazovom bloku, ktorý sa nazýva telo cyklu. Podmienka určuje, či sa má telo cyklu zopakovať (čiže dokedy sa bude telo cyklu opakovať).

V Jave rozlišujeme tri základné druhy cyklov:

- Cyklus so známym (explicitným) počtom opakovaní: `for`.
- Cyklus s podmienkou na začiatku: `while`.
- Cyklus s podmienkou na konci: `do-while`.

Cyklus `for`

Pri použití tohto cyklu sa dá počet opakovaní vyjadriť vopred. V Jave má hlavička tejto riadiacej štruktúry tri zložky oddelené bodkočiarkami.

Inicializačná časť slúži na naštartovanie cyklu, určenie akejsi „počiatočnej/vstupnej podmienky“. Obvykle sem vkladáme definíciu riadiacej premennej cyklu, čiže deklaráciu s inicializáciou (určením počiatočnej hodnoty), napríklad `int i = 0`.

Časť s podmienkou je vyhodnocovaná pred každou iteráciou cyklu a pravdivostná hodnota podmienky určuje, či sa cyklus zopakuje alebo nie. Obvykle sem vkladáme podmienku, ktorá akoby vyjadrovala hornú hranicu cyklu, napríklad: `i < 10`.

Iteračný príkaz je vykonaný na konci každej iterácie a jeho účelom je zmeniť hodnotu riadiacej premennej tak, aby nevznikol nekonečný cyklus, čiže pre stúpajúci cyklus treba hodnotu zvyšovať a naopak. Príkladom je inkrementácia premennej: `++i`.

Syntax príkazov:

```
for (inicializácia; podmienka; iteračný príkaz/y)
{
    príkazy;
}
```

Príklad použitia cyklu `for` – cyklus, ktorý vykoná desať iterácií a vypíše pri tom hodnoty od `0` do `9`:

```
for (int i = 0; i < 10; ++i)
{
    System.out.println("Hodnota počítadla je: " + i);
}
```

Výsledkom príkladu je nasledujúci výpis:

```
Hodnota počítadla je: 0
Hodnota počítadla je: 1
Hodnota počítadla je: 2
Hodnota počítadla je: 3
Hodnota počítadla je: 4
Hodnota počítadla je: 5
```

```
Hodnota počítadla je: 6
Hodnota počítadla je: 7
Hodnota počítadla je: 8
Hodnota počítadla je: 9
```

Cyklus `while`

Ide o cyklus s podmienkou na začiatku. Ak je podmienka splnená, vykoná sa telo cyklu a vyhodnotenie podmienky sa zopakuje. Ak podmienka splnená nie je, telo cyklu sa „preskočí“ (nevykoná) a pokračuje sa vo vykonávaní príkazov nasledujúcich za telom cyklu. Ak vstupná podmienka nie je splnená už pri prvom pokuse o vykonanie príkazov tela cyklu, tak cyklus nevykoná ani jednu iteráciu. To znamená, že pri tomto type cyklu sa príkazy tela cyklu nemusia vykonať ani raz.

Syntax príkazov:

```
while (podmienka)
{
    príkazy;
}
```

Príklad použitia cyklu `while`:

```
int i = 3;

while (i > 0)
{
    System.out.println("Hodnota premennej i je " + i + ".");
    --i;
}
```

Výsledkom príkladu je nasledujúci výpis:

```
Hodnota premennej i je 3.
Hodnota premennej i je 2.
Hodnota premennej i je 1.
```

Cyklus `do-while`

Najprv sa vykoná telo cyklu a až potom sa overuje splnenie podmienky. Ak je podmienka splnená, telo cyklu sa vykoná (resp. zopakuje) a ak nie je splnená, tak sa cyklus ukončí. Z toho vyplýva, že telo cyklu sa vykoná minimálne raz.

Syntax príkazov:

```
do
{
    príkazy;
}
while (podmienka);
```

Príklad (simulácia vstupného prúdu znakov):

```
String s = "1940aka";
int i = 0, n = 0;
char ch = 'k';

do
{
    ch = s.charAt(i++);
    if (ch >= '0' && ch <= '9')
    {
        n += ch - '0';
```

```
            System.out.println("Aktuálny počet: " + n);
        }
        else if (ch >= 'a' && ch <= 'z')
        {
            System.out.print("Výpis: ");
            for (int j = 0; j < n; ++j)
                System.out.print(ch);
            System.out.println();
        }
    }
    while ('k' != ch);
```

Výsledok:

```
Aktuálny počet: 1
Aktuálny počet: 10
Aktuálny počet: 14
Aktuálny počet: 14
Výpis: aaaaaaaaaaaaaa
Výpis: kkkkkkkkkkkkkk
```

Tento príklad je simuláciou vstupného prúdu znakov. V tele cyklu sa „čítajú“ znaky z reťazca `s`. Ak je znak číslica (rozsah `'0'` až `'9'`), tak sa jej hodnota pripočíta k počítadlu `n`. Ak je znak malé písmeno bez diakritiky (rozsah `'a'` až `'z'`), tak sa vypíše počet písmen podľa hodnoty `n`. Malé písmeno `'k'` ukončuje činnosť cyklu.

Je dôležité povedať, že tento príklad je vhodným kandidátom na vznik výnimiek (pozri kapitolu o výnimkách). Ak by bola reťazcová premenná `s` (ktorou v tomto príklade reprezentujeme vstupný prúd znakov) nesprávne inicializovaná, tak by vznikla niektorá z nasledujúcich výnimiek:

- `java.lang.StringIndexOutOfBoundsException` – ak by premenná `s` neobsahovala ani jeden výskyt znaku `'k'` (prípadne ak by bol reťazec v `s` prázdny).
- `java.lang.NullPointerException` – keby premenná `s` neobsahovala žiadny objekt – obsahovala by hodnotu `null`.

Zdroje

- «*neuvedené*»

Spracovanie výnimiek (v jazyku Java)

Výnimky (angl. exceptions) – implementujú mechanizmus zabezpečenia vzniknutej chyby počas činnosti programu. Mechanizmus vykonáva určité kroky už počas prekladu programu (napríklad overuje, či môže požadovaná výnimka v bloku vzniknúť, alebo či sú vznikajúce výnimky správne zachytávané).

Výnimky môžu vznikať z rôznych príčin, či už z hardvérových (poškodenie disku) alebo softvérových (delenie nulou). Najpoužívanejším príkladom používania výnimky je indexovanie poľa. V prípade, že sa usilujeme použiť index väčší, než sme uviedli pri deklarácii poľa, tak vznikne výnimka.

Mechanizmus vzniku (vrhnutia, angl. throw) výnimky

Ak sa v nejakej metóde vyskytne chyba, tak metóda vytvorí (vrhne) objekt typu `Exception` a posunie ho časti kódu na spracovanie. V tomto objekte sú uchované informácie o stave programu počas vzniku výnimky a informácie o type výnimky. Potom interpreter nájde miesto, kde sa má výnimka spracovať a tento kód spustí, pričom výnimka (objekt typu `Exception`) je tomto kódu posunutá ako parameter. Ako sme naznačili, vytvorenie objektu typu `Exception` a jeho posunutie časti kódu na spracovanie sa

nazýva „vrhanie výnimky“, angl. throwing an exception. Dôvodom na vrhnutie výnimky je väčšinou neschopnosť vykonať požadovanú akciu a vrátiť požadovanú hodnotu.

Príčinou môže byť:

Volajúci kód porušil kontrakt – neboli dodržané podmienky, ktoré musia platiť, aby metóda mohla vykonať očakávanú činnosť.

(**Poznámka:** Na tomto mieste treba vysvetliť, čo sa myslí pod termínom „kontrakt.“ Programovanie, najmä rozsiahlejších projektov, je vo veľkej miere závislé od dodržania tzv. „dohôd“ – kontraktov, ktoré budú všetci členovia tímu dodržiavať. Sú to pravidlá, ktoré majú zabezpečiť čo najnižšiu chybovosť kódu a ich stanovenie je nevyhnutné, pretože programovanie je tvorivá činnosť a mnohé veci sa dajú riešiť viacerými, často diametrálne odlišnými, spôsobmi. Dokonca aj programátor „samotár“ si musí pri vytváraní (riešení) väčších projektov stanoviť svoje vlastné „kontrakty“, ktoré potom dodržuje. Niekedy kontrakt zabezpečuje efektívnosť, inokedy jeho stanovenie smeruje vyslovene k zníženiu chybovosti výsledného softvéru a niekedy ide len o akési „nájdenie spoločnej reči“ v rámci tímu. Kontraktom sa môže rozumieť aj určitá filozofia zabudovaná priamo do jazyka. Nej sú potom prispôsobené typy výnimiek a situácie, pri ktorých vznikajú.)

Nie sú splnené podmienky potrebné na úspešné vykonanie kódu:

- nie je dostupné pripojenie k internetu,
- server, s ktorým potrebujem komunikovať zlyhal,
- súbor sa nenašiel,
- disk sa zaplnil,
- požadované údaje nie sú k dispozícii
- a podobne.

Pracovať môžeme s výnimkami, ktoré vrhajú cudzie metódy alebo môžeme výnimky vrhať aj v nami napísaných metódach. Vrhať môžeme už vopred definovaná výnimky (`java.lang.NumberFormatException`, `java.lang.IllegalArgumentException`…) alebo môžeme „vyrobiť“ (definovať) vlastné výnimky. Výnimky sú v podstate obyčajné triedy, ktoré dedia vlastnosti od rodičovskej triedy `java.lang.Exception`. Takže môžu mať vlastné inštančné premenné (atribúty), metódy a konštruktory. S ich pomocou dokážeme odoslať veľmi komplexnú informáciu o vzniknutej situácii a často aj návrh na jej vyriešenie. Definujme výnimku `ZápornýVstupException`.

```
public class ZápornýVstupException extends Exception
{
}
```

Majme príklad, ktorý počíta faktoriál čísla `n`. Ak bude hodnota parametra `n` záporná, tak vrhneme výnimku `ZápornýVstupException`.

```
public class Počítadlo
{
    public int faktoriál(int n) throws ZápornýVstupException
    {
        if (n < 0)
            throw new ZápornýVstupException(); // vrháme výnimku

        int faktoriál = 1;

        for (int i = 1; i < n; i++)
        {
            faktoriál = faktoriál * i;
        }

        return faktoriál;
    }
}
```

Na tomto príklade je zároveň ukázané, že výnimka je obyčajný objekt, vytvorený s pomocou operátora `new`. Kľúčové slovo `throw` tvorí príkaz, ktorý okamžite preruší činnosť metódy a vrhne výnimku. Výnimky, ktoré môže metóda vrhať (to znamená, že ich nezachytáva vo svojom tele, ale posúva ich na vyššiu úroveň volania metód), treba uviesť v jej hlavičke za kľúčovým slovom `throws`. Zoznam výnimiek v hlavičke je oddelený čiarkami. Takto označená metóda hovorí Jave o všetkých druhoch výnimiek, ktoré vrhá a ktoré je potrebné zachytiť („ošetriť“, spracovať). Ak takúto metódu zavoláme, Java bude požadovať zachytenie a spracovanie výnimky. To vykonáme tak, že volanie metódy umiestnime do bloku `try-catch`.

```
public class Spúšťač
{
    public static void main(String[] args)
    {
        try
        {
            Počítadlo p = new Počítadlo();
            System.out.println(p.faktoriál(-10));
        }
        catch (ZápornýVstupException e)
        {
            e.printStackTrace();
        }
    }
}
```

Aby programy, ktoré napíšete mohli používať aj iní ľudia (resp. po dlhšom čase aj vy sami), nikdy nevrhajte všeobecné výnimky, napríklad:

```
public int faktoriál(int n) throws Exception
{
    …
}
```

Ak vznikne všeobecná výnimka, používateľ kódu nemusí pochopiť, a obvykle ani nepochopí, čo presne sa stalo, čo bolo príčinou. A naopak, keď vznikne výnimka s rozumným názvom (a prípadne s rozumnými doplnkovými informáciami), napríklad `ZápornýVstupException`, tak už z názvu sa dá veľmi rýchlo pochopiť, že problém je pravdepodobne v zápornom vstupe.

Ak výnimku spôsobilo volanie inej metódy, tak sú dve možnosti:

- výnimku zachytiť v bloku `catch`, kde bude spracovaná,
- výnimku poslať na spracovanie metóde, ktorá túto metódu volala – výnimka „vybuble“ vyššie.

  (**Poznámka:** Termín „vybublať“ („bublať“) síce nepovažujem za práve najodbornejšie, ale v časovom strese som nevymyslel lepšie… Ide o posúvanie určitej informácie volaným metódam. V angličtine sa však pravdepodobne tiež používa rovnako „ľudovo“ znejúci termín – to bubble, it bubbles… neoveroval som to…)

V druhom prípade je potrebné uviesť nezachytenú výnimku za klauzulou `throws` v hlavičke metódy. Tým sa zodpovednosť za spracovanie výnimky posunie vyšším metódam. (Napríklad, keď sa pri čítaní súboru nezachytí výnimka `FileNotFoundException`, nechá sa „vybublať“ vyššie – pozri príklad nižšie.) Skôr, či neskôr sa však toto presúvanie zodpovednosti za „ošetrenie“ vzniknutého stavu musí niekde zastaviť – v opačnom prípade výnimka „vybuble“ von z metódy `main()`, čo vyústi do násilného ukončenia programu.

```
public class Počítadlo
{
    public List<Integer> načítajČísla(File f) throws FileNotFoundException
    {
        List<Integer> čísla = new ArrayList<Integer>();
        Scanner čítač = new Scanner(f); // nezachytená výnimka
```

```
                                        // FileNotFoundException
        while (čítač.hasNextInt())
        {
            čísla.add(čítač.nextInt());
            čítač.close();
        }
        return čísla;
    }
}
```

„Prebaľovanie“ výnimiek

(**Poznámka:** Termín „prebaľovanie“ je pravdepodobne odvodený z dvoch anglických termínov, ktoré sa v tejto súvislosti používajú: encapsulating – zapuzdrovanie; a wrapping – obaľovanie. V použitej literatúre sa používa termín „prebaľovanie“, avšak ja ho nepovažujem za práve najvhodnejší…)

Zaujímavejšou možnosťou je „prebaliť“ (uzavrieť) výnimku do novej, lepšie vypovedajúcej o vzniknutej situácii. Vytvoríme novú výnimku `PočítadloException`, ktorej cez konštruktor vložíme textový opis o podrobnostiach a aj pôvodnú výnimku `FileNotFoundException`, ktorú „prebaľujeme“. Ten, kto zachytí výnimku, má vďaka tomu presnejšiu informáciu o tom, čo sa stalo, pretože má novú lepšie zrozumiteľnú výnimku a pôvodnú výnimku v jednom.

```
public class Počítadlo
{
    public List<Integer> načítajČísla(File f) throws PočítadloException
    {
        List<Integer> čísla = new ArrayList<Integer>();
        Scanner čítač = null;

        try
        {
            čítač = new Scanner(f);
            while(čítač.hasNextInt())
            {
                čísla.add(čítač.nextInt());
            }
        }
        catch (FileNotFoundException e)
        {
            throw new PočítadloException("Číslo sa nepodarilo načítať ", e);
        }
        finally
        {
            if (čítač != null) čítač.close();
        }

        return čísla;
    }
}
```

„Prebaľovanie“ výnimiek má zmysel aj preto, že používateľ cudzieho kódu môže mať s nízkoúrovňovými výnimkami „vybublanými“ z vnútra programu problém. Ak má pochopiť, čo sa stalo a hlavne, čo bolo príčinou a ako to treba riešiť, musí dostať čo najviac informácií.

Príklad správneho „prebaľovania“ výnimiek „bublajúcich“ z vnútra nejakého projektu:

- `VSúboreChýbaPremennáException`: počet Balíčkov,
- `KonfiguráciaNemáPremennúException`: počet Balíčkov,
- `ModulKonfiguráciaOutOfDateException`,

- `StarýJarSúborException`: konfiguracia.jar.

Keby programátor dostal na spracovanie prvú výnimku (`VSúboreChýbaPremennáException`), pravdepodobne by nevedel, čo to presne znamená. Na základe poslednej výnimky (`StarýJarSúborException`) by mu už malo byť jasné, že potrebuje získať novšiu verziu balíčka s konfiguračným modulom.

Čo všetko sa dá vrhať?

V Jave sú dva druhy výnimiek – kontrolované a nekontrolované. Okrem nich môžu metódy vrhať aj chyby (`Errors`).

- Kontrolované výnimky musia byť buď zachytené v bloku `try-catch`, alebo sa môžu nechať „vybublať“ do volajúcej metódy. Ak sa kontrolované výnimky nechajú „vybublať“ (môžu to byť aj výnimky, ktoré priamo vytvára táto metóda), musia sa uviesť v hlavičke metódy za klauzulou `throws`.
- Nekontrolované výnimky sú potomkovia triedy `RuntimeException`. Nemusia byť zachytené a nemusia sa uvádzať za klauzulou `throws`.
- Chyby sú potomkovia triedy `Error`. Podobne ako nekontrolované výnimky sa nemusia zachytávať, ani uvádzať za klauzulou `throws`.

Chyby (`Errors`) predstavujú taký stav systému, z ktorého sa aplikácia nemá šancu zotaviť. Väčšinou ju vrhajú nízkoúrovňové metódy zvnútra Javy. Sú fatálne chyby, z ktorých (ako bolo povedané) nie je možné sa zotaviť. Príkladmi sú: `OutOfMemoryError` – indikácia vyčerpanej pamäte alebo `VirtualMachineError` – stav, keď virtuálny stroj Javy nedokáže ďalej vykonávať Java kód (napríklad, keď bol virtuálny stroj poškodený alebo vyčerpal zdroje nevyhnutné na ďalšiu činnosť).

Odpoveď na otázku, kedy použiť kontrolovanú a kedy nekontrolovanú výnimku, nie je jednoduchá. Sú vývojári, ktorí nepoužívajú kontrolované výnimky vôbec (napríklad preto, lebo sú zvyknutí na iné OOP jazyky, ktoré kontrolované výnimky nemajú). Nekontrolované výnimky majú však tú nevýhodu, že zvádzajú k lenivosti. Dobrý programátor, keď vrhá nekontrolované výnimky, tak ich uvádza do dokumentácie aj do klauzuly `throws` aj napriek tomu, že to nie je nevyhnutné.

Výhodou nekontrolovaných výnimiek je to, že keď sme si istí, že sme dodržali kontrakt a nevzniknú žiadne výnimky identifikujúce nedodržanie kontraktu, tak ich nemusíme zachytávať v `try-catch` blokoch. Ak sa vrhané nekontrolované výnimky navyše uvedú do dokumentácie a do klauzuly `throws`, tak nenastane situácia, kedy by nejaká neočakávaná (nezdokumentovaná) výnimka „vybublala“ z vnútra cudzieho kódu.

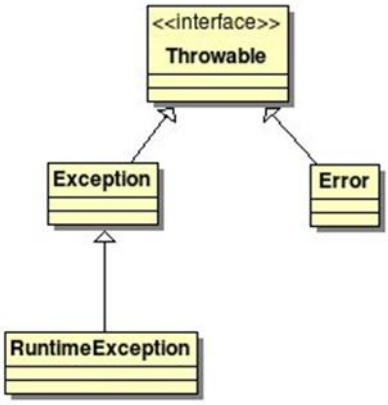
Výnimky pri prekrývaní metód

Pri kombinácii výnimiek a dedičnosti si treba dávať pozor. Platí totiž, že predvolene môžu prekrývajúce metódy vrhať iba také výnimky, ktoré vrhajú ich náprotivky (prekrývané metódy) v rodičovských triedach. Inými slovami, ak chceme v odvodenej triede v nejakej metóde vrhať viac výnimiek, než vrhá pôvodná metóda (tá, ktorú prekrývame), tak musíme zoznam výnimiek pôvodnej rodičovskej metódy uvedený za klauzulou `throws` rozšíriť.

Zachytávanie viacerých výnimiek

Zachytávanie výnimiek sa deje v blokoch `catch`. Blokov `catch` môže byť viac. Fungovanie viacerých blokov funguje na tomto princípe: najskôr sa výnimku pokúsi zachytiť prvý blok `catch`, ktorý dokáže prijať výnimku zadaného údajového typu. V každom bloku `catch` sa totiž uvádza ako parameter určitá premenná a jej údajový typ v tomto prípade určuje, či je alebo nie je možné konkrétnu výnimku v tomto bloku `catch` spracovať.

Na to, aby sme vedeli, ktoré výnimky budú zachytené niektorým blokom `catch`, potrebujeme poznať hierarchiu dedenia „vrhateľných“ (`Throwable`) objektov.



Do premennej údajového typu nadradenej triedy vieme ukladať ľubovoľný objekt (inštanciu) potomka tejto triedy (to vyplýva z princípov dedičnosti). Tento vzťah platí aj v oblasti výnimiek. To znamená, že keby sme v prvom bloku `catch` zachytávali výnimku do premennej typu `Exception`, tak by sme zachytili všetky výnimky, ktoré sú od nej odvodené. V tomto prípade by sa ako jediné nezachytili chyby (odvodené od triedy `Error`). Z toho vyplýva, že keby sme v ďalších `catch` blokoch zachytávali napríklad výnimku `FileNotFoundException`, ktorá je potomkom triedy `Exception` tak, by sme ju týmto `catch` blokom nikdy nezachytili, pretože by bola zachytená prvým `catch` blokom s typom `Exception`. Java by však na tento nesúlad upozornila pri kompilácii, preto musíme všetky bloky `catch` uvádzať vždy od najviac konkrétnych po tie najvšeobecnejšie. Príklad správneho poradia zachytávania výnimiek:

```
try
{
    …
}
catch (FileNotFoundException e)
{
    System.err.println("Súbor nebol nájdený.");
}
catch (IOException e)
{
    System.err.println("Vstupno-výstupná chyba!");
}
catch (Exception e)
{
    System.err.println("Nastala nejaká výnimka, " +
        "napríklad aj ľubovoľná RuntimeException");
}
```

Časté chyby pri používaní výnimiek

Chyba č. 1 – zachytíme výnimku, a nespravíme nič.

```
try
{
    čítač = new Scanner(f);
}
catch (FileNotFoundException e) {}
```

- Dôsledkom chyby č. 1 je, že program obvykle „spadne“ (zlyhá, zrúti sa) na úplne inom mieste a nikto netuší, prečo sa to stalo.

Chyba č. 2 – banality riešené výnimkami.

V prípade, že sa dá výnimkám predchádzať overovaním jednoduchých podmienok, je vhodnejšie výnimky nepoužiť (ich použitie vyžaduje ich vrhanie a následné spracovanie – zachytávanie). Nasledujúci príklad ukazuje nesprávne použitie výnimiek. Ukazuje prechádzanie poľa, avšak nie je použitý cyklus `for`, ale sa čaká na chybný stav – pokus o prečítanie nejestvujúceho prvku poľa.

```
try
{
    for (int i = 0;;++i) // nekonečný cyklus
        pole[i] = 2 * pole[i];
}
catch (ArrayIndexOutOfBoundsException e)
{
    // Ignorujeme výnimku – žiadne spracovanie,
    // čo je zároveň vyššie spomínaná chyba…
}
```

- Dôsledkom chyby č. 2 je, že kód je veľmi ťažko čitateľný.

Chyba č. 3 – zle zvolené a málo zmysluplné mená výnimiek alebo jednoducho priame vrhanie všeobecnej výnimky typu `Exception`.

```
void metóda() throws Exception
{
    // …
}
```

- Dôsledok chyby č. 3 – používateľ metódy netuší, čo sa môže stať a nevie sa na to pripraviť (nevie napísať správne cielený kód).

Chyba č. 4 – necháme „vybublať“ výnimky príliš vysoko, pretože „sa nám nechce písať `try-catch` bloky, veď predsa `throws` je napísané rýchlejšie.“

```
void transferÚdajov() throws IOException, SQLException
{
    // …
}
```

- Dôsledok chyby č. 4 – vo vysokoúrovňových metódach netušíme, čo sa mohlo na nižších úrovniach pokaziť a niekedy ani nie sme schopní to riešiť. Práve preto sa výnimky majú zachytiť na mieste, kde sa s tým ešte vieme vysporiadať. Ak sa s tým nevedia vysporiadať nižšie vrstvy, treba takéto nízkoúrovňové výnimky „prebaliť“ (pozri „Prebaľovanie“ výnimiek) do zrozumiteľnejších, aby bolo aspoň trochu jasné, ako by sa to malo na vyššej úrovni vyriešiť.

Zdroje

- http://www2.fiit.stuba.sk/~kapustik/JAVA/jazyk/exceptions.html#PE
- http://paz1a.ics.upjs.sk/Material/Vynimky2