

Kolekce ArrayList

napsal Pajclín

Tento článek jsem se rozhodl věnovat kolekci ArrayList, protože je to jedna z nejpoužívanějších. Tento článek není kompletním popisem třídy ArrayList, ale budu se snažit naznačit základní práci s ním. Pokud vám zde něco chybí, určitě další odpovědi najdete v dokumentaci Javy nebo v knize Java - Bohatství knihoven Pavla Herouta.

Import

První věcí, co budeme potřebovat, je ve své třídě, kde s ArrayListem budeme pracovat, zdělit překladači, že budeme používat třídu ArrayList nacházející se v balíčku java.util. To uděláme následujícím způsobem:

```
import java.util.ArrayList;
```

Důležité je to uvést ještě před hlavičku samotné třídy. Teď už překladač bude vědět, že když používáme třídu ArrayList, myslíme tím právě tu z java.util. Toto není povinné, ale museli by jsme pak všude psát java.util.ArrayList, zatímco takto stačí již dále psát jenom ArrayList.

Deklarace proměnných

Práci s ArrayListem si budeme ukazovat na instančních proměnných, ale stejně tak by to platilo pro lokální proměnné.

První věcí, co musíme udělat, než s nějakou instanční proměnnou začneme pracovat, je ji deklarovat. Řekneme překladači, jakého bude typu (v našem případě ArrayList) a jak jí budeme říkat (třeba list). Poté se v programu již uvádí pouze její jméno (v našem př. list), překladač už ví, že je to ta proměnná, kterou jsme deklarovali.

```
private ArrayList list;
```

Od verze Javy 5.0 (1.5 původního číslování) je k dispozici tzv. typový ArrayList. V této verzi říkáme překladači, jakého typu budou objekty, které do ArrayListu (do proměnné) ukládáme (v původní verzi se ukládaly Objekty a pak bylo nutné Objekt zpět přetypovat). Předpokládejme pro náš příklad, že budeme ukládat řetězce.

```
private ArrayList<String> list;
```

Vytvoření prázdné kolekce

Důležitou věcí, než začneme s kolekcí pracovat, je vytvoření samotného objektu kolekce. To uděláme pomocí konstruktoru ArrayListu. (Aby se to trochu pletlo, tak většinou tento úkon je součástí konstruktoru naší vytvářené třídy ;-))

```
list = new ArrayList();  
... v případě netypového ArrayListu nebo ...
```

```
list = new ArrayList<String>();  
... v případě typového ArrayListu.
```

Počet prvků v kolekci

Docela často se nám stane, že bychom rádi věděli, kolik máme vlastně v naší kolekci uložených prvků. Na to slouží metoda `size()`.

```
int pocet = list.size();
```

V našem příkladě by jsme v lokální proměnné `pocet` (právě jsme si ji vytvořili) měli počet prvků, které máme uložené v kolekci `list`. Pokud by tam nic nebylo, dostali bychom samozřejmě číslo nula.

Přidání prvku do kolekce

Jedním z prvních úkonů, který jistě budete chtít udělat, je nějaký prvek vložit. K tomu slouží metoda `add()`. Uvedu zde pouze tu základní verzi. Předpokládejme, že máme nějaký objekt (v př. se jmenuje `retezec`), který chceme do kolekce přidat. Pak to bude vypadat následovně:

```
list.add(retezec);
```

Daný objekt se přidá na konec kolekce. Možná se ptáte, kolik můžeme nandat objektů do naší kolekce. Je to libovolný počet, dokud nám nedojde paměť.

Získání objektu z kolekce

Když jsme do kolekce nějaké objekty naskládali, určitě je někdy budeme chtít zpět (mimo jiné proto, že dokud jsou v kolekci, tak s nimi nemůžeme přímo pracovat). K tomu máme metodu `get()` (dá se použít i metoda `remove()`, viz. dále). Jako parametr se uvádí index objektu (Tady by se mohl stát kámen úrazu pro některé programátory. Jak jej zjistím. Metod je celá řada a liší se případ od případu. To je na celé téma, takže budu předpokládat, že jej známe.).

```
String retezec = (String)list.get(2);
```

... v případě netypového ArrayListu (nutnost přetypování) nebo ...

```
String retezec = list.get(2);
```

... v případě typového ArrayListu (bez přetypování).

Parametr metody `get` musí být z intervalu $<0, \text{počet prvků} - 1>$. Pokud by byl mimo tento interval, metoda vyvolá výjimku a dojde k havárii programu.

Smazání objektu v kolekci

Čas od času se stane, že některý objekt už v kolekci nechceme. Pak jej prostě potřebujeme odstranit. Na to slouží metoda `remove()`. Jako parametr má opět index objektu.

```
list.remove(2);
```

Důležité je si uvědomit, že pokud jsme neodebírali z konce, dojde ke změně indexů všech prvků za odebraným objektem, protože se třída ArrayList snaží nemít žádné díry.

Metoda remove() nám ale zároveň s odstraněním objektu vrací i odkaz na tento objekt, podobně jako metoda get(). To se občas může hodit. Např. když prvek potřebujeme, ale zároveň ho už nechceme mít v naší kolekci.

```
String retezec = (String)list.remove(2);
```

... v případě netypového ArrayListu (nutnost přetypování) nebo ...

```
String retezec = list.remove(2);
```

... v případě typového ArrayListu (bez přetypování).

Vyčistění kolekce

Čas od času potřebujeme naši kolekci prostě vysypat a začít od začátku. Na rychlé vymazání slouží metoda clear(). (pozn. můžeme to i udělat pomocí příkazu remove() tak, že postupně odstraníme položku po položce, ale je to pomalejší)

```
list.clear();
```

Pohádka o cyklech - neboli cyklická :-)

napsal tokos

Co je to cyklus

Cyklem nazýváme jistou sadu příkazů pevně ohraničenou, kterou potřebujeme v programu opakovat. Klasicky, pro každý prvek seznamu, či pole dělej nějakou činnost(porovnej, vytiskni...). Cyklus musíme vymezit i přesnou podmínkou. Opakuj cyklus dokud platí, nebo dokud neplatí "nějaký výraz". Nyní si rozebereme jednotlivé cykly:

Cyklus FOR

For cyklus je asi nejpoužívanější a nejoblíbenější konstrukce cyklu. Využíváme ho všude tam, kde předem známe přesný počet opakování cyklu. Zde vidíme jeho přesnou syntaxi. Deklarujeme a definujeme proměnnou cyklu - ono int i. Poté definujeme jednoznačnou podmínku, dokud i je větší než nula. Na závěr přidáme změnu proměnné cyklu, která se provede vždy na konci jednoho cyklu. Tedy po každém provedení cyklu na jeho konci sniž hodnotu i o jedničku. Do těla cyklu můžeme napsat v podstatě cokoliv, to nás teď příliš nezajímá.

```
for(int i=10;i>0;i--)  
{  
    ...  
}
```

Spočítáme tedy, kolikrát se takto zapsaný cyklus provede. Provede se přesně desetkrát, skončí na i=1, i=0 už neprovede, protože jsme zadali ostrou nerovnost.

Proměnnou cyklu (i) často využíváme v těle cyklu, například při průchodu polem, kde potřebujeme přistupovat do pole podle indexů. V těle cyklu pak pracujeme s proměnnou i:

```
{  
    pole[i]=0;  
}
```

Vidíme, že pro prvek pole s indexem i se provede přiřazení. Tedy se tento prvek číselného pole upraví na nulu.

Cyklus WHILE

Cyklus while se používá tam, kde nám nestačí for, kde neznáme přesný počet opakování, definujeme pouze podmínku:

```
int i=10;  
while(i>0){  
    ...;  
    i--;  
}
```

Takto zapsaný cyklus while nám funguje stejně jako předchozí cyklus for. S tím rozdílem, že na začátku testuje VSTUPNÍ podmínku $i > 0$. Proměnnou i ale musíme deklarovat a definovat mimo podmínku. Stejně tak určujeme nastavení $i--$ po provedení jednoho opakování. A to přímo v těle cyklu. Pro tento konkrétní příklad by tedy bylo lepší použít přímo cyklus for. Cyklus while nám poslouží například zde:

```
//mame vytvoreny iterator it nejake kolekce např. typu ArrayList.  
while(it.hasNext()) {  
    ...=it.next();  
    ...;  
}
```

Zde přesně nevíme kolikrát se cyklus bude provádět. Nejistili jsme si přesný počet míčků v kolekci. Tedy provádíme cyklus dokud ještě nějaké míčky v kolekci jsou. Dokud má daný míček(zastoupený iterátorem) svého následovníka. V těle cyklu pak tohoto následovníka přiřadíme do nějaké proměnné, vybereme ho z kolekce, abychom s ním mohli dále pracovat.

Stejně tak by se dal v takovéto podobné situaci využít i cyklus for. To bychom ale museli mít trochu upravený vlastní ArrayList, spíše svůj vlastní Seznam. Kde by každý míček měl instanční proměnnou například další, která by v seznamu ukazovala na dalšího následovníka v seznamu pomocí reference-odkazu na objekt. K tomu všemu bychom museli také definovat v našem seznamu jistý první prvek(míček) - první. Hlavička cyklu for by pak vypadala následovně:

```
for(Micek x=prvni;x!=null;x=x.dalsi) {  
    x.getCislo(); //jakákoliv metoda z třídy Micek  
    x. ...;  
}
```

Cyklus tedy pracuje takto. Na začátku se přiřadí do proměnné cyklu x první míček ze seznamu. Podmínka cyklu se dá přechít jako: Prováděj dokud proměnná $x.dalsi$ není null , tedy nemá prázdný odkaz(což znamená, dokud má x svého následovníka). Na konci každého opakování cyklu se pak přiřadí do aktuálního x hodnota odkazu na další míček v seznamu a znovu se provede cyklus. Samozřejmě zde nepoužíváme oficiální iterátor, na pochopení seznamů je ale lepší naprogramovat si svůj vlastní seznam a vyzkoušet si takovéto metody s cykly na jeho procházení. (Brzy dodám příklad kódu pro takovýto jednoduchý seznam.)

Cyklus DO WHILE

Tento cyklus má podobné použití jako cyklus while s jedním hlavním rozdílem. Podmínka cyklu se netestuje hned na začátku jako u WHILE, ale až na konci, po provedení těla cyklu. Což znamená, že cyklus do while se provede VŽDY alespoň jednou! Tohoto jevu se dá využít třeba u ošetřování zadávání proměnných na konzoli. Uživatel zadává například nějaké číslo. Tedy jednou opakování proběhnout musí. Zadá číslo, na konci se provede test podmínky - pokud je číslo zadané správně, cyklus končí, pokud ne, cyklus pokračuje znovu(uživatel zadává špatně zadané číslo znovu).
Syntaxe DO WHILE:

```
do {  
    ...;  
    ...;  
} while(i>0);
```

Vtip spočívá v tom, že tento cyklus se provede i v případě pokud by na začátku bylo i menší než nula. Cyklus by se jednou provedl a až na konci by zjistil, že i je menší než nula a skončil by.