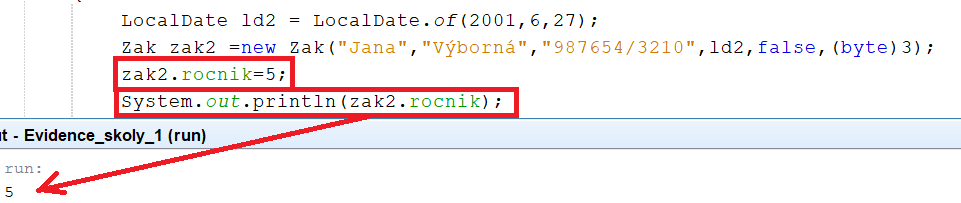
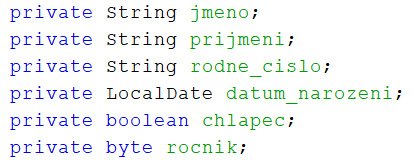
# Gettry a Settry

Při konstrukci třídy Zak jsme veškeré naše proměnné označili modifikátorem přístupu public. To znamená, že vlastnosti žáka jsou veřejné. Jak pro zápis, tak pro čtení. Lze je přepisovat či zobrazovat z kterékoli části programu. Práce s proměnnými objektu je triviální. Pomocí tečky za konkrétním objektem se k proměnné vždy jednoduše dostaneme.



Tato výhoda se však může záhy obrátit v nevýhodu. Co když díky nepozornosti během programování umožníme z úplně jiného místa programu modifikaci proměnné, kterou nechceme. Nebo tuto modifikaci někdo provede záměrně, protože prostě může. Změna jména žáka by byla lehce dohledatelná a opravitelná. Kdybychom ale například do programu přidali známkování, změna konkrétních hodnocení by se napravovala velice těžko. Navíc program by přestal být důvěryhodný a stal by se neprodejným.

Řešením je označení proměnných modifikátorem přístupu private. Proměnné přestanou být viditelné v jiné třídě než v té, ve které se aktuálně nachází. Stejného výsledku dosáhneme, když modifikátor přístupu nenapíšeme vůbec.



Zde vidíte reakci třídy Program.java, která Vám znemožní přistupovat k private atributům přímo. Červeně označené syntaktické chyby:



Jak si s tím poradit? Našimi jedinými metodami pro přístup k proměnným jsou v aktuální programu pouze:

* Konstruktory a přetížené varianty konstruktoru (pro zápis)
* Metoda toString (pro čtení)
* Metoda vratInfo (pro čtení)
* Metoda zmenPrijmeni

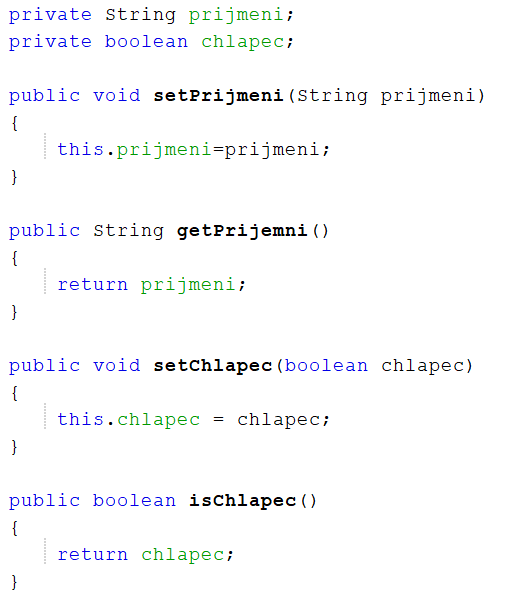
Ztratili jsme možnost pracovat s jednotlivými proměnnými. Pokud toto od programu očekáváme, je nezbytné doprogramovat příslušné metody. Jinak, než skrze metodu se k proměnné nedostaneme. Může Vám to přijít zbytečně složité. Uvědomte si ale, že Vám tato strategie umožňuje mít plnou kontrolu nad tím, které konkrétní vlastnosti žáka lze:

* Pouze číst
* Pouze zapisovat
* Číst a zapisovat
* Nelze číst a nelze ani zapisovat

Pravidla tvorby názvů metod **get** a **set** jsou ustálená a velmi jednoduchá:

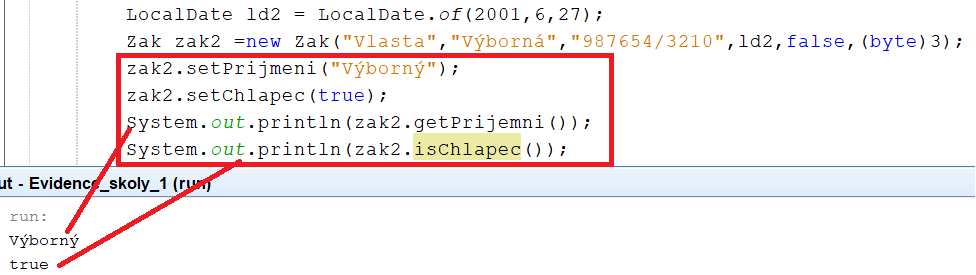
* Pro settery – zápis: setNazevPromenne
* Pro gettery - čtení: getNazevPromenne
* Pro getter typu boolean: isNazevPromenne

Zde na příkladech:



Pokud Vám nyní přijde zbytečná metoda zmenPrijmeni programovaná v minulé kapitole, máte pravdu, klidně ji vymažte. SetPrijmeni Vám udělá stejnou službu.

Samotné gettry a settry jsou samozřejmě public, aby bylo možno je volat ze třídy Program.java:



Každý atribut Vaší třídy musíte podrobit pečlivé rozvaze pro určení vhodných přístupových práv. V případě příjmení dává smysl vytvořit jak getter, tak setter. Kdyby ale žák disponoval například proměnnou pro uložení průměru všech známek, jistě by se tato hodnota vypočítávala z žákových dosavadních výsledků. Použít zde setter pro přímý přepis průměru by jistě nebylo rozumné. Z žáka nevalných výsledků by se mohl rázem stát premiant. Metoda get by byla pro zjištění hodnoty průměru naprosto v pořádku. Proměnnou průměr bychom tedy z hlediska logiky programovali pouze pro čtení.

Navíc je vždy nutné zamyslete se, zda změna hodnoty jedné proměnné by neměla zapříčinit změnu jiné proměnné programu. Zda existuje nějaká logická souvislost či vazba proměnných mezi sebou. Změna známky ovlivní celkový průměr žáka. Rodné číslo má závislost na datu narození a pohlaví. Číslo ročníku má souvislost s konkrétní třídou, ve které se žák nachází. Při dovršení 18ti let se žák stává svým vlastním zákonným zástupcem… takových spojení může být ve vašich programech celá řada a je nutné si je pokud možno všechny ohlídat. K implementaci těchto vazem Vám jistě poslouží právě settry. V těle metody set nemusíte pouze měnit hodnotu daného atributu. Lze zde rovněž zabezpečit vazby na ostatní části programu. Předejdete tak hromadě logických nesmyslů, které by Váš program jinak umožnil.

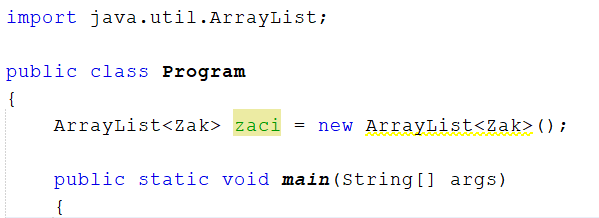
Vyvarujme se používání modifikátoru přístupu public u našich atributů. Metody get a set nám udělají mnohem lepší službu.

# Objekt ArrayList

Třídu Zak máme z podstatné části nadefinovanou. Ve třídě Program.java nyní můžeme vytvořit libovolné množství objektů – instancí třídy Zak. Pojmenujeme je zak1 až zak1000... ne, toto určitě není vhodný postup ukládání většího množství objektů. Použijeme raději **kontejner**. Nebude se ovšem jednat o klasické pole, nýbrž o vylepšený objekt typu ArrayList. Oproti poli můžeme u tohoto kontejneru dynamicky měnit počet prvků. Nadefinované metody nám například dovolí snadno vložit nový prvek doprostřed listu či jej z listu odstranit.

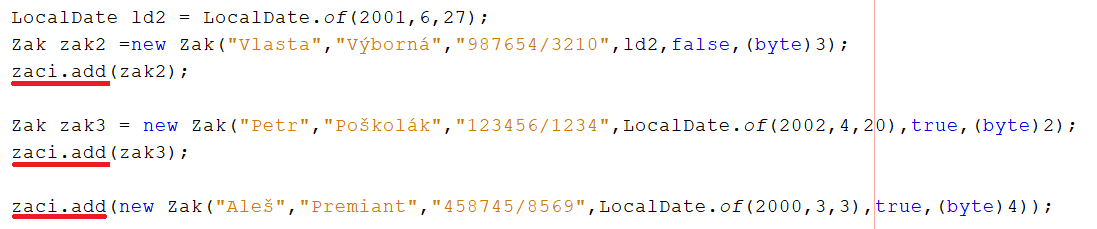
Zde uvádím příklad deklarace ArrayListu v metodě main třídy Program.java. Jedná se o tzv. **generický kontejner** – v jeho definici uvádíme, co přesně do něj chceme ukládat. V našem případě se nejedná o celá čísla int, ani textové řetězce String, ale přímo o objekty typu Zak.

Nezapomeneme importovat Java.util.ArrayList.



Žáky v metodě main snadno přidáme pomocí předdefinované metody add. Ta vždy přidává nový prvek na konec seznamu. Každým voláním metody add tedy zvětšujeme počet prvků seznamu o 1.

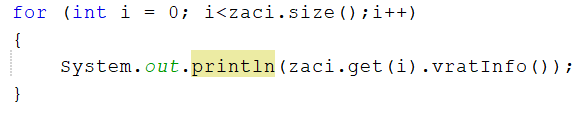
Zde uvádím 3 možné způsoby deklarace žáka a jeho uložení do seznamu.



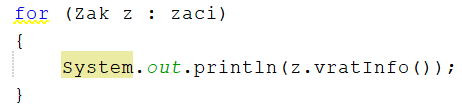
Poslední způsob je nejkratší. Má též výhodu, že nemusíme řešit pojmenování instance objektu, jako např. zak100.

Abychom si ověřili, že došlo k zápisu objektů do RAM paměti, celý seznam si projdeme index  
po indexu pomocí cyklu for. V těle cyklu pak na každé z instancí zavoláme metodu vratInfo(). Na konzoli se objeví příslušné informace o třech uložených žácích.

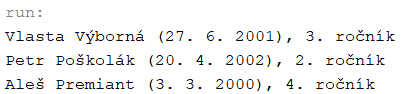
V definici cyklu voláme metodu size(), která vrací počet prvků seznamu.  
Metoda get nám umožní přečíst ze seznamu položku uloženou na daném indexu.



Jednodušší způsobem pro průchod našeho seznamu je cyklus foreach. Při použití tohoto cyklu se vůbec nestaráme o indexaci (řídící proměnnou i). Tento cyklus vždy striktně předpokládá pohyb od 0tého k poslednímu prvku. Indexace je implementována do jeho vnitřní logiky a my už ji tím pádem nemusíme programovat. Stačí nám určit zástupný název pro objekt třídy Zak, zde jsem zvolil malé z:



Konzolové výpisy při použití obou cyklů jsou totožné, dané definicí metody vratInfo():



Jak vidíte, používání seznamu k ukládání objektů je jednoduché a efektivní.

## Metody na ArrayListu

Zde přikládám seznam často používaných metod, které má třída ArrayList k dispozici:

Přidávání prvků:

* add(prvek) – na konec seznamu
* add(index, prvek) – doprostřed seznamu – index nesmí být vyšší než aktuální maximální počet prvků seznamu +1!
* addAll(konekce prvků) – více prvků najednou

Odstraňování prvků:

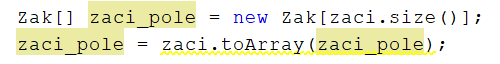
* remove(prvek) – odstraní prvek, ale pozor, pouze jeho první výskyt!
* remove(index) – odstraní prvek na daném indexu
* removeAll(index, počet) – odstraní daný počet prvku od daného indexu
* clear() – odstraní všechny prvky

Pozn. Bystřejší z Vás jistě zaznamenali možný zádrhel. Pokud bychom měli v ArrayListu uložen seznam intergerů a chtěli bychom použít metodu remove, co by znamenalo číslo na vstupu? Prvek, nebo index? Logika je nastavena způsobem, že se **smaže daný index** (aktuální hodnotu na indexu). Pokud bychom chtěli vyhledat a smazat konkrétní celočíselnou hodnotu (respektive její první výskyt), nejjednodušším způsobem je zápis: **remove(new Integer(prvek));**

Diagnostika:

* contains(prvek) – vrátí hodnotu true/false dle toho, zda se daný prvek v seznamu vyskytuje či nikoli.
* indexOf(prvek) – vrátí pořadové číslo prvního indexu, na kterém se hledaný prvek nachází. V případě nenalezení prvku vrací metoda hodnotu -1.
* lastIndexOf(prvek) – vrátí pořadové číslo posledního indexu, na kterém se hledaný prvek nachází. V případě nenalezení prvku vrací metoda hodnotu -1

Na závěr ještě zmíním metodu ToArray, která je schopna převést ArrayList na klasické pole. Někdy se může stát, že metody které chceme používat na svém vstupu tolerují pouze Array. Potom se Vám tato metoda bude jistě hodit. Zde přikládám příklad použití:



V příští kapitole se pokusíme ArrayList zakomponovat do našeho programu vhodnějším způsobem. Budeme totiž tvořit objekty typu Trida.