Polia & Java

- v Jave sú polia objekty – vytvárajú sa dynamicky
- referencia na pole celých čísel: „int[] a;“
- vytvorenie poľa celých čísel dĺžky 5: „a=new int[5];“
- incializácia polí je implicitná
  - String [] ff=new String[10];
  - stringy majú null ...
- explicitná inicializácia: „int[] a= {1, 2, 3, 4, 5};“
- explicitná inicializácia2: „Student[] a= {new Student(1),new Student(2)};“
- veľkosť poľa
  - nemusí byť známa v čase prekladu, po vytvorení je veľkosť poľa fixná
  - uchováva sa v atribúte „length“
- prístup k prvkom poľa
  - indexuje sa od 0: a[0], a[1]...
- polia sa prenášajú referenciou
- viacrozmerné polia – ako v C: „int[][] a={{...},{...},,...};“
- pomocné triedy: System, Array, Arrays (java.util.Arrays)
  - kopírovanie polí „System.arraycopy“
    - public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length))
  - porovnávanie polí
  - triedenie polí
  - binárne vyhľadávanie v utriedenom poli
  - ...

Polia2 ...

Porovnávanie polí

```
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (arr1 == arr2) {
            // zeby takto ??????
        }
    }
```

- **Nie!** Treba použiť niečo ako

```
import java.util.Arrays;
...
if (Arrays.equals(arr1, arr2)) { ... }
```

- Pre vnorené polia (polia polí) treba použiť „**Arrays.deepEquals**“ ...
  - obsahuje aj detekciu cyklov (napr. A->B->C->A), takže vie porovnať hocijaké objekty

Polia & Slučky ...

Základná slučka for pre polia	int sum(int[] a) {
	  int sum = 0;
	  for (int i=0 ; i < a.length ; i++) sum += a[i];
	  return sum;
	}

Vylepšená slučka for pre polia	int sum(int[] a) {
	  int sum = 0;
	  for (int i : a) sum += i;
	  return sum;
	}

Štruktúry v Jave

- iba pomocou tried a objektov, napr:

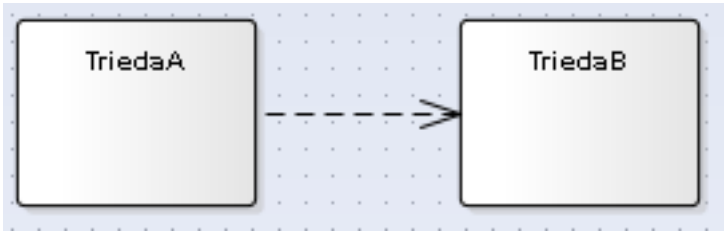
Java	C

class Book {
String title;
String author;
String subject;
int book_id;
}...
 	
public static void main(String[] args){
Book book=new Book();
...
}
	typedef struct Books {
	char title[50];
	char author[50];
	char subject[100];
	int book_id;
	} Book;
	
	int main( ) {
	Book book;
	...
	}

- ak máme mať obmedzenia dĺžky, môžeme použiť napr. : „byte [] title= new byte[50];“

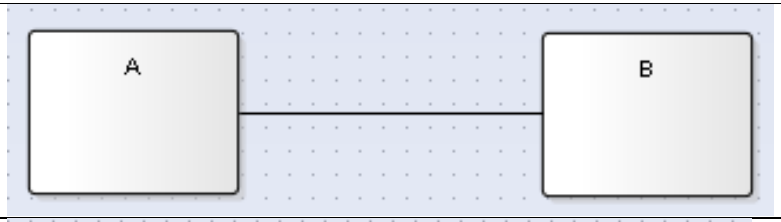
Vzťahy medzi objektami a triedami – Závislosť

- najslabšia forma vzťahu
- napr. keď:
  - metóda triedy A potrebuje argument triedy B
  - metóda triedy A vracia hodnotu triedy B
  - Metóda triedy A používa objekt triedy B v ľubovoľnom mieste implementácia, nie však ako atribút




```
class TriedaA {
...
void metodaX()
   TriedaB myB = new TriedaB()
   // tu budeme pouzivat B ako lokalnu premennu
   ...
}
```

Všeobecné vzťahy medzi objektami a triedami – Asociácia

Asociácia		Označuje, že medzi objektami týchto tried je spojenie (obojsmerné).
Asociácia Jednosmerná		Označuje, že z objektov triedy A je spojenie na objekt(y) triedy B
Reflexívna asociácia		Objekty triedy A sú vo vzťahu sami zo sebou

- Triedy môžu byť poprepájané formou atribútov, atď ...

```
class TriedaA {
    ...
    TriedaB varB = new TriedaB()
    ...
}
class TriedaB {
    ...
    static final TriedaA varA = new TriedaA()
    ...
}
```

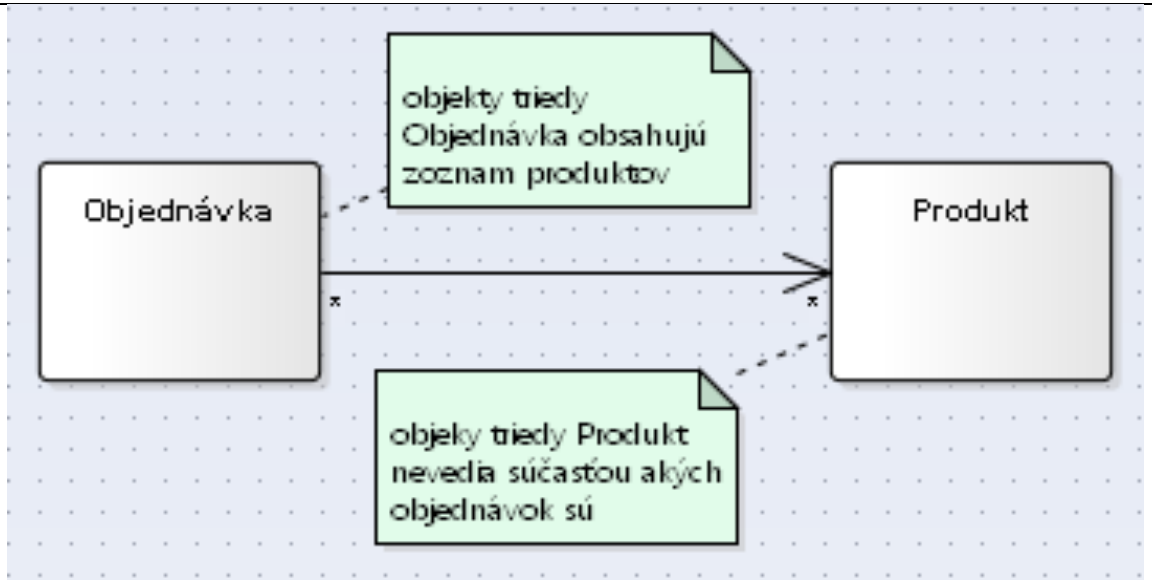
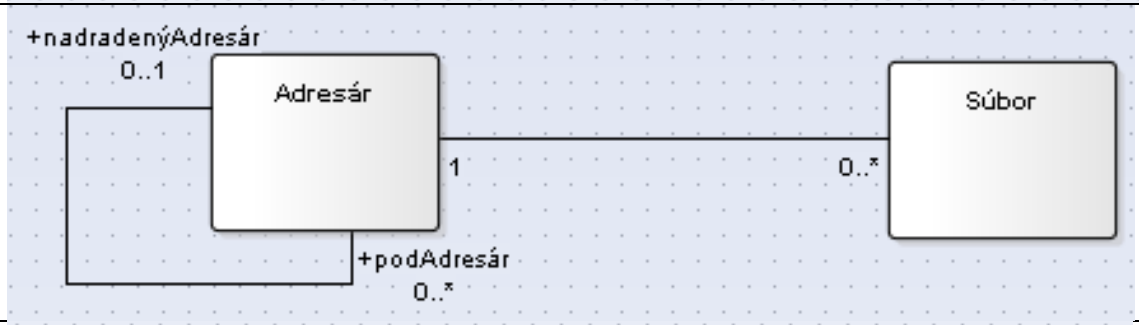
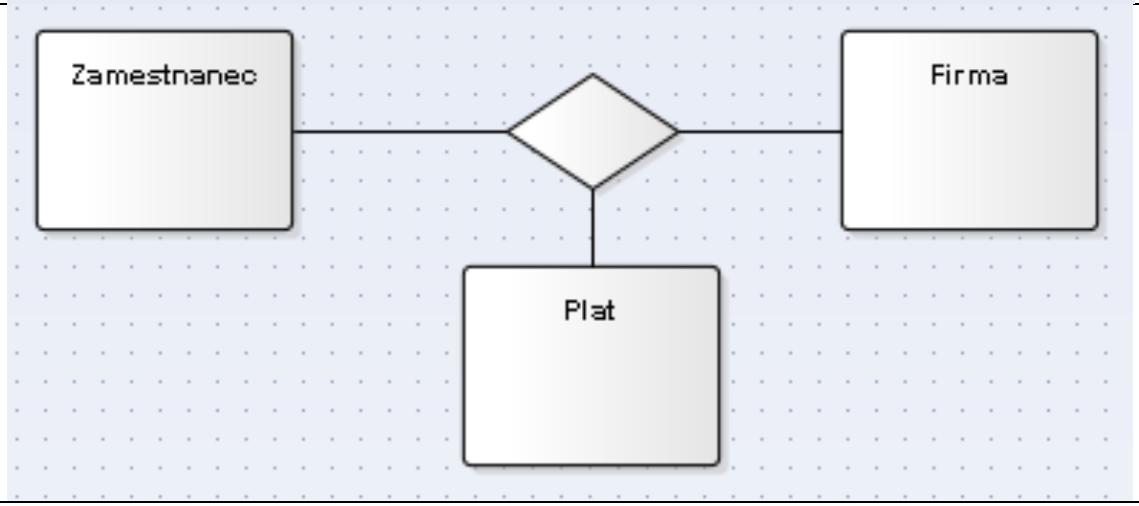
Násobnosť (kardinalita)

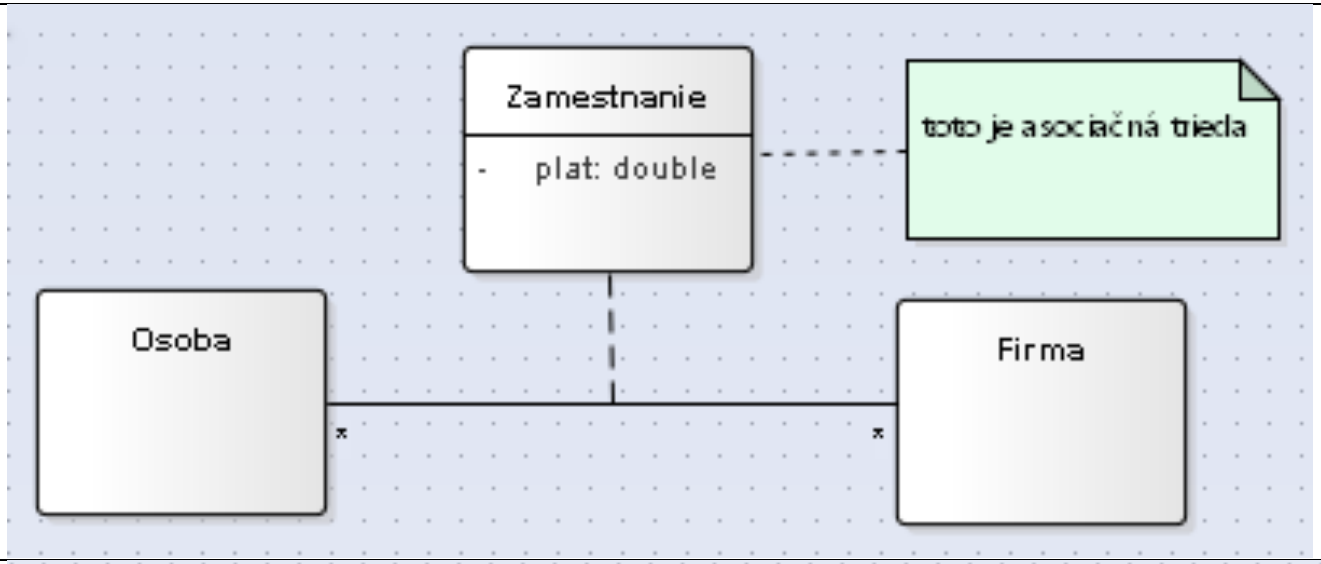
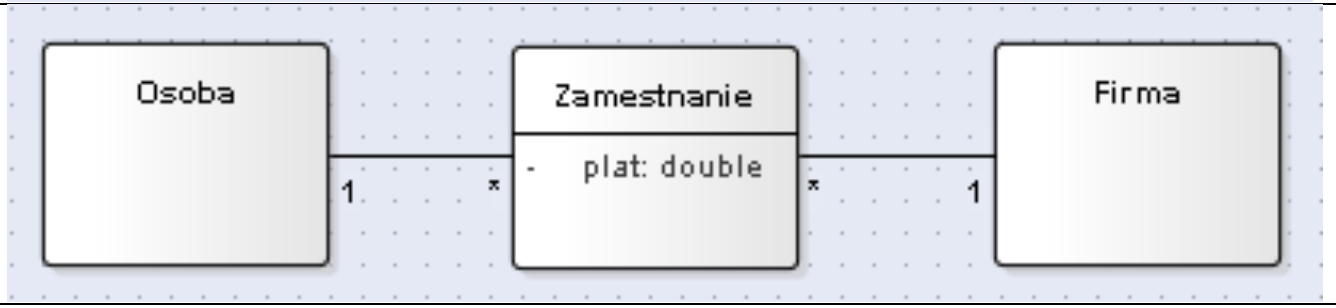
- Násobnosti udávajú v akom vzťahu sú, resp. môžu byť objekty, ktoré vytvoríme z tried.

Napr:

Napr:

vľavo	vpravo	Príklad
1	1	Osoba - rodné číslo Osoba - dátum narodenia
1	0..1	Osoba - manžel/manželka Osoba - dátum smrti
1	0..* (*)	Osoba - telefónne číslo
1..*	1	Osoba - miesto narodenia
1..*	1..*	Osoba - bydlisko
*	1..*	Osoba - adresa
*	*	Osoba - telefón Osoba - kniha

Jednosmerná asociácia	
Reflexívna asociácia	
N-árna asociácia	

Keď má asociácia zložitejšie vlastnosti ...	
Vyjadrenie asociačnej triedy ako normálnej triedy	

Vzťahy medzi objektami a triedami – Agregácia

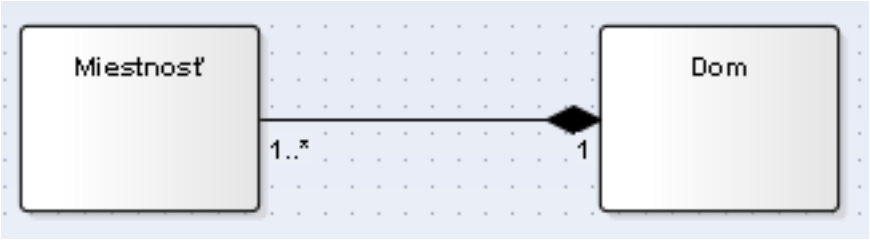
- Modeluje vzťah celok-súčasť („má“)
- Objekty v našom systéme existujú nezávisle, avšak celok, je bez týchto častí akosi neúplný
- Celok býva „dominantnejší“ v tomto vzťahu
- Časť obvykle poskytuje služby a reaguje na požiadavky celku
- Časť môže byť zdielaná viacerými celkami
- Agregácia je tranzitívna (ak A ma B a B ma C , potom A ma C)
- Agregácia je asymetrická (objekt nemôže byť súčasťou seba samého)


```
class NPrinter {
   private static final ArrayList<Nprinter> zoznam = new ArrayList();
...
}
class Printer {
   private static final ArrayList<Printer> zoznam = new ArrayList();
...
}
...
class Comp {
   private static final ArrayList<Comp> zoznam = new ArrayList();
   ArrayList<Nprinter> zoznam_nprint;
   ArrayList<Printer> zoznam_print;
...
}
```

- počítače (objektypočítačov) „majú“ odkazy na objekty Nprinter a Printer. Pri zrušení Počítača z evidencie nedôjde k zrušeniu tlačiarní

Vzťahy medzi objektami a triedami – Kompozícia

- Modeluje vzťah celok-súčasť („skladá sa z“)
- Je to silnejšia forma agregácie, pričom:
  - časti nemôžu existovať mimo celok
  - každá časť patrí jedinému celku
- celok je zodpovedný za vytvorenie aj zničenie svojich častí
  - keď je zničený celok, sú zničené aj jeho časti ak ich niekomu úmyselne neprevedie




```
class Miestnost {
...
}
class Dom {
   ArrayList<Miestnost> zoznam_miestnosti;
...
}
```

- inštancie Domov majú odkazy na objekty miestnosti
- jedna miestnosť nesmie byť naraz v dvoch domoch (nedáva to zmysel)
- keď zničíme objekt domu, zničia sa automaticky aj všetky miestnosti domu
  - dom obsahuje ako jediný referencie na objekty miestností, pri jeho zničení sú automaticky zničené aj miestnosti.

Vzťahy medzi triedami - dedenie a zovšeobecnenie

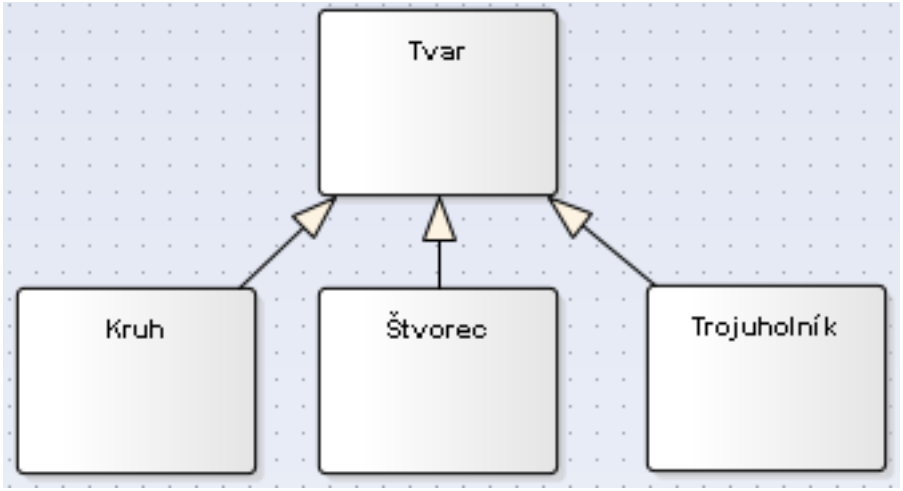
- Ide o oveľa pevnejší vzťah ako pri asociáciách
- Zovšeobecnenie (**generalizácia)** prvku A na prvok B znamená:
  - prvok A je presnejšie špecifikovaný, obsahuje viac informácii (rozširuje a konkretizuje, je špecializovaný)
  - prvok A môže nahradiť prvok B
- Mechanizmus generalizácia sa aplikuje na triedy
  - Prvok B je predok/rodič (nadtrieda)
  - Prvok A je potomok (podtrieda)
    - dedí všetky charakteristické vlastnosti predka (nadtriedy): atribúty, metódy, relácie, obmedzenia
    - môže pridať nové atribúty, metódy, relácie
    - môže redefinovať vybrané metódy
- Syntax:

```
class Nadtrieda {
...
}
class Podtrieda extends Nadtrieda {
...
}
```

Príklad 1:

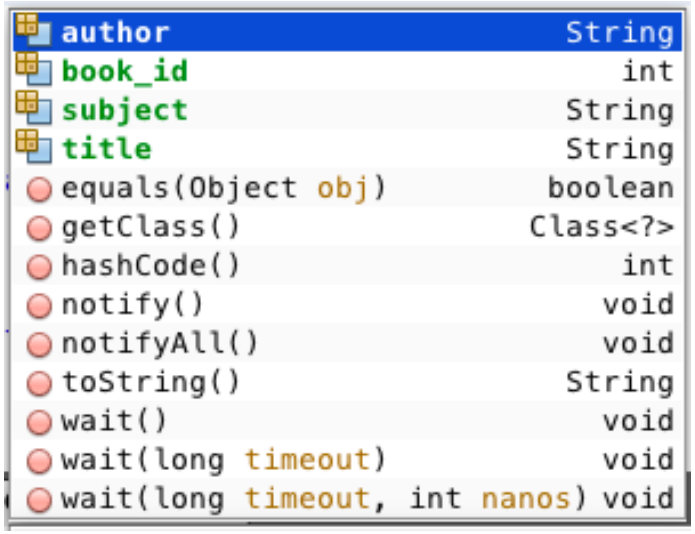
```
class Osoba {
   int vek;
   String dajPopis() {/* zakladny popis cloveka ... napr vek*/}
   ...
}
class Student extends Osoba{
   int id_studenta;
   String dajPopis() {/*do popisu zahrnie aj popis studia ...*/
      Return Super.dajPopis()+ ... ; //popis od rodica + vlastny popis
}
```

Príklad2: Diagram tried z geometrie:



Dedičnosť v Jave

- každá trieda má predka triedu Object
  - poskytuje užitočné metódy, napr „toString()“. **Príklad „class Book“:**
- každá trieda môže mať iba jedného predka! (nie ako v C++)
- k metódam predka sa dostaneme pomocou kľúčového slova „super“
- Treba dbať o správnu inicializáciu nadtriedy!
  - Implicitne sa volajú len konštruktory bez parametrov
  - Ostatné konštruktory treba explicitne zavolať (pomocou „super“)
  - Toto volanie musí byť prvý príkaz
- Keď je trieda „final“ nedá sa od nej dediť



Predefinovanie/prekonávanie (override) metód

- aby mohol potomok prekonať metódu predka, musí mať totožnú signatúru
  - prekonaná metóda predka sa dá zavolať „super.menoMetody()“
  - finálne metódy sa nedajú prekonávať
  - pri finalnych triedach sú všetky jej metódy implicitne finálne
- typ návratovej hodnoty síce nie je časťou signatúry, ale aj tak sa nesmie meniť
- Priklad: môžeme prekonať metódu „toString“ [Prednaska03.java]

Abstraktná trieda

```
class Tvar {
   Double ratajObsah(){ /* co tu ma byt ????*/}
   ...
}
class kruh extends Tvar{
   Double ratajObsah(){ /* tu bude pi*r^2*/}
   ...
}
```

- pri všeobecných triedach niektoré metódy nedávajú zmysel, až pri ich potomkoch
- aby sme mohli implementáciu „odložiť“ na potomkov, používame abstraktné triedy
- Takto by to malo vyzerať:

```
abstract class Tvar {
   abstract Double ratajObsah();
   ...
}
class Kruh extends Tvar{
   @Override   // prekladaču dávame touto anotáciou najavo, že chceme preťažovať/prekonávať
   Double ratajObsah(){ /* tu bude pi*r^2*/ ...}
   ...
}
```

Polymorfizmus

- znamená mnohotvárnosť
- podstata je, že objekty rôznych tried obsahujú metódy s rovnakou signatúrou ale s inou implementáciou
- napr. potomkovia vedia zastúpiť rodičov, pričom (možno) majú iné implementácie metód
    - napr. všetci potomkovia triedy „Tvar“ explicitne musia implementovať vlstnú metódu „ratajObsah“, ak chcú inštancovať objekty
    - potomkovia môžu byť napr: Kruh, Štvorec, Trojuholnik, Lichobežník, ...

```
...
   tvar[] polozka ={new Stvorec(1), new Kruh(1), new Stvorec(2), new Kruh(2)};
   for(int i=0;i<polozka.length;i++)
      System.out.println("Objekt "+i+" má obsah:"+polozka[i].ratajObsah());
...
```

Riadenie prístupu v dedení

- Čím viac z nadtriedy treba skryť – prístup private
- Atribúty nadtriedy s prístupom private sú v podtriedach nedostupné
- Často však treba umožniť prístup aj v podtriedach – vhodný je prístup protected:
  - prvok je dostupný každej podtriede (v celej hierarchii dedenia)
  - prvok je dostupný triedam v tom istom balíku

Balíky (packages) a triedy (opakovanie v kontexte tried)

- **triedy, ktoré spolu súvisia** možno zoskupiť do balíka (package)
- do každej triedy treba napísať „package nazov_balika;“
- baliky su organizovane hierarchicky „package nanadbalik.nadbalik.balik;“
- subory balika musia byť v adresari nadnadbalik/nadbalik/balik
- začiatok cesty je daný premennou „classpath“
- baliky sa importuju pomocou „import nadnadbalik.nadbalik.balik.*“
  - alebo sa použujú plne klasifikované názvy:

```
...
   nadnadbalik.nadbalik.balik.Student s = new nadnadbalik.nadbalik.balik.Student();
...
```

- môže sa importovať iba statická časť balíka, napr: import static java.lang.Math, potom môžeme používať:

```
...
   System.out.println(abs(cos(2*PI/3)));
...
```

Balíky – riadenie prístupu

- modifikátory prístupu (access modifiers!) public, **protected**, private
- bez modifikátora = implicitný prístup v rámci balíka („package access“)
- prístup k prvkom balíka – k triedam a rozhraniam
  - public – prvok je prístupný všetkým alebo
  - „package access“ – prvok je prístupný len prvkom toho istého balíka
  - Triedy a ich prvky definované v úplne nezávislých zdrojových súboroch, pre ktoré nie je uvedený balík sú jedna druhej prístupné
- prístupu k prvkom tried (class members) – k atribútom a metódam, ale aj k triedam a rozhraniam
  - public: prvok je dostupný všade kde trieda
  - package access – prístup v rámci balíka: prvok je dostupný len triedam v rámci balíka, napr. používateľ knižnice im nemôže pristúpiť
  - private: prvok je dostupný len v rámci triedy samotnej niekedy je potrebné zabrániť priamemu prístupu k niektorým atribútom alebo volaniu niektorých metód zvonku
  - protected: prvok je dostupný v rámci balíka a v rámci hierarchie tried, do ktorej trieda patrí
  - zobrazenie v tabuľkovej forme:

Modifikátor	Class (v rámci samotnej triedy)	Package (iné triedy v tom istom balíku)	Subclass (podtriedy danej triedy z iných balíkov)	World (všetky ostatné triedy)
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Vsuvka: Java - konvencia pomenovaní (opakovanie)

Všeobecné pravidlá:

- pokiaľ možno nepoužívať skratky, používať celé slová
- používať CamelCase / camelCase

Položka	Pravidlo	Príklad
Balík	• Forma reverznej domény (ako v DNS) firmy/organizácie • iba malé písmená + čísla oddelené bodkami • nepoužívať „_“	sk.stuba.fei.umikt.tst.tools

Trieda, Rozhranie	• skladá sa z reťazcov kapitalizovaných slov (upper camel case)
	• nepoužívať „_“
	• pri triedach
	  ○ nepoužívať slovesá, iba podstatné mená s prípadnými prívlastkami ...
	  ○ iba v jednotnom čísle (okrem kolekcií
	• názov rozhrania by mal byť prídavné meno, resp. popis vlastnosti/schopnosti
		class Student;
		class ZmluvaZamestnanca;
		
		class ZoznamAut extends ArrayList
		interface Verzionovatelna;
		interface VieRatatObsah;

Metóda
Premenná
	• skladá sa z reťazcov slov, prvé nekapitalizované, ostatné ano (lower camel case)
	• nepoužívať „_“
	• názov metódy má obsahovať sloveso
		main();
		dajSlovo();
		int myCount;

Konštanty	• veľké písmená + čísla oddelené znakom „_“	static final int SIZE=5; static final int START_VAL=2;

POJMY

Deklarácia/definícia (opakovanie)

- Deklarácia je časť kódu, v ktorom oznamujeme niektoré vlastnosti vytváraného programu. De facto je to informácia pre prekladač, na základe ktorých môže kontrolovať, či v programe nie sú nejaké evidentné chyby
- Definícia – je časť kódu, v ktorej naozaj niečo vytvárame, popisuje, ako metódy naozaj pracujú, atď. ...

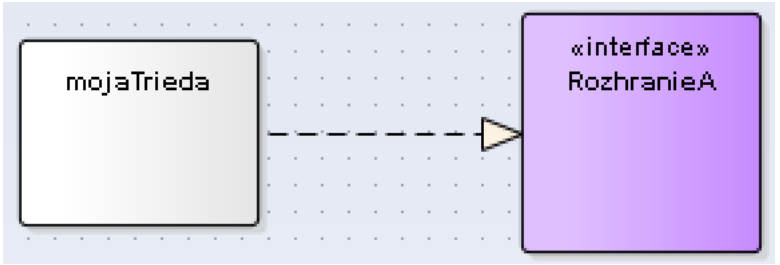
Signatúra/kontrakt

- Používajú sa napr. **pri metódach, triedach**
- Signatúra – označuje všetky vlastnosti, ktoré môže skontrolovať prekladač (v podstate všetko, čo nám prog. Jazyk dovolí deklarovať) (POZOR: ako už bolo povedané v JAVE prekladač nedovolí v tej istej triede dve metódy aby sa v signatúre líšili iba návratovým parametrom)
- Kontrakt- označuje všetky vlastnosti, ktoré prekladač skontrolovať nemôže (nemôžu sa deklarovať).
    - Dodržanie týchto vlastností je na zodpovednosti programátora.
    - Ich popis by mal byť súčasťou dokumentácie.

Rozhranie/implementácia

- Rozhranie entity – špecifikuje čo nejaká entita vie/sľubuje, zhŕňa to, čo by mal okolitý svet o danej entite na jej použitie vedieť
- Implementácia – ma na starosti to, aby daná entita robila to, čo nasľubovala

Rozhranie



- Novy dátový typ „rozhranie“ („interface“):
  - posúva koncept abstraktnej triedy ďalej ...
  - len deklaruje rozhranie
  - nemá implementáciu (nemôže definovať!)
  - môže obsahovať atribúty, tie sú však „public“, „static“ a „final“, t.j. jedná sa o konštanty.
- inštancuje sa pomocou objektov tried, ktoré dané rozhranie implementujú
  - inštancia rozhrania = inštancia triedy (objekt), ktorá dané rozhranie implementuje
- keďže rozhrania nemajú žiadnu implementáciu nehrozia problémy pri násobnom dedení a triedy môžu mať teda ľubovoľný počet rodičov – interfejsov bez spôsobovania zmätku
  - keď je treba viacnásobnú dedičnosť → vhodné použitie rozhrania!
- Rozhranie predstavuje niečo ako „protokol“ medzi triedami
- Rozhranie sa dedí

```
interface RozhranieA {
...
}
class MojaTrieda implements RozhranieA {
...
}
```

- Kedy použiť rozhranie a kedy abstraktnú triedu?
  - Keď tvoríte základnú triedu – **skúste najprv rozhranie**
  - ak treba mať definície metód, alebo atribúty … ok, zľaviť na abstraktnú triedu …
  - Ak má mať všetky definície … ok, zmeniť na konkrétnu triedu