Pár poznámok o Jave a Eclipse

Java ako programovací jazyk

V rámci predmetov PAZ budeme programy písať v programovacom jazyku Java. Javu možno charakterizovať ako moderný, objektovo-orientovaný a platformovo nezávislý programovací jazyk. Syntax jazyka Java je podobná jazyku C. Aj vďaka tomu si Java oveľa ľahšie získala veľkú obľubu u profesionálnych programátorov a silnú pozíciu na trhu programovacích jazykov.

Java ako softvérová platforma

Java však nie je len programovací jazyk – je to celá softvérová platforma. Kľúčovými prvkami tejto platformy sú Java ako programovací jazyk a **JVM (Java Virtual Machine)**. Základný rozdiel oproti iným programovacím jazykom je to, že výsledkom kompilácie nie je priamo spustiteľný program (`exe` súbor v operačnom systéme Windows), ale tzv. **byte-code** (v súboroch s príponou `class`). Byte-code je platformovo nezávislý (je jedno, či používate Windows, Linux, MacOS, Intel, AMD, ...), vykonávaný je prostredníctvom JVM. Vďaka tomuto prístupu je možne spustiť skompilované Java programy (byte-code) na ľubovoľnom zariadení s JVM (počítač s nainštalovaným JRE, mobilný telefón, čipová karta, ...). Len pre zaujímavosť poznamenajme, že motiváciou pre vznik Javy (v rokoch 1991 – 1995 v spoločnosti Sun Microsystems v tíme pod vedením Jamesa Goslinga) bolo programovanie novej generácie inteligentných zariadení (mikrovlnky, chladničky, ...). Rôzne zariadenia totiž bežali na rôznom hardvéri (procesory, atď.), čo v praxi znamenalo vytváranie špecializovaných programov pre každé z hardvérových zariadení. Paradoxne popularitu Jave prinieslo až masívne rozšírenie internetu a webu, kedy došlo k "spojeniu" rôznych typov počítačov (Windows, Unix, Linux, ...) a s tým aj obrovská potreba používateľov na softvér, ktorý by nebol závislý na konkrétnom operačnom systéme. Z histórie Javy ešte spomeňme, že v roku 2010 bola spoločnosť Sun Microsystems (v nej bola Java vytvorená) odkúpená spoločnosťou Oracle.

Dnes je Java dostupná v 4 edíciach:

- **Java Card** - pre "smart" karty (SIM karty, ...)
- **Java Micro Edition** - pre mobilné zariadenia,
- **Java Standard Edition** - pre bežné počítače,
- **Java Enterprise Edition** - pre podnikové a "biznis" aplikácie.

Keďže javovské programy sú vykonávané (interpretované) JVM (Java Virtual Machine), pre ich spustenie na konkrétnom počítači je treba, aby "na tomto počítači bola nainštalovaná Java". Je však treba rozlišovať medzi JRE a JDK:

- *Java Runtime Environment* (JRE) - je kolekcia programov, ktoré umožňujú spúšťať Java programy (kľúčovým programom je JVM - program `java`)
- *Java Development Kit* (JDK) - kolekcia programov, ktoré umožňujú vytvárať (kompilovať) Java programy (kľúčovým programom je Java kompilátor - program `javac`). Súčasťou JDK je aj JRE (povedané matematicky: JDK je nadmnožinou JRE)

Kým väčšine používateľov stačí JRE (pre nich je synonymom Javy), my potrebujeme JDK, keďže Java programy chceme nielen spúšťať, ale aj vytvárať.

Viac o Jave sa môžete dozvedieť na oficiálnych stránkach Javy, resp. na predmetoch počas ďalších rokov štúdia.

Pár reklamných videí o tom, kde všade je možné nájsť Javu a ako ovplyvňuje (a bude ovplyvňovať) fungovanie mnohých aspektov nášho života možno nájsť tu:

- http://www.youtube.com/watch?v=SRLU1bJSLVg
- http://www.youtube.com/watch?v=mWhDAh2Ys6s&videos=esHHNijSsXk

V súčasnosti začal význam Javy opäť narastať v súvislosti s príchodom mobilných aplikácii a platformy Android pre smartphony a tablety od spoločnosti Google. Vývoj androidových aplikácií, ako aj samotný Android, sú totiž postavené na jazyku Java.

Vývojové prostredie Eclipse

Ako vývojové prostredie budeme používať prostredie Eclipse pôvodne od spoločnosti IBM. Je to profesionálne vývojové prostredie, ktoré výrazne uľahčuje všetky aspekty práce programátora. Z nášho pohľadu je dôležité najmä to, že je dostupné zadarmo. Medzi jeho hlavné výhody patrí jeho platformová nezávislosť (beží pod Windows aj Linuxom, je naprogramované v Jave) a množstvo pluginov. V neposlednom rade umožňuje vyvíjať projekty nielen v Jave, ale (po doinštalovaní pluginov) aj v C/C++, PHP, Cobole, či Pythone.

Základným prvkom pri práci s Eclipse je **Workspace** (pracovný priestor). Vo Workspace môžeme mať umiestnených aj niekoľko projektov. Workspace zodpovedná konkrétnemu jednému adresáru (priečinku) na disku. Zmeniť aktívny Workspace môžeme cez menu *File -> Switch Workspace*. Taktiež pri každom spustení Eclipse (pokiaľ si svoje rozhodnutie nenecháme zapamätať) sme vyzvaní zvoliť si aktívny Workspace.

Ďalším dôležitým prvkom pri práci s Eclipse je **Project** (projekt). Každý projekt má svoje meno. Toto meno určuje podadresár v adresári Workspace-u, do ktorého sú ukladané všetky súbory týkajúce sa daného projektu. Vo všeobecnosti možno na projekt pozerať ako na akúsi množinu súvisiacich súborov. Každý projekt môže mať aj svoje individuálne nastavenia (dostupné cez kontextové menu projektu v položke *Properties*).

Častou činnosťou pri práci s Eclipse je **pridávanie** nových "vecí" (tried - Classes, súborov - Files, balíčkov - Packages, ...) do projektu. Tie je možné pridávať cez položku *New* v kontextovom menu projektu. Opäť pripomíname, že všetky tieto vytvorené "veci" sú uložené v adresári projektu.

Inštalácia Javy (JDK) a Eclipse

Návod na inštaláciu Javy a vývojového prostredia Eclipse možno nájsť na podstránke Návod na inštaláciu Javy a Eclipse.

<< | Obsah | Prvý projekt s JPAZ2 frameworkom >>

Prvý projekt s JPAZ2 frameworkom

V prvých týždňoch semestra budeme na programovanie využívať **JPAZ2 framework**. Je to pomocný nástroj, ktorý nám umožní využívať **korytnačiu grafiku** (http://www.youtube.com/watch?v=e1vXYveYj-A) a na intuitívnej báze zoznámiť so základnými konceptami **objektovo-orientovaného programovania**.

Vytvorenie projektu a pripojenie JPAZ2

Ako sme si povedali v predošlej časti, práca v prostredí Eclipse je organizovaná v projektoch. Nasledujúci komentovaný video-tutoriál nám ukáže:

1. ako v prostredí Eclipse vytvoríme projekt s menom `PrvyProjekt`
2. ako k projektu pripojíme knižnicu `jpaz2.jar` (JPAZ2 framework), ktorá nám umožní využívať korytnačiu grafiku
3. ako vytvorím triedu `Spustac`, ktorá bude obsahovať "metódu `main`" - príkazy, ktoré sú v nej napísané, sa postupne jeden za druhým v poradí, ako sú napísané, vykonajú po spustení triedy `Spustac`
4. ako spustiť triedu `Spustac`

 Vytvorenie projektu s JPAZ2 (video-tutoriál)

Projekt v Eclipse je v podstate len kopa súborov, ktoré patria k sebe. Keďže Java je objektovo orientovaná, najdôležitejšie pojmy pri tomto prístupe sú objekt a trieda (čo to je, si vysvetlíme neskôr, odporúčame však nateraz zabudnúť na všetky asociácie, ktoré by vám v súvislosti s pojmom trieda mohli napadnúť). V túto chvíľu nám stačí vedieť, že súbory, ktoré budeme v projekte vytvárať, aby sme do nich písali príkazy (zdrojový kód), budeme volať triedy. Predošlý video-tutoriál demonštruje vytvorenie triedy `Spustac`. Ako už názov napovedá, bude to trieda ("súbor"), určená výhradne na spúšťanie nami vytvorených programov. Aby sme ju odlíšili od iných, budeme ju volať stále rovnako - `Spustac` (samozrejme, Jave je v skutočnosti jedno, ako si ju nazveme). Od iných tried, ktoré budeme vytvárať, sa bude líšiť tým, že bude obsahovať metódu `main`. Čo to je metóda, si taktiež vysvetlíme neskôr. Nateraz je to čosi, čo sa nám samo v triede `Spustac` vygenerovalo po zakliknutí zaškrtávacieho políčka vo formulári pri vytváraní triedy. Dôležité je pamätať si, že všetky príkazy, ktoré napíšeme medzi "kučeravé" zátvorky {} za `main`, sa vykonajú po spustení triedy `Spustac` v takom poradí, ako sú tam napísané.

ObjectInspector a WinPane

Pozrime sa, čo sa stane, ak spustíme takto upravenú triedu `Spustac` (príkaz `JPAZUtilities.sunDemo();` sme nahradili 3 inými príkazmi - každý v samostatnom riadku ukončenom bodkočiarkou):

```
import sk.upjs.jpaz2.*;

public class Spustac {

        /**
         * @param args
         */
        public static void main(String[] args) {
                WinPane plocha = new WinPane();
                ObjectInspector oi = new ObjectInspector();
```

```
                oi.inspect(plocha);
        }

}
```

Jednotlivé príkazy možno slovne opísať nasledovným spôsobom:

1. Vytvor nový objekt **triedy** (=typu) `WinPane` vo "svete objektov" a zároveň vytvor premennú s menom `plocha`, ktorá je schopná komunikavať s novovytvoreným objektom triedy `WinPane`. Slovíčko `WinPane` pred slovíčkom `plocha` znamená, že vytvorená premenná `plocha` vie komunikovať len s objektami triedy `WinPane`. Zároveň slovíčko **new** (nový) a za ním `WinPane` hovorí, že vytvárame nový objekt triedy `WinPane`. Symbol `=` v strede príkazu je príkazom priradenia - ním sa vytvorený objekt prepojí s premennou `plocha`.
2. Analogicky, ako v kroku 1, vytvoríme nový objekt triedy `ObjectInspector` a premennú s menom `oi`, cez ktorú budeme môcť komunikovať s novovytvoreným objektom triedy `ObjectInspector`.
3. Cez premennú `oi` povieme `ObjectInspectoru` vytvorenému v bode 2, aby skúmal objekt, s ktorým komunikujeme cez premennú `plocha` (a my vieme, že to je objekt triedy `WinPane` vytvorený v kroku 1).

Pozrime sa na správanie vytvoreného programu:

 Plocha a inšpektor objektov (video-tutoriál)

Ako môžeme vidieť vo video-tutoriále, objekt triedy `WinPane` je vlastne len obyčajné prázdne okienko. Naopak vytvorený objekt triedy `ObjectInspector` je "zložitejšie" okienko. Vidíme, že `ObjectInspector` zobrazuje rôzne informácie o skúmanom objekte (napr. šírku, výšku, farbu, ...) a tieto vlastnosti navyše umožňuje aj meniť (editovať). Informácie o skúmanom objekte sú v záložke **Properties** (Vlastnosti). Zároveň v záložke **Methods** vidíme, že objekt má nielen nejaké vlastnosti, ale aj metódy. Po zadaní potrebných parametrov (upresnenie, čo chceme) vieme metódu aj spustiť. Spustenie metódy zvyčajne povedie k tomu, že čosi sa nejako zmení. V túto chvíľu vieme, že objekty majú nejaké **vlastnosti** popisujúce **stav objektu** a **metódy** vykonávajúce nejaké "akcie".

Turtle - prvá korytnačka

Pridaním ďalších príkazov upravme triedu:

```
import sk.upjs.jpaz2.*;

public class Spustac {

        /**
         * @param args
         */
        public static void main(String[] args) {
                WinPane plocha = new WinPane();

                ObjectInspector oi = new ObjectInspector();
                oi.inspect(plocha);

                Turtle jozko = new Turtle();
                plocha.add(jozko);
                oi.inspect(jozko);
        }

}
```

Pridaním 3 riadkov príkazov sme analogicky ako v prvom príklade vyrobili objekt triedy `Turtle` a povedali sme `ObjectInspector`-u komunikovanému cez premennú `oi`, aby skúmal vytvorený objekt. Novinkou je príkaz:

```
plocha.add(jozko);
```

Ním povieme objektu triedy `WinPane` komunikovanému cez premennú `plocha`, aby si "adoptoval" vytvorený objekt triedy `Turtle` komunikovaný cez premennú `jozko`. Alebo ináč, do plochy pridáme vytvorenú korytnačku. Poznamenajme, že v informatickej terminológii sa namiesto frázy "objekt komunikovaný cez premennú X" používa fráza "objekt **referencovaný** premennou X" alebo niekedy aj skrátene "objekt X". My od tejto chvíle budeme používať túto novú terminológiu.

Po krátkom experimentovaní s príkladom (aktuálne skúmaný objekt v `ObjectInspector`-e možno zmeniť v zozname umiestnenom úplne hore) môžeme pozorovať, že objekt triedy Turtle (korytnačka) má mnoho vlastností a ešte viacej metód. Typickými vlastnosťami korytnačky sú `x` a `y`, ktoré určujú súradnice korytnačky v ploche, v ktorej "žije". Korytnačka má aj ďalšie vlastnosti:

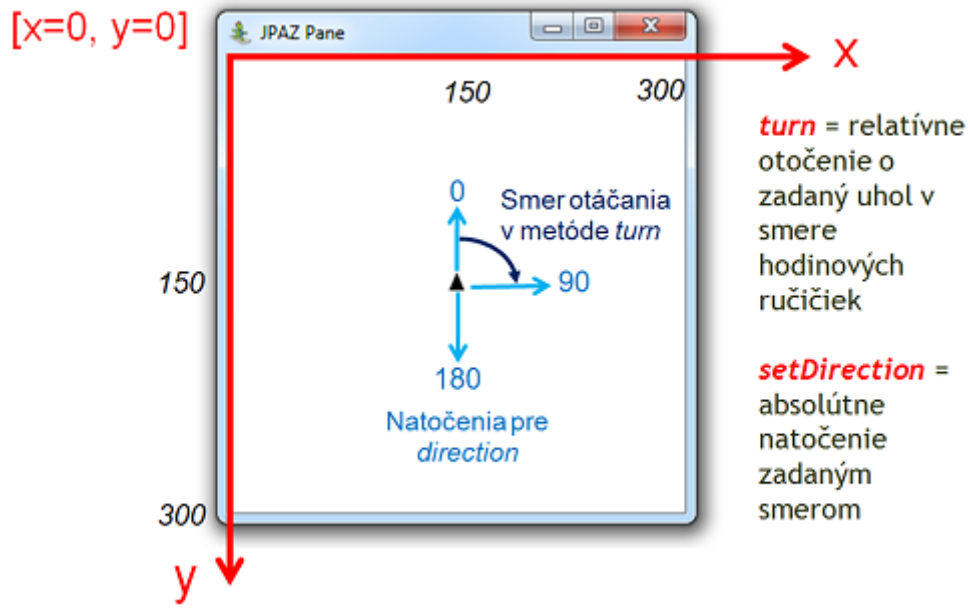
- `direction` – smer natočenia v stupňoch. Smer 0 zodpovedá smeru hore.
- `penColor` – farba kresliaceho pera
- `penWidth` – hrúbka kresliaceho pera
- `penDown` – hovorí, či je kresliace pero aktívne
- `visible` – či je korytnačka vidieľná
- `rangeStyle` – spôsob, ako sa korytnačka správa pri okrajoch plochy

Spomedzi metód sú najdôležitejšie metódy tieto:

- `step` – korytnačka spraví krok zadanej dĺžky a ak je kresliace pero aktívne, kreslí aj čiaru
- `turn` – korytnačka sa otočí o zadaný uhol v smere hodinových ručičiek
- `penUp` – deaktivuje kresliace pero
- `penDown` – aktivuje kresliace pero
- `moveTo` – korytnačka sa presunie na zadanú pozíciu a ak je kresliace pero aktívne, kreslí aj čiaru
- `setPosition` – korytnačka zmení svoju pozíciu (čiara sa nekreslí nikdy)
- `dot` – nakreslí bodku s daným polomerom
- ďalšie korytnačie metódy možno nájsť na stránke s ich sumarizáciou (v slovenčine).

Všimnime si, že niektoré metódy len čosi spravia, iné vypočítajú aj nejaký "Result" (výsledok). Niektoré metódy nemajú parametre, iné parametrom majú viacero. Pomocou parametrov akoby upresňujeme, ako presne sa má príkaz vykonať. Jednotlivé parametre majú svoje meno. Ďalej každý parameter, výsledok, ale i vlastnosť majú svoj typ (`double`, `int`), ktoré akoby určujú povolené hodnoty (napr. ak sa niekde čaká `double`, nemôžeme tam napísať nejaký "nečíselný" text). Zároveň si všimnime, že niektoré metódy a vlastnosti sú si akosi podobné (napr. vlastnosť `x` a metódy `setX` a `getX`). Dá sa pozorovať, že ak máme vlastnosť s menom `vlastnost`, tak metóda `getVlastnost` nám vráti (vypočíta) aktuálnu hodnotu tejto vlastnosti. Naopak metóda `setVlastnost` má práve jeden parameter a jej zavolaním zmeníme hodnotu vlastnosti na hodnotu určenú parametrom. Môžeme sa tiež utvrdiť v tom, čo sme pozorovali pri úvodnom skúmaní plochy. Metódy sú len príkazy pre daný objekt (môžu mať upresňujúce parametre, či môžu poskytnúť nejaký výsledok).

Dôležité je si zapamätať, ako vyzerá **súradnicový systém plochy**. V ľavom hornom rohu je bod s x-ovou súradnicou 0 a y-vou súradnicou 0 (x=0, y=0). X-ová súradnica rastie zľava doprava a Y-ová zhora nadol.



Po chvíli experimentovania ľahko prídeme na postup, ktorým korytnačka namaľuje rovnostranný trojuholník s dĺžkou strany 100:

- spusti metódu `step` s parametrom `100`
- spusti metódu `turn` s parametrom `120`
- spusti metódu `step` s parametrom `100`
- spusti metódu `turn` s parametrom `120`
- spusti metódu `step` s parametrom `100`
- spusti metódu `turn` s parametrom `120` (ak chceme, aby korytnačka bola nasmerovaná tak, ako na začiatku)

Práca s objektami z Java programu

Skúsme, čo sa stane, ak do našej triedy `Spustac` pridáme ďalší príkaz:

```
import sk.upjs.jpaz2.*;

public class Spustac {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		WinPane plocha = new WinPane();

		ObjectInspector oi = new ObjectInspector();
		oi.inspect(plocha);

		Turtle jozko = new Turtle();
		plocha.add(jozko);
		oi.inspect(jozko);

		jozko.center();
	}
}
```

Hneď po spustení triedy môžeme pozorovať, že korytnačka sa presunula do stredu svojej domovskej kresliacej plochy. Inými slovami, teraz už máme spôsob, ako komunikovať s korytnačkou z nášho Java programu. Dokonca si ľahko domyslíme spôsob, ako spúšťať metódy ľubovoľného objektu. Najprv napíšeme meno premennej, ktorá referencuje objekt, ktorého metódu chceme spustiť (odborne povieme, že **volať**

metódu). Potom nasleduje bodka a za ňou napíšeme meno metódy, ktorú chceme spustiť. Nakoniec za meno metódy pridáme 2 okrúhle zátvorky () a za ne dáme bodkočiarku. Treba si zapamätať, že v uvedenom postupe nikdy nepíšeme medzeru. Pozor, takto to ale nestačí! Ak má naša metóda nejaké parametre, musíme ich hodnoty napísať medzi zátvorky. Jednotlivé parametre oddeľujeme čiarkou. Príklady:

```
jozko.stamp();
jozko.setPenWidth(2.5);
jozko.setX(300);
jozko.setPosition(300, 200);
plocha.resize(300, 200);
```

Skúsme napísať príkazy, ktoré namaľujú trojfarebný rovnostranný trojuholník.

```
import sk.upjs.jpaz2.*;

public class Spustac {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		WinPane plocha = new WinPane();
		ObjectInspector oi = new ObjectInspector();
		oi.inspect(plocha);
		Turtle jozko = new Turtle();
		plocha.add(jozko);
		oi.inspect(jozko);

		jozko.center();

		jozko.setPenColor(Color.red);
		jozko.step(100);
		jozko.turn(120);
		JPAZUtilities.delay(1000);

		jozko.setPenColor(Color.blue);
		jozko.step(100);
		jozko.turn(120);
		JPAZUtilities.delay(1000);

		jozko.setPenColor(Color.green);
		jozko.step(100);
		jozko.turn(120);
	}
}
```

V príklade sme použili aj príkaz `JPAZUtilities.delay(1000);` Tento príkaz spôsobí pozastavenie programu na zadaný čas v milisekundách (1000 milisekúnd je 1 sekunda).

Všimnite si, že je veľmi dôležité nezabudnúť **napísať bodkočiarku** za každým napísaným príkazom.

No a na úplný záver si môžeme vyskúšať vytvoriť ďalšiu korytnačku a umiestniť ju do plochy.

```
import sk.upjs.jpaz2.*;

public class Spustac {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		WinPane plocha = new WinPane();
		ObjectInspector oi = new ObjectInspector();
```

```
                oi.inspect(plocha);
                Turtle jozko = new Turtle();
                plocha.add(jozko);
                oi.inspect(jozko);

                jozko.center();

                Turtle katka = new Turtle();
                plocha.add(katka);
                oi.inspect(katka);

                katka.setPosition(100, 50);
                katka.setPenColor(Color.green);
        }
}
```

Pozorujme, čo sa stane po spustení programu. Vidíme, že v ploche už máme 2 objekty - korytnačky. Každá má rovnaký zoznam vlastností aj metód. Avšak hodnoty jednotlivých vlastností (napr. pozícia, farba) sa líšia. Ak zadáme korytnačke nejaký príkaz zavolaním metódy, obe korytnačky vykonajú to isté (samozrejme s ohľadom na aktuálne hodnoty vlastností konkrétnej korytnačky).

Sumarizácia

Prvé nepresné predstavy a definície na základe doterajších pozorovaní správania sa vytvorených programov:

- **trieda** - typ/druh čohosi (napr. telefón "HTC Desire", "Škoda Fabia"). Ak ako triedu zoberieme "Škoda Fabia", tak táto trieda je presne určená technickou špecifikáciou a konštrukčnými výkresmi.
- **objekt** - konkrétna "vec" nejakého typu (triedy). Napríklad ak ako triedu zoberieme "Škoda Fabia", tak každé jedno konkrétne auto tohto typu je objektom triedy "Škoda Fabia". Všetky objekty sú konštrukčne navlas rovnaké, keď vyjdú z výrobnej haly (keď sa akoby spraví `new Fabia()`). Neskôr, keď autá "žijú svoj život", majú rôznych šoférov, nastriekajú ich na rôznu farbu, nachádzajú sa na inom mieste - menia sa ich vlastnosti.
- **vlastnosť** - niečo, čo charakterizuje aktuálny stav konkrétneho objektu. Rôzne objekty rovnakej triedy majú rovnaké vlastnosti (napr. stav nádrže), ale konkrétna hodnota vlastností môže byť iná (napr. jedno auto ma v nádrži 10l paliva, iné 20l paliva).
- **metóda** - nejaký príkaz, ktorý môžeme dať konkrétnemu objektu. Aké metódy ("akým príkazom rozumie") má konkrétny objekt a čo sa stane, keď objekt požiadame, aby ich vykonal, je presne určené tým, akej triedy je objekt.
- **parameter metódy** - má meno a typ. Typ určuje, aká hodnota (číslo, farba, ...) môže byť zadaná pre daný parameter. Samotné parametre slúžia na upresnenie toho, ako sa má metóda vykonať (napr. o koľko presne sa má korytnačka posunúť dopredu).

<< Pár poznámok o Jave a Eclipse | Obsah | Pridávanie vlastných metód - rozširovanie tried >>

Pridávanie vlastných metód - rozširovanie tried

Vylepšovanie existujúcich dobrých vecí je v bežnom živote užitočná stratégie. Určite je lepšie niečo dobré vylepšiť, než začať vytvárať niečo nové úplne od základov. A táto stratégia je jeden z najdôležitejších konceptov objektovo orientovaného programovania.

Doteraz nami vytvárané korytnačky, ktoré prišli do nášho programu z JPAZu (tam je nejako napísané, aké vlastnosti a metódy má korytnačka, ako sa kreslia čiary, ako sa otáča, ...), vedia toho celkom dosť. Radi by sme však v našich programoch mali "chytrejšie" korytnačky, ktoré toho vedia ešte viacej. Ako ale získať "chytrejšie" korytnačky? Jedna z inšpirácii je šľachtenie rastlín - zoberieme nejakú odrodu rastlín, "čosi porobíme", a dostaneme novú lepšiu odrodu.

Teraz sa naučíme postup, ktorým môžeme vytvoriť nový - lepší druh korytnačiek. Taký druh, že korytnačky tejto novej triedy budú poznať nejaké metódy navyše. Novú triedy spravíme rozšírením existujúcej triedy `Tutle`. Triedu si možno predstaviť ako akúsi **šablónu**, vzor – "genetickú informáciu", ktorú pri narodení (vytvorení) dostáva objekt danej triedy. Táto šablóna **predpisuje** aké **metódy** má objekt (aké "príkazy" pozná) a ako sa objekt **správa** pri konkrétnych metódach.

 Vytvorenie vlastnej rozšírenej triedy s novými metódami (video-tutoriál)

Pripomeňme, že v tomto postupe sme vytvorili novú triedu s menom `MojaKorytnacka` pričom sme pri jej vytváraní povedali prostrediu Eclipse, aby táto nová trieda rozširovala triedu `Turtle`. Inými slovami, aby každý vytvorený objekt triedy `MojaKorytnacka` vedel nielen všetko čo bežné korytnačky, ale aj niečo navyše. V príklade sme naučili objekty triedy `MojaTurtle` novú metódu, ktorá nakreslila trojuholník:

```
import sk.upjs.jpaz2.Turtle;

public class MojaKorytnacka extends Turtle {

        public void trojuholnik() {
                this.step(100);
                this.turn(120);
                this.step(100);
                this.turn(120);
                this.step(100);
                this.turn(120);
        }
}
```

Všimnime si, že v jednotlivých príkazoch používame slovíčko `this`. Toto slovíčko hovorí, že chceme volať metódu samého seba. `This` si môžeme predstaviť ako slovíčko "ja" (je to forma uvedomenia objektu). Pri písaní metód treba mať stále na zreteli, že to, čo programujeme, je šablóna "správania" pre **všetky** objekty vytváranej triedy. A práve slovíčko `this` reprezentuje ten objekt, ktorý sa v danom okamihu správa podľa tejto šablóny.

Aby sme naše "novo vyšľachtené plemeno" korytnačiek vyskúšali, musíme upraviť kód v triede `Spustac`. Pridáme teda pár už známych príkazov, ktoré vytvoria premennú referencujúcu objekt triedy `MojaKorytnacka` a vytvorený objekt nechajú skúmať `ObjectInspector`-om:

```
import sk.upjs.jpaz2.*;

public class Spustac {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		WinPane plocha = new WinPane();
		ObjectInspector oi = new ObjectInspector();
		oi.inspect(plocha);

		Turtle jozko = new Turtle();
		plocha.add(jozko);
		oi.inspect(jozko);

		jozko.setPosition(50, 50);

		MojaKorytnacka katka = new MojaKorytnacka();
		plocha.add(katka);
		oi.inspect(katka);

		katka.center();
		katka.trojuholnik();
	}
}
```

Po spustení programu môžeme v `ObjectInspector`-e vidieť, že objekt triedy `MojaKorytnacka` (`katka`) pozná naučenú metódu `trojuholnik`. Je dôležité si všimnúť, že v objektoch triedy `Turtle` (`jozko`) sa nič nezmenilo. Samozrejme aj v Java programe už môžeme volať novopridané metódy u objektov triedy `MojaKorytnacka`.

Pri programovaní nových metód môžeme využívať nielen metódy z triedy `Turtle`, ale aj novonaučené metódy z triedy `MojaKorytnacka` (viď metóda `zahada`):

```
import sk.upjs.jpaz2.Turtle;

public class MojaKorytnacka extends Turtle {

	public void trojuholnik() {
		this.step(100);
		this.turn(120);
		this.step(100);
		this.turn(120);
		this.step(100);
		this.turn(120);
	}

	public void zahada() {
		this.trojuholnik();
		this.turn(120);
		this.trojuholnik();
		this.turn(120);
		this.trojuholnik();
		this.turn(120);
	}
}
```

Poznámka: Pri programovaní vlastných "maľovacích" metód pre korytnačky je dobré dodržiavať pravidlo, že korytnačka je po skončení metódy na tej pozícii a v takom smere, ako bola pred vykonaním metódy.

Vlastné metódy s parametrom

Z experimentovania s `ObjectInspector`-om, či z písania kódu, vieme, že metódy môžu mať aj parametre. Tie do novopridaných metód definovaných v triede `MojaKorytnacka` pridáme veľmi jednoducho. Stačí medzi okrúhle zátvorky pri názve metódy pridať zoznam parametrov, ktoré má mať táto metóda. V tomto zozname sú informácie o jednotlivých parametroch oddelené čiarkou. Každý jeden **parameter** je definovaný dvojicou **typ** a **názov parametra**. Typ parametra je magické slovíčko, ktoré hovorí, aké sú povolené hodnoty pre parameter. Napríklad pre typ **double** je povolená hodnota ľubovoľné reálne číslo. Meno parametra je označenie parametra vo vnútri našej metódy. Vo všetkých príkazoch v metóde môžeme použiť namiesto konkrétnej hodnoty (100, 200, 3.4) aj parameter. Meno parametra vo vnútri metódy zastupuje konkrétnu hodnotu, ktorá bola zadaná pri volaní (spustení) tejto metódy.

Pozrime sa, ako bude vyzerať metóda trojuholník, ak ju zmeníme tak, že bude vyžadovať zadanie parametre určujúceho dĺžku strany kresleného rovnostranného trojuholníka.

```
import sk.upjs.jpaz2.Turtle;

public class MojaKorytnacka extends Turtle {

        public void trojuholnik(double dlzkaStrany) {
                this.step(dlzkaStrany);
                this.turn(120);
                this.step(dlzkaStrany);
                this.turn(120);
                this.step(dlzkaStrany);
                this.turn(120);
        }

        public void zahada() {
                this.trojuholnik(100);
                this.turn(120);
                this.trojuholnik(100);
                this.turn(120);
                this.trojuholnik(100);
                this.turn(120);
        }
}
```

A ešte metóda na nakreslenie obdĺžnika s parametrami určujúcimi jeho šírku a výšku.

```
public void obdlznik(double sirka, double vyska) {
        this.step(vyska);
        this.turn(90);
        this.step(sirka);
        this.turn(90);
        this.step(vyska);
        this.turn(90);
        this.step(sirka);
        this.turn(90);
}
```

Pravidlá pre pomenovania

Parametre, metódy a triedy si môžeme pomenovať skoro ľubovoľným spôsobom. Z dôvodu čitateľnosti však budeme používať tieto pravidlá:

- Názvy tried: Názov triedy začína veľkým písmenom, všetky ďalšie písmena sú malými písmenami abecedy. V prípade, že chceme použiť viacslovný názov, tak názov zlepíme do jedného slova a každé písmeno, ktoré zodpovedá začiatku slova zapíšeme veľkým písmenom: MojaKorytnacka, MojaTurtle, MojaNajmudrejsiaKorytnacka.

- Názvy metód a parametrov: úplne rovnaké pravidlá ako pre názvy tried s jedinou výnimkou - prvé písmeno názvu je vždy malé: setPenColor, width, dlzkaStrany.

<< Prvý projekt s JPAZ2 frameworkom | Obsah | Opakovanie skupiny príkazov (pevný počet opakovaní) >>

Opakovanie skupiny príkazov (pevný počet opakovaní)

Mnohé naše doposiaľ vytvorené metódy v triede `MojaKorytnacka` majú tú vlastnosť, že istá skupina navlas rovnakých príkazov sa v nich niekoľko krát za sebou opakuje. Napríklad taký trojuholník:

```
public void trojuholnik(double dlzkaStrany) {
        this.step(dlzkaStrany);
        this.turn(120);
        this.step(dlzkaStrany);
        this.turn(120);
        this.step(dlzkaStrany);
        this.turn(120);
}
```

V metóde `trojuholnik` ľahko identifikujeme, že 6 príkazov, ktoré ju tvoria, je vlastne 3 krát zopakovaná dvojica príkazov:

```
this.step(dlzkaStrany);
this.turn(120);
```

Opakovanie je jedným z dôležitých princípov programovania. Preto aj Java ponúka (nateraz) magickú formulku, ktorou vieme zabezpečiť zopakovanie nejakej skupiny príkazov. Pozrime sa ako bude vyzerať metóda na namaľovanie trojuholníka s využitím tejto formulky.

```
public void trojuholnik(double dlzkaStrany) {
        for (int i = 0; i < 3; i++) {
                this.step(dlzkaStrany);
                this.turn(120);
        }
}
```

V tejto formulke je dôležité zapamätať si, že skupinu príkazov, ktorá sa má opakovať, píšeme medzi kučeravé zátvorky { a } tejto formulky. Ďalšia vec na zapamätanie je, že za znak < píšeme, koľko krát sa má skupina príkazov zopakovať. Vo všeobecnosti môže túto formulku zapísať takto:

```
for (int i = 0; i < kolkoKratSaMajuPrikazyOpakovat; i++) {
        // príkazy, ktoré sa maju opakovať
}
```

Pozrime sa, ako by vyzerala metóda na namaľovanie štvorca s využitím tejto opakovacej formulky:

```
public void stvorec(double dlzkaStrany) {
        for (int i = 0; i < 4; i++) {
                this.step(dlzkaStrany);
                this.turn(90);
        }
}
```

A ešte metóda pre obdĺžnik:

```
public void obdlznik(double sirka, double vyska) {
        for (int i = 0; i < 2; i++) {
```

```
                this.step(vyska);
                this.turn(90);
                this.step(sirka);
                this.turn(90);
        }
}
```

Ešte pred úplným záverom sa pozrime na metódu, ktorá namaľuje jednoduchú 12-cípu vločku s parametrom definovanou dĺžkou lúča.

```
public void vlocka(double dlzkaLuca) {
        for (int i = 0; i < 12; i++) {
                this.step(dlzkaLuca);
                this.step(-dlzkaLuca);
                this.turn(30);
        }
}
```

V príklade sme využili fintu, že ak do príkazu `step` dáme ako parameter záporné číslo, korytnačka bude cúvať.

Čo si treba pamätať?

- **Ak chceme zopakovať vykonanie skupiny príkazov nejaký počet krát, môžeme použiť formulku na opakovanie (nazývanú aj "príkaz for" alebo "for-cyklus"):**

  ```
  for (int i = 0; i < pocetOpakovani; i++) {
          // tu napíšeme príkazy, ktoré sa maju opakovať
  }
  ```

<< Pridávanie vlastných metód - rozširovanie tried | Obsah | Opakovanie skupiny príkazov (variabilný počet opakovaní) >>

Opakovanie skupiny príkazov (variabilný počet opakovaní)

V predchádzajúcej kapitole sme sa dozvedeli ako zapísať to, že nejaká skupina príkazov sa má zopakovať zadaný počet krát. Toto vieme použiť na kreslenie štvorca, 12-cípej hviezdy (alebo vločky?), pravidelného 6-uholníka alebo iného pravidelného útvaru. Čo ale v prípade, keď potrebujeme nakresliť všeobecnejšie definovaný obrazec ako je napríklad `n`-cípa hviezda (vločka) alebo pravidelný `n`-uholník, kde `n` nie je vopred dané, ale je určené ako parameter metódy?

Zamyslime sa najprv, ako by mala vyzerať **hlavička metódy** (riadok hovoriaci o názve metód a jej parametroch). Doposiaľ sme sa stretli iba sa parametrami typu `double` - povolená hodnota bola číslo s desatinnou čiarkou ("reálne číslo"). Tento typ "povolenej" hodnoty pre parameter `n` nie je veľmi vhodný. Ako by totiž mala vyzerať `5.5`-cípa hviezda alebo pravidelný `6.35`-uholník? Ak chceme obmedziť hodnotu parametra na celé čísla, namiesto typu `double` použijeme typ `int`, ktorý dovoľuje len celé čísla. Iste tušíte, že aj tu je jeden menší problém. Keďže aj záporné čísla sú celé čísla, aj záporná hodnota je povolená ako hodnota parametra `n`. Toto však nebudeme teraz riešiť a na chvíľu sa spoľahneme, že používateľ bude zadávať len kladné čísla.

Vytváranie metódy na nakreslenie `n`-cípej hviezdy začneme napísaním "príkazov", ktorými povieme, že všetky objekty triedy `MojaKorytnacka` majú vedieť nakresliť pravidelnú `n`-cípu hviezdu (zatiaľ ale nepovieme ako):

```
public void nHviezda(int n, double dlzkaLuca) {

}
```

Čím sa líši nakreslenie `12`-cípej hviezdy od `n`-cípej? Jednoduchou úvahou prídeme na to, že

- potrebujeme nakresliť `n` lúčov (cípov) namiesto 12,
- uhol, ktorý zvierajú lúče nie je `360/12=30`, ale v prípade `n` lúčov sa uhol `360` ° rozdelí na `n` rovnakých úsekov, t.j. uhol každého z nich bude `360/n`.

Pri programovaní metódy `nHviezda` teraz využijeme, že vo formulke na opakovanie skupiny príkazov ("for-cykle") môžeme napísať ako počet opakovaní nielen konkrétne číslo, ale aj meno parametra. Dôsledkom bude to, že formulka zabezpečí opakovanie skupiny príkazov taký počet krát, akú hodnotu aktuálne zastupuje zadaný parameter.

Výsledkom bude nasledujúca metóda:

```
public void nHviezda(int n, double dlzkaLuca) {
        for (int i = 0; i < n; i++) {
                this.step(dlzkaLuca);
                this.step(-dlzkaLuca);
                this.turn(360/n);
        }
}
```

Takto naprogramovaná metóda bude takmer dobrá. Vždy sa nakreslí zadaných `n` lúčov. Ak budeme trochu experimentovať s hodnotou parametra `n`, ľahko prídeme na to, že ak zadáme počet lúčov rovný 100, nakreslí sa síce 100 lúčov (môžete ich skúsiť spočítať...), no uhly medzi nimi nebudú rovnaké. Ako tento problém

vyriešiť si vysvetlíme neskôr. Každopádne však vieme, že príkazy sa vykonali 100 krát a to nám nateraz stačí.

Čo si treba pamätať?

- **Ako počet opakovaní skupiny príkazov pomocou príkazu "for" (opakovacej formulky) môžeme uviesť nielen konkrétne číslo, ale aj meno parametra. Aktuálna hodnota tohto parametra potom určuje, koľko krát sa skupina príkazov vykoná.**
- **Ak ako typ parametra uvedieme `int`, ako hodnoty parametra budú dovolené len celé čísla.**

<< Opakovanie skupiny príkazov (pevný počet opakovaní) | Obsah | Premenné >>

Premenné

Premenné (pozor, nezamieňať si s premennými v matematike) slúžia na uloženie jednej hodnoty. Najčastejšie použitie premenných pri programovaní je dočasné uloženie si nejakej (napr. vypočítanej) hodnoty na neskoršie použitie.

Premenné:

- sú pomenované (majú meno),
- majú svoj typ určujúci hodnoty, ktoré je dovolené v premennej uložiť.

Premennú si môžeme predstaviť ako nejaké **pomenované úložisko** (skrinka, "šuflík", papierik) na odloženie hodnôt (napr. čísel).

Premenné, ktoré vzniknú počas vykonávania metód voláme **lokálne premenné**. V tejto kapitole sa budeme venovať iba lokálnym premenným.

Vytvorenie premennej

Na to, aby sme s nejakou premennou mohli pracovať, musíme si ju najprv vytvoriť - **deklarovať**. Premenné sa vytvárajú príkazom, ktorý nazývame **deklarácia premennej**. V tomto príkaze uvedieme typ a názov premennej, ktorú chceme vytvoriť. Napríklad, ak chceme vytvoriť (deklarovať) premennú s menom `suradnicaX`, ktorá má byť schopná uchovať jedno číslo s desatinnou čiarkou, napíšeme príkaz:

```
double suradnicaX;
```

Prvé "slovo" v príkaze deklarácie premennej je teda **typ** vytváranej premennej a druhé slovo je **meno** vytváranej premennej. Ak by sme chceli premennú schopnú uchovávať celé čísla, vytvorili by sme ju analogicky príkazom `typ meno;`:

```
int pocetKrokov;
```

Deklaráciu premennej si môžeme predstaviť ako príkaz, ktorý kdesi v pamäti počítača vytvorí premennú s definovaným názvom a schopnú uchovávať hodnoty definovaného typu. Premenná je po vytvorení (deklarovaní) **neinicializovaná**. To je stav, kedy v premennej nie je ešte uložená žiadna hodnota. Zapamätajme si, že kým premennú neinicializujeme, nemôžeme s ňou pracovať. Po deklarácii premennej (premenných) teda musíme vykonať inicializáciu - uloženie prvej (tzv. inicializačnej) hodnoty do premennej.

Priradenie hodnoty

Keď už máme premennú vytvorenú, môžeme do nej uložiť (uschovať) nejakú hodnotu. To je možné spraviť **príkazom priradenia**. Ak chceme do (už vytvorenej) premennej `suradnicaX` uložiť hodnotu `10.5` spravíme to takto:

```
suradnicaX = 10.5;
```

Príkaz priradenia sa teda skladá z týchto častí:

- názvu premennej, do ktorej priraďujeme (uchovávame) hodnotu

- znaku `=`
- hodnoty, ktorú chceme premennej priradiť (konkrétna hodnota alebo výraz)
- znaku `;`

Pamätajme si, že príkazom priradenia:

- sa premenná inicializuje, ak bola neinicializovaná,
- hodnota v nej uložená sa prepíše novou (priraďovanou) hodnotou, ak v premennej už predtým bola nejaká hodnota uložená.

Do premennej je možné uložiť len hodnotu, ktorá je dovolená pre daný typ premennej. Napríklad do premennej typu `int` nemôžeme priradiť číslo s desatinnou čiarkou.

Keďže v praxi skoro vždy za príkazom deklarácie premennej nasleduje príkaz priradenia (do vytvorenej premennej si potrebujeme uložiť nejakú hodnotu), Java umožňuje aj skombinovanie týchto dvoch príkazov.

Namiesto dvoch príkazov

```
int pocetKrokov;
pocetKrokov = 100;
```

stačí napísať jeden

```
int pocetKrokov = 100;
```

Použitie hodnoty premennej

Ak chceme v niektorom z príkazov namiesto konkrétnej hodnoty použiť hodnotu, ktorá je aktuálne uložená v nejakej premennej, spravíme to tak, že na danom miestne namiesto konkrétnej hodnoty napíšeme názov premennej:

```
double dlzkaKroku=100.7;
this.step(dlzkaKroku);
```

Stačí si len pamätať, že názov premennej vždy zastupuje hodnotu aktuálne uloženú v danej premennej (tak ako to bolo pri parametroch). Dokonca aj **parametre metód** sú len obyčajné premenné, ktoré sú inicializované na základe hodnôt, s ktorými sa metóda volá. Teda všetko to, čo platí pre premenné, platí aj pre parametre.

Aké môžu byť typy premenných?

Premenné v Jave môžu byť **primitívneho typu** (uchovávajú nejakú hodnotu) alebo **referenčného typu** (uchovávajú referenciu na objekt - zatiaľ sme tieto premenné nazývali komunikátormi). V Jave existuje len 8 premenných primitívneho typu. Zatiaľ sme sa stretli s typmi `double` a `int`. Pozrime sa aj na ďalšie:

reprezentujú	**typ**	**možné hodnoty**	**počet bitov**
celé čísla	byte	-128 až 127	8
	short	-32 768 až 32 767	16
	int	-2 147 483 648 až 2 147 483 647	32

	long	-9 223 372 036 854 775 808 až 9 223 372 036 854 775 807	64
reálne čísla	float	(-16 777 216 až 16 777 216)*$2^{(-126 \text{ až } 127)}$	32
	double	(-9 007 199 254 740 992 až 9 007 199 254 740 992)*$2^{(-1022 \text{ až } 1023)}$	64
pravdivosť	boolean	**true** alebo **false**	1
znaky	char	všetky možné znaky kódovania známeho ako UTF-8 napríklad: **'A'** , **'č'** , **'4'** , **','** (čiarka), **' '** (medzera) a podobne	16

Premenným typu `boolean` a `char` sa budeme venovať neskôr. Na tomto mieste ich uvádzame len kvôli úplnosti. Všimnime si, že Java ponúka viacero typov na uloženie čísel s aj bez desatinnej čiarky. Dôležité je všimnúť si, že v nich nemôžeme uložiť akékoľvek čísla. Každý z typov má definovaný povolený rozsah hodnôt. Dôvod, prečo je v Jave niekoľko typov, je veľmi jednoduchý. Každá premenná zaberá nejakú pamäť v počítači. Tá je však obmedzená. Aby sa pamäťou neplytvalo, programátor si môže vybrať, aký typ premennej použije. Platí, že čím väčší rozsah ponúka daný typ, tým viac pamäti zaberie premenná tohto typu v pamäti počítača.

Kedy premenná zanikne?

Pre lokálne premenné platí, že premenná zanikne akonáhle vykonávanie metódy opustí blok príkazov, v ktorom bola premenná deklarovaná. Blok príkazov je ohraničený príslušným párom zložených ("kučeravých") zátvoriek {}.

<< Opakovanie skupiny príkazov (variabilný počet opakovaní) | Obsah | Aritmetické výrazy >>

Aritmetické výrazy

Do premenných nemusíme ukladať iba konkrétne hodnoty. Oveľa častejšie do premenných ukladáme hodnoty iných premenných a ešte častejšie výsledky nejakých výpočtov. Podobne, keď voláme metódy, ako parameter môžeme uviesť nielen konkrétnu hodnotu, názov premennej, ale aj jednoduchý výpočet hodnoty.

Najjednoduchšie numerické výpočty sa zapisujú pomocou aritmetických výrazov.

S počítaním aritmetických výrazov sa stretávame už od základnej školy. Takže iba stručné zhrnutie. Aritmetické výrazy sa vyhodnocujú podľa priority operátorov. Najvyššiu prioritu majú výrazy v zátvorkách, potom násobenia a delenia a nakoniec sčítania a odčítania. V prípade, ak je za sebou viac operátorov s rovnakou prioritou, vyhodnocujú sa z ľava do prava.

Aritmetické operácie s reálnymi číslami

Spravme si príklad, na ktorom je vidieť použitie.

```
double a = 6.0;
double b = 4.0;
double c;
c = a + b; // vypočítame 6.0 + 4.0 výsledok je 10.0 a ten sa vloží do premennej c
c = a - b; // vypočítame 6.0 - 4.0 výsledok je 2.0 a ten sa vloží do premennej c
          // predchádzajúca hodnota 10.0 sa nenávratne prepíše novou hodnotou 2.0
c = a * b; // vypočítame 6.0 * 4.0 výsledok je 24.0 a ten sa vloží do premennej c
c = a / b; // vypočítame 6.0 / 4.0 výsledok je 1.5 a ten sa vloží do premennej c
a = 3 * c + b / (2 + a) * 5; // na konci bude v premennej a hodnota 7
```

Posledný výraz sa vyhodnocuje v nasledujúcom poradí:

- 3 * c + b / (2 + a) * 5
- 3 * 1.5 + 4.0 / (2 + 6.0) * 5, teda dosadíme aktuálne hodnoty premenných
- 3 * 1.5 + 4.0 / 8.0 * 5
- 4.5 + 0.5 * 5
- 4.5 + 2.5
- 7.0

Takže nakoniec máme vypočítanú hodnotu výrazu 7.0, ktorá sa uloží do premennej a, čím sa nenávratne prepíše pôvodná hodnota 6.0 uložená v premennej a doteraz.

Z predchádzajúceho príkladu môžeme vidieť, že **vždy sa najskôr vyhodnotí pravá strana priradenia a až po jej vyhodnotení sa ukladá výsledok do premennej na napísanej ľavo od = (rovná sa)**.

Čo sa týka výrazov z matematického hľadiska, zrejme ste nenašli žiadne prekvapenia a všetko fungovalo podľa predstáv.

Aritmetické operácie s celými číslami

V prípade práce s celými číslami si už treba dať väčší pozor. Príkazy sčítania, odčítania a násobenia fungujú rovnako ako pri reálnych číslach.

Pri delení dvoch celých čísiel v bežnej matematike sa často stane, že výsledok už nie je celým číslom, napríklad ak delíme 3 a 2. **V Jave je výsledok delenia dvoch celých čísiel VŽDY celé číslo!** Operácia sa volá celočíselné delenie. S týmto delením ste sa určite stretli na základnej škole, keď ste sa učili deliť bez použitia kalkulačky.

Príklady celočíselného delenia:

```
5 / 2 je 2 zvyšok 1
16 / 4 je 4 zvyšok 0
-52 / 7 je -7 zvyšok 3
```

Napíšme si program, ktorý nám tieto výpočty overí. Pribudne nám nový operátor %, ktorého aplikáciou získame zvyšok po delení. Tento zvyšok je tiež vždy celé číslo.

```
int vysledok;
int zvysok;
vysledok = 5 / 2; // priradí sa hodnota 2
zvysok = 5 % 2; //priradí sa hodnota 1
vysledok = 16 / 4; // priradí sa hodnota 4
zvysok = 16 % 4; //priradí sa hodnota 0
```

Ako bude fungovať náš program s premennými a, b, c po zmene typov premenných?

```
int a = 6;
int b = 4;
int c;
c = a + b; // vypočítame 6 + 4 výsledok je 10 a ten sa vloží do premennej c
c = a - b; // vypočítame 6 - 4 výsledok je 2 a ten sa vloží do premennej c
c = a * b; // vypočítame 6 * 4 výsledok je 24 a ten sa vloží do premennej c
c = a / b; // vypočítame 6 / 4 výsledok je 1 (zvyšok 2). teda 1 sa vloží do premennej c
a = 3 * c + b / (2 + a) * 5; // do premennej a sa priradí hodnota 3
```

Posledný výraz sa vyhodnocuje v nasledujúcom poradí:

- 3 * c + b / (2 + a) * 5
- 3 * 1 + 4 / (2 + 6) * 5, teda dosadíme aktuálne hodnoty premenných
- 3 * 1 + 4 / 8 * 5
- 3 + 0 * 5 (lebo 4 / 8 je 0 zvyšok 4)
- 3 + 0
- 3

Aritmetické operácie s celými aj reálnymi číslami

Čo sa stane, ak začneme miešať celé a reálne čísla? V nasledujúcej tabuľke si namiesto znaku $ dosaďte ľubovoľný operátor +, -, * alebo /:

typy hodnôt vo výraze	typ hodnoty výsledku
int $ int	int
int $ double	double
double $ int	double
double $ double	double

Z tejto tabuľky vyplýva, že stačí aby aspoň jedna hodnota lebo premenná bola typu `double` a výsledok výrazu je typu `double`. Ukážme si to na príkladoch:

```
int vysledokTypuInt;
double vysledokTypuDouble;
vysledokTypuInt = 5 / 2; // priradí sa hodnota 2 (ide o celočíselné delenie)
vysledokTypuDouble = 5.0 / 2; // priradí sa hodnota 2.5
vysledokTypuDouble = 5 / 2.0; // priradí sa hodnota 2.5
vysledokTypuDouble = 5.0 / 2.0; // priradí sa hodnota 2.5
vysledokTypuDouble = 3 + 5.0 / 2; // priradí sa hodnota 3 + 2.5 teda 5.5
vysledokTypuDouble = 3.0 + 5 / 2; // priradí sa hodnota 3.0 + 2 (ide o celočíselné
delenie), teda 5.0
vysledokTypuDouble = 5 / 2; // priradí sa hodnota 2 (ide o celočíselné delenie, po
výpočte sa vykoná konverzia na reálne číslo 2.0)
```

Z predchádzajúcom príklade môžeme vidieť, že aj konkrétne hodnoty (nazývané tiež **literály**) majú svoj typ. Konkrétne číselné hodnoty nazývame numerické literály. Platí pravidlo, že ak numerický literál neobsahuje desatinnú čiarku (či presnejšie bodku), potom ide o celočíselnú hodnotu (podľa rozsahu, zvyčajne `int`). Naopak ak numerický literál obsahuje desatinnú čiarku, typ tejto hodnoty je `double`. Na to je dôležité myslieť najmä pri delení. Spomínate si na problém s nakreslením `n`-cípej hviezdy, kde `n` bolo 100? Problém bol spôsobený tým, že uhol natočenia sme vypočítali ako `360/n`. Teraz už vieme, že numerický literál je typu `int` (neobsahuje desatinnú bodku) a premenná (resp. parameter) `n` je tiež typu `int`. Keďže oba operandy sú celočíselné, znamená to, že operácia `/` bude operáciou celočíselného delenia. Ak teda `n` bolo 100, tak `360/100` nebude `3.6` ale `3`. My však chceme realizovať reálne delenie. Na to musí byť jeden z operandov reálne číslo. Výpočet uhla preto zapíšeme výrazom `360.0/n`, ktorý nám zabezpečí aplikovanie neceločíselného delenia.

Na záver tejto podkapitoly si ešte ukážeme, ako sa vyhnúť celočíselnému deleniu aj napriek tomu, že oba operátory sú celé čísla. Musíme urobiť takzvané **pretypovanie**, teda násilnú konverziu jedného z celočíselných operátorov operátorov na reálne číslo. Spravíme to tak, že pred niektorý z operátorov napíšeme `(double)`.

```
int a = 3;
int b = 2;
double vysledok = (double) a / b; // vypočíta sa 3.0 / 2 a priradí sa hodnota 1.5
double rovnakyVysledok = a / (double) b; // vypočíta sa 3 / 2.0 priradí sa hodnota 1.5
```

Aritmetické príkazy v skrátených tvaroch

Keďže programátori sú leniví, snažia sa niektoré príkazy skracovať, aby nemuseli veľa písať. Ide o príkazy, v ktorých sa na pravej strane priradenia použije stará hodnota premennej v nejakom jednoduchom aritmetickom výraze a výsledok sa zapíše opäť do tej istej premennej. Nasledujúca tabuľka vám ukáže prehľad takýchto skratiek - funguje pre celé aj reálne čísla s výnimkou operátora %, ktorý funguje iba pre celé čísla:

normálny zápis	**skrátený zápis**
a = a + b;	a+=b;
a = a - 10;	a-=10;
a = a * 5;	a*=5;
a = a / 7;	a/=7;

a = a % 3;	a%=3;
a = a + 1;	a++;
a = a - 1;	a--;

<< Premenné | Obsah | Podmienkový príkaz a logické výrazy >>

Podmienkový príkaz a logické výrazy

Podmienkové príkazy

Podmienkové príkazy (alebo aj príkazy vetvenia) nám umožňujú vykonať nejaký príkaz alebo postupnosť príkazov, iba ak je splnená nejaká podmienka.

Na vetvenie budeme používať príkaz **if**. Jeho syntax je nasledovná:

```
if ( podmienka ) {
   príkazy vykonané ak je podmienka splnená
}
```

Podmienka je buď logický literál (`true` alebo `false`), premenná typu `boolean`, metóda, ktorá vráti hodnotu typu `boolean` alebo logický výraz - proste všetko čoho výsledkom je **true** alebo **false**. Príkazy, ktoré sú v zátvorkách sa vykonajú, iba ak je podmienka pravdivá (**true**), inak sa preskočia.

V nasledujúcom príklade presunieme korytnačku na pozíciu [50,50] a otočíme ju o 90 stupňov iba ak je jej x-ová súradnica aspoň 150. Potom sa korytnačka bez ohľadu na to, kde sa nachádza, posunie o vzdialenosť 30 pixelov.

```
if (this.getX() > 150) {
   this.moveTo(50,50);
   this.turn(90);
}
this.step(30);
```

Rozšírený typ podmienkového príkazu **if-else** obsahuje aj blok pre príkazy, ktoré sa majú vykonať, v prípade, že podmienka nie je splená. Syntax je nasledovná:

```
if ( podmienka ) {
   príkazy vykonané ak je podmienka splená
}
else {
   príkazy vykonané ak podmienka nie je splená
}
```

Logické výrazy

Logické výrazy sú výrazy, ktorých výsledkom je buď **true** (pravda) alebo **false** (nepravda).

Základnými operáciami v logických operáciách sú príkazy porovnania:

operátor	význam
x == y	x sa rovná y
x != y	x sa nerovná y
x > y	x je väčšie ako y
x >= y	x je väčšie alebo rovné y
x < y	x je menšie ako y

x <= y	x je menšie alebo rovné y

`x` a `y` v tejto tabuľke môžu byť napríklad premenné, literály alebo metódy, ktoré vracajú nejaké hodnoty (napr. `Math.sqrt()`) Týmto spôsobom môžeme porovnávať iba primitívne hodnoty.

Ak `x` a `y` sú liretály, premenné typu `boolean`, alebo iné logické operácie, môžeme používať aj relačné operátory:

operácia	význam
x && y	x a súčasne y - nazývaný aj ako operátor AND
x || y	x alebo y - nazývaný aj ako operátor OR
!x	negácia x - nazývaný aj ako operátor NOT

podľa toho akú hodnotu majú operátory `x` a `y` sú výsledkom výrazu nasledovné hodnoty:

x	y	x && y	x || y	!x
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Pri operátoroch && a || je potrebné upozorniť na to, že sa vyhodnocujú z ľava do prava. Dôsledok toho je taký, že ak sa vo výraze (`x && y`) vyhodnotí `x` ako nepravdivé, pravdivosť `y` sa už neoveruje, lebo je jasné, že celý výraz bude nepravdivý. Podobne vo výraze (`x || y`) ak sa `x` vyhodnotí ako pravdivé, pravdivosť `y` sa už neoveruje, lebo celý výraz je už pravdivý. Zrejmešie to bude asi z príkladu:

```
double hodnota = -5.0;
boolean mamVelkuOdmocninu;

mamVelkuOdmocninu = (hodnota >= 0) && (Math.sqrt(hodnota) > 1000); // vráti false lebo hodnota je menšia ako 0, odmocnina sa nepočíta

mamVelkuOdmocninu = (Math.sqrt(hodnota) > 1000) && (hodnota >= 0); // vyhodí chybu lebo odmocnina zo záporného čísla sa počítať v reálnych číslach nedá
```

<< Aritmetické výrazy | Obsah | Cykly >>

Cykly

Cyklus (angl. loop) v programovaní slúži na to aby sme nejaké príkazy vykonávali opakovane. V Jave poznáme 3 tri základné cykly **for**, **while** a **do-while**. Líšia sa spôsobom riadenia toho, koľko krát sa má cyklus vykonať.

Cyklus for

Cyklus for sa používa vtedy, keď vieme dopredu počet opakovaní cyklu. Syntax je nasledovná:

```
for ( inicializácia ; podmienka cyklenia ; zmena riadiacej premennej ) {
  prikazy vykonávané pokiaľ je podmienka cyklenia splnená
}
```

Inicializácia sa vykonáva iba raz úplne na začiatku. Inicializácia obsahuje obvykle deklaráciu a inicializáciu riadiacej premennej napr. `int i = 0` čím sme si deklarovali premennú `i` typu int a priradili sme jej úvodnú hodnotu nula.

Testovanie podmienky cyklenia sa deje PRED každou iteráciou príkazov cyklu. Prikazy cyklu sa vykonajú, ak je podmienka cyklenia splená, v opačnom prípade cyklus ukončí a pokračuje s vykonávaním príkazov pod blokom tohto for cyklu. Podmienka cyklenia obvykle obsahuje nejaký jednoduchý logický výraz, ktorý porovnáva hodnotu riadiacej premennej s nejakou kritickou hodnotou napr. `i<5`.

Zmena riadiacej premennej sa vykonáva PO každom iterácii príkazov cyklu. Najčastejšie sa zvyšuje riadiaca premenná o 1 napr. `i++`.

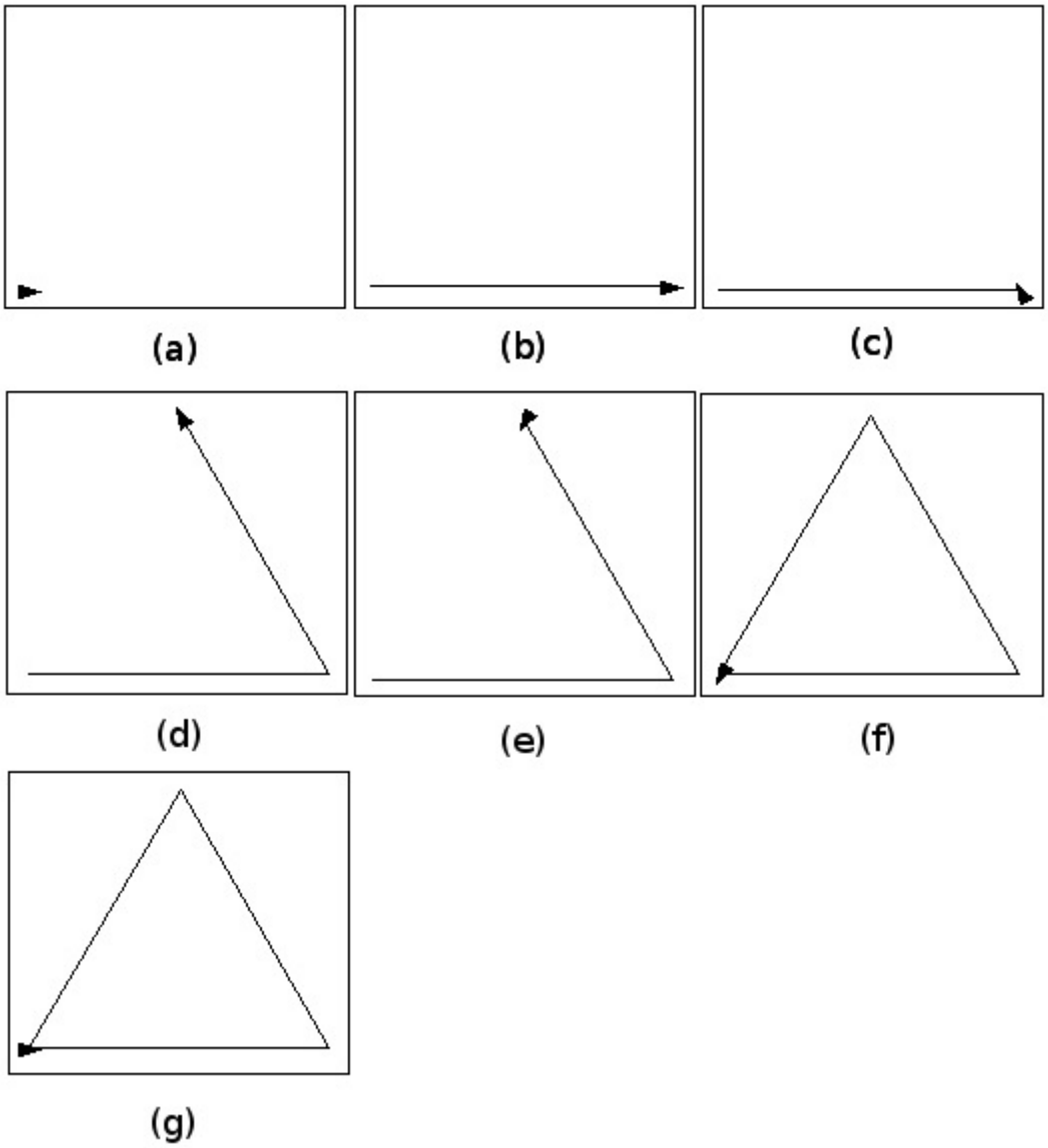
Ukážme si jednoduchý cyklus, v ktorom korytnačka nakreslí rovnostranný trojuholník - vykoná dokopy 3 takty cyklu, pričom v každom takte cyklu urobí krok dĺžky 100 a otočí sa o 120 stupňov.

```
1: public void trojuholnik() {
2:  for (int i = 0; i < 3; i++) {
3:    this.step(100);
4:    this.turn(-120);
5:  }
6: }
```

Poďme si rozanalyzovať, čo sa počas behu tejto metódy deje. Budeme sa pozerať na to, v ktorom riadku je práve výpočet a ako vyzerá kresliace plátno (obrázok pod tabuľkou)

riadok	akcia	hodnota v i	popis	stav plátna
1		i neexistuje	začína sa metóda	(a)
2	int i = 0	0	deklarácia a inicializácia riadiacej premennej i	(a)
2	i < 3	0	nula je menšia ako 3 - ideme spraviť takt cyklu	(a)
3	turtle.step(100)	0	korytnačka sa pohne	(b)
4	turtle.turn(-120)	0	korytnačka sa otočí	(c)
2	i++	1	zvýšime riadiacu premennú o 1	(c)

2	i < 3	1	1 je menšia ako 3 - ideme spraviť takt cyklu	(c)
3	turtle.step(100)	1	korytnačka sa pohne	(d)
4	turtle.turn(-120)	1	korytnačka sa otočí	(e)
2	i++	2	zvýšime riadiacu premennú o 1	(e)
2	i < 3	2	2 je menšie ako 3 - ideme spraviť takt cyklu	(e)
3	turtle.step(100)	2	korytnačka sa pohne	(f)
4	turtle.turn(-120)	2	korytnačka sa otočí	(g)
2	i++	3	zvýšime riadiacu premennú o 1	(g)
2	i < 3	3	3 nie je menšie ako 3 - vyskočíme z cyklu, premenná i zaniká	(g)
6		i neexistuje	končíme metódu	(g)



Povedali sme si, že cyklus for sa používa, keď dopredu vieme, koľko krát budeme tento cyklus opakovať. V predchádzajúcom príklade sa cyklus opakoval 3 krát. Analogicky môžeme cyklus vykonať napríklad 27 krát :

```
for (int i = 0; i < 27; i++) {
  ...
}
```

Cyklus while

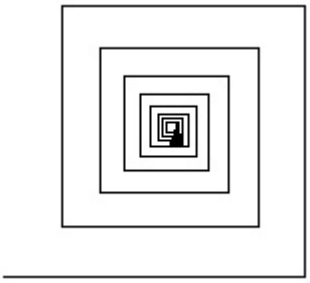
Tento cyklus použijeme vtedy, ak vieme napísať nejakú podmienku, za splnenia ktorej chceme aby sa cyklus opakoval. Syntax je nasledovná:

```
while (podmienka cyklenia) {
  prikazy vykonávané pokiaľ je podmienka cyklenia splnená
}
```

Uveďme si ako príklad novú metódu pre korytnačku, ktorá umožní korytnačke kresliť štvorcovú špirálu.

```
public void stvorcovaSpirala() {
  double dlzkaKroku = 150;
  while (dlzkaKroku > 5) {
    this.step(dlzkaKroku);
    this.turn(-90);
    dlzkaKroku = dlzkaKroku * 0.90;
  }
}
```

Tento cyklus teda vykonáva 3 príkazy - posun, otočenie a zmenu dĺžky kroku na 90% predchajúcej dĺžky. Cykus sa skončí, keď dĺžka kroku bude menšia alebo rovná 5. Výsledok tejto metódy je nasledovný:



Možete vidieť, že pri tejto úlohe sa naozaj ťažko odhaduje, koľko bude potrebných taktov cyklu, preto for cyklus nevieme použiť. Počet taktov cyklu je pre nás irelevantný. My sme chceli len to, aby sa korytnačka už netočila, keď už by mala začať kresliť čiaročky menšie ako 5 pixelov.

Príkazy break a continue

Príkazy **break** a **continue** sa dajú použiť v ľubovoľnom type cyklu.

Príkaz **break** slúži na okamžité ukončenie cyklu bez ohľadu na to, či je podmienka cyklenia splnená, alebo nie.

Príkaz **continue** zabezpečí ukončenie daného taktu cyklu teda preskočenie všetkých zvyšných príkazov v cykle a opätovné overovanie podmienky cyklenia a v prípade jej správnosti normálne pokračovanie v ďalších taktoch cyklu. Ak ide o cyklus for, tak ešte pred overením podmienky sa iteruje riadiaca premenná cyklu.

```
for (int i = 0; i < 20 ; i++) {
  System.out.println( i );
  if (i % 2 == 0) {  // ak je zvyšok po delení dvojkou rovný 0, tj. ak i je párne číslo
    continue;
  }
  if (i > 5) {
    break;
  }
  System.out.println("ahoj");
```

```
}
System.out.println("koniec");
```

Výpis z tohoto programu je nasledovný:

```
0
1
ahoj
2
3
ahoj
4
5
ahoj
6
7
koniec
```

Vidíte, že po každom párnom čísle sa všetko ostatné za **continue** preskočí, zvýši sa `i` o 1 a začína ďalší cyklus. V prípade, že `i` je nepárne, príkaz **continue** sa nevykoná, ale overí sa či `i` nie je väčšie ako 5.

Ak `i` nie je väčšie ako 5 vykoná sa ešte posledný príkaz v cykle, ktorý vypíše ahoj a cyklus normálne pokračuje v svojom ďalšom takte.

Ak `i` je väčšie ako 5, tak sa vykoná príkaz **break**, celý cyklus skončí a pokračuje sa vo vykonávaní príkazov za cyklom. V tomto prípade sa už vykoná iba príkaz, ktorý vypíše koniec.

Cyklus do-while

Tento cyklus sa od cyklu while líši tým, že podmienku cyklenia testuje PO vykonaní jedného taktu cyklu. Z toho vyplýva, že prvý takt sa spraví vždy a ďalšie iba ak je podmienka cyklenia splnená. Syntax je nasledovná:

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

<< Podmienkový príkaz a logické výrazy | Obsah | Metódy vracajúce hodnoty - funkcie >>

Metódy vracajúce hodnoty - funkcie

Pri skúmaní metód korytnačiek sme mohli pozorovať, že niektoré metódy nielen niečo vykonajú, ale naviac aj vrátia (vypočítajú) nejakú hodnotu. Typickým príkladom takýchto metód sú `getX()`, ktorá vráti x-ovú súradnicu aktuálnej pozície korytnačky, či `distanceTo(double x, double y)`, ktorá vráti vzdialenosť korytnačky k bodu na súradniciach `(x, y)`. Takéto metódy vracajúce nejakú hodnotu sa zvyknú označovať aj ako **funkcie**. V predchádzajúcich častiach sme sa už naučili vytvárať vlastné metódy pre objekty tried rozširujúcich triedu `Turtle`. Tieto metódy však len vykonali nejakú postupnosť príkazov - žiadnu hodnotu nevypočítavali a nevracali.

Vytvorenie metód vracajúcich hodnotu

Spôsob, ako si vytvoriť vlastnú metódu, ktorá vracia nejakú hodnotu, je v Jave veľmi jednoduchý a skladá sa z 2 fáz:

- definovanie, že metóda vracia hodnotu nejakého typu
- použitie príkazu, ktorý vráti nejakú hodnotu ako výsledok vykonávania metódy

Na definovanie toho, že metóda vracia hodnotu nejakého typu, stačí zameniť slovíčko **void** za typ hodnoty, ktorú metóda vracia. Ak teda chceme mať metódu, ktorá vráti hodnotu typu **double**, jej hlavička bude vyzerať nasledovne:

```
public double stvorcovaSpirala()
```

Analogicky, hlavička metódy, ktorá by mala vrátiť počet deliteľov čísla `n` by mohla vyzerať takto:

```
public int pocetDelitelov(int n)
```

Ak to zosumarizujeme, tak slovíčko **void** za slovom **public** hovorí, že metóda nevracia žiadnu hodnotu. Naopak použite označenia nejakého typu (`double`, `int`, `float`, `boolean`, ...) za slovom **public** hovorí, že metóda vracia hodnotu daného typu.

Ostáva nám už len povedať, ako v Jave povieme, že nejaká hodnota je tou hodnotou, ktorú má vrátiť metóda. Na to, aby sme vrátili nejakú hodnotu sa používa príkaz **return**:

```
return hodnotaKtoruVraciame;
```

Prirodzene, ak nejaká hodnota má byť metódou vrátená, tak musí byť takého typu, ako sme povedali (definovali) v hlavičke metódy. O príkaze **return** je dôležité zapamätať si, že ukončuje vykonávanie metódy, t.j. žiadne ďalšie príkazy metódy sa po ňom už nevykonávajú.

Pozrime sa na niekoľko príkladov:

- metóda, ktorá nakreslí štvorcovú špirálu a vráti, koľko toho korytnačka prešla:

```
public double stvorcovaSpirala() {
        double prejdene = 0;
        double dlzkaKroku = 150;
```

```
        while (dlzkaKroku > 1) {
                this.step(dlzkaKroku);
                prejdene = prejdene + dlzkaKroku;

                dlzkaKroku = dlzkaKroku * 0.9;
                this.turn(90);
                JPAZUtilities.delay(30);
        }

        return prejdene;
}
```

- metóda, ktorá vráti menšiu z hodnôt parametrov (obe metódy vrátia to isté, ale pozor, obe sa nemôžu naraz nachádzať v tej istej triede lebo sa volajú rovnako a majú rovnaký počet a rovnaké typy parametrov):

```
public double min(double c1, double c2) {
        if (c1 < c2) {
                return c1;
        } else {
                return c2;
        }
}

public double min(double c1, double c2) {
        if (c1 < c2) {
                return c1;
        }

        return c2;
}
```

Čo je dôležité si pamätať:

- Vykonanie príkazu `return` ukončuje ďalšie vykonávanie príkazov metódy.
- Každá metóda, ktorá nevracia `void`, musí každé svoje vykonanie skončiť príkazom `return`, t.j. vrátením hodnoty
- Vrátená hodnota musí byť kompatibilná s definovaným typom návratovej hodnoty.
    - napr. metóda definovaná ako metóda vracajúca `int` nemôže vrátiť reálne číslo (`double` hodnotu)
- Príkaz `return;` možno použiť v metódach vracajúcich void na okamžité ukončenie vykonávania metódy.

Rôzne matematické metódy

Metódy vracajúce (počitajúce) nejaké hodnoty môžeme celkom dobre využiť na výpočet rôznych matematicky motivovaných úloh pracujúcich s číslami. Pri riešení takýchto úloh si poväčšine vystačíme so šikovnou kombináciou pár fínt:

- poslednú cifru čísla uloženého v premennej `c` dostaneme výrazom `c % 10` (pripomíname, že `%` označuje zvyšok po delení)
    - 135 % 10 = 5
    - 12 % 10 = 2
- ak chceme vytvoriť číslo, ktoré vznikne odstránením poslednej cifry čísla uloženého v premennej `c`, použijeme výraz `c / 10` (pripomíname, že ak oba operandy sú celé čísla, tak `/` reprezentuje celočíselné delenie)
    - 135 / 10 = 13
    - 12 / 10 = 1

- ak chceme vytvoriť číslo, ktoré vznikne pridaním cifry `cifra` za cifry čísla `c`, použijeme výraz `c * 10 + cifra`
  - cifra 8 a číslo 135: 135 * 10 + 8 = 1350 + 8 = 1358
- ak chceme zistiť, či číslo `a` je deliteľom čísla `b`, tak otestujeme, či `b % a == 0`
- počítače ľúbia (a vedia) počítať a preto sa niekedy na riešenie úloh používa metóda otestovania všetkých možností
  - na zistenie počtu deliteľov čísla `n` jednoducho otestujeme každé celé číslo medzi `1` a `n`, či náhodou nie je deliteľom čísla `n`

Typickou úlohou, ktorá využíva vyššie uvedené finty, je výpočet ciferného súčtu. Ciferný súčet čisla `154` je `10`, lebo `1+5+4=10`. Výpočet môže postupovať takto: postupne z čísla odtrhávame poslednú cifru (lebo tú dokážeme vybrať) a odtrhnutú cifru pripočítavame k aktuálne zapamätanému súčtu cifier. Tento postup opakujeme, kým sú nejaké cifry, t.j. kým je číslo rôzne od `0`.

V Jave by sme to naprogramovali takto:

```
public int cifernySucet(int cislo) {
        // premenna, do ktorej budeme pocitat sucet
        int vysledok = 0;

        // kym sme z cisla "nevyhodili" vsetky cifry ...
        while (cislo > 0) {
                // vyberieme poslednu cifru
                int cifra = cislo % 10;
                // pridame cifru do suctu
                vysledok = vysledok + cifra;
                // skratime cislo o poslednu cifru
                cislo = cislo / 10;
        }

        // vratime vypocitany sucet
        return vysledok;
}
```

Trieda Math

Aritmetické výrazy naberajú na svojej sile, ak už vieme používať aj zložitejšie algebraické operácie ako sú napríklad sínus, odmocnina, mocnina, zaokrúhlenie, absolútna hodnota, logaritmus a podobne. Všetky tieto operácie sú už samozrejme vytvorené. Zoznam všetkých takýchto operácií si môžete pozrieť v dokumentácii triedy Math. Okrem metód tu môžeme nájsť aj dve konštanty: Ludolfovo číslo `Math.PI` a Eulerovo číslo `Math.E`. Použitie je jednoduché. Povedzme, že chceme vypočítať vzdialenosť dvoch bodov v dvojrozmernom priestore podľa Pytagorovej vety.

Použijeme metódu na odmocninu `Math.sqrt` a metódu na mocninu `Math.pow`.

```
double x1 = 5.0;
double y1 = 1.0;
double x2 = 2.0;
double y2 = -3.0;
double d = Math.sqrt( Math.pow( x2 - x1, 2 ) + Math.pow( y2 - y1, 2 ) );
System.out.print("Vzdialenosť je : ");
System.out.println(d); // vypíše 5.0
```

Komentáre

Isto pri pozretí do rozsiahlejšieho programu rýchlo stratíte prehľad, kde sa čo robí. Dokonca aj keď je to náš program, po dlhšom čase v ňom stratíme prehľad. Pri programovaní píšeme príkazy tak, aby im rozumela najmä Java a podľa nich vykonávala presne to, čo chceme. Avšak určite sa mnohokrát stane, že vytvorené programy bude čítať človek: niekto iný (napr. cvičiaci pri kontrolovaní domácich zadaní) alebo aj my po nejakom dlhšom čase. Aby sme sa v programoch zorientovali, je dobré do nich písať **komentáre**. Komentár je časť textu, ktorá slúži len ako informácia pre ľudských čitateľov (Java ich ignoruje). Pre komentáre platia tieto pravidlá:

- všetky znaky, ktoré nasledujú za dvojicou znakov `//` až do konca riadka, sú chápané ako komentár. Takýto druh komentárov sme už "tajne" použili v niektorých skorších príkladoch.
- všetky znaky medzi dvojicou znakov `/*` a `*/` sú chápané ako komentár. Týmto spôsobom vieme vytvárať aj niekoľko riadkové komentáre.

Debugovanie

Človek je tvor omylný. Neraz sa stane, že programátor spraví pri programovaní chybu a program nerobí, to čo má. Skúsený programátor vtedy nepanikári, ale začne hľadať chybu tak, že vykonáva program po jednotlivých krokoch a skúma, ako sa menia premenné a čo všetko sa popri tom deje. Tomuto sa hovorí **debugovanie**, alebo tiež správne slovensky: krokovanie, ladenie, či trasovanie. Debugovanie v Jave zachycuje nasledujúci video-tutoriál

- Krátka ukážka základov debugovania

Referencie a konštruktory

Pri našom programovaní doposiaľ sme sa stretli s 2 typmi premenných. Prvým typom boli premenné primitívneho typu (`int`, `double`, `boolean`, ...), ktoré uchovávali jednoduchú hodnotu. Druhým typom premenných boli premenné, prostredníctvom ktorých sme komunikovali s vytvorenými objektami vo "svete objektov" (napr. s kresliacou plochou, korytnačkou, či ObjectInspectorom). O týchto premenných sme si povedali, že referencujú objekty. V Jave už viac typov premenných nejestvuje.

V Jave existujú 2 typy premenných (a tiež parametrov a návratových hodnôt):

- premenné primitívneho typu
  - `int`, `byte`, `short`, `long`, `double`, `float`, `boolean` a `char`
  - ich úlohou je uchovávať jednoduchú hodnotu
- premenné referenčného typu
  - ich úlohou je uchovávať referenciu na objekt (vo "svete objektov")

Referenciu na objekt si môžeme predstaviť ako akúsi "šípku na objekt", či "adresu" miesta, kde objekt "býva". Ďalšou metaforou pre referenciu je metafora rodného čísla. Pre premenné referenčného typu platia všetky pravidlá, ako pre premenné primitívneho typu (práca s neinicializovanou premennou, priradenie, ...).

Premenná referenčného typu nemôže referencovať hocijaké objekty, môže referencovať iba objekty určitej triedy. Premennú `refPremenna`, ktorá môže referencovať objekty nejakej triedy `Trieda` deklarujeme tak, že najprv napíšeme názov triedy (v našom prípade `Trieda`), potom medzeru a nakoniec pridáme názov premennej. To všetko ukončíme bodkočiarkou:

```
Trieda refPremenna;
```

Takto vytvorená premenná je neinicializovaná. Do premennej referenčného typu môžeme uložiť:

- referenciu na objekt (vhodnej triedy)
- špeciálnu hodnotu **null**, ktorá hovorí, že daná premenná nereferencuje žiaden objekt

Ako si predstaviť abstraktný pojem referencia? Každý človek na Slovensku je jednoznačne identifikovaný svojim rodným číslom. Rodné číslo je jedinečným identifikátorom každého človeka, ktorý sa narodil na Slovensku. Podobne si referenciu na objekt môžeme predstaviť ako rodné číslo objektu, ktoré dostáva objekt, keď sa vytvorí vo svete objektov (pomocou `new`). Teda, ak máme premennú `k` deklarovanú ako `Turtle k`, tak premenná k je schopná uchovávať jedno rodné číslo nejakého korytnačieho objektu alebo hodnotu `null`. Ak píšeme `k.step(100)`, tak hovoríme, aby objekt, ktorého rodné číslo je uložené v premennej `k`, vykonal metódu `step` s parametrom 100.

Veľmi často sa môžeme v programoch stretnúť s takouto podmienkou, ktorá testuje, či premenná naozaj referencuje nejaký objekt:

```
if (refPremenna != null) {
        // volanie metod referencovaneho objektu
}
```

Takýto test je dôležitý, ak nemáme informácie o hodnote v premennej. Ak je v nej totiž uložená hodnota `null`, tak premenná nereferencuje žiaden objekt. To má za následok to, že ak pomocou takejto premennej

(obsahujúcej `null`) skúsime zavolať nejakú metódu objektu (komunikovať s objektom), program skončí s chybovou hláškou.

Ďalším typickým testom je test, či 2 premenné referenčného typu, ktoré referencujú objekty rovnakej triedy, referencujú ten istý objekt:

```
if (refPremenna1 == refPremenna2) {
        // ...
}
```

Pozrime sa na pár komentovaných ukážok priradení referenčných hodnôt:

```
Turtle k1;
Turtle k2;
WinPane p;

...

// Po priradení bude premenná k1 referencovať ten istý objekt, ako premenná k2
k1 = k2;
// Premenná k1 nebude referencovať žiaden objekt
k1 = null;
// Takéto priradenie je nekorektné, keďže k1 referencuje objekty triedy Turtle, kým p
objekty triedy WinPane
p = k1;
```

Konštruktory - použitie

Referencie na objekty môžeme nájsť v iných premenných referenčného typu, či referenciu na objekt nám môžu vrátiť metódy. Referenciu na objekt ale vracia aj príkaz `new`, ktorý už dlho používame. V skutočnosti príkaz `new` spraví to, že najprv vo "svete objektov" vytvorí a inicializuje objekt danej triedy, pričom výsledkom tejto operácie je referencia na vytvorený objekt (a túto hodnotu môžeme niekam uložiť, čo zvyčajne aj robíme).

```
WinPane plocha = new WinPane();
```

Guľaté zátvorky sú podobné zátvorkám pri volaní metód. V skutočnosti sa pri vytvorení a inicializácii objektu vykonáva tzv. **konštruktor** objektu, čo je špeciálna "metóda" (postupnosť príkazov), ktorá skonštruuje objekt. Konštruktory, tak ako všetky metódy, môžu mať aj parametre. Napríklad v prípade triedy `WinPane`, okrem konštruktora bez parametrov, existuje aj konštruktor s parametrami, ktorý vytvorí kresliacu plochu požadovaných rozmerov:

```
WinPane plocha = new WinPane(400, 200);
```

O tom, aké konštruktory majú tie-ktoré triedy, sa dozvieme v inej prednáške.

Pravidlá pre referencie a objekty

Pri práci s objektami a premennými referenčného typu by sme si mali zapamätať niekoľko pravidiel:

- Nie je pravdou, že neinicializovaná premenná obsahuje hodnotu `null`. Hodnotu do neinicializovanej premennej treba vždy najprv priradiť.
- Objekty sa vytvárajú príkazom `new` v spolupráci s konštruktorom (a ten môže mať parametre). Výsledkom príkazu je referencia na vytvorený objekt.

- Objekty vieme len vytvoriť, nevieme ich "zničiť"
- Jeden objekt môže referencovať mnoho premenných referenčného typu.
- Ak objekt nie je referencovaný žiadnou premennou referenčného typu, t.j. nikde nie je uchovaná referencia naň, objekt je "stratený", lebo s ním nevieme komunikovať.

<< Debugovanie programov | Obsah | Typ char a znakové reťazce >>

Typ char a znakové reťazce

Primitívny typ `char` (znak)

Posledný primitívny (jednoduchý) typ, ktorý sme nespomínali, je typ `char`. Typ `char` predstavuje jeden znak. Znakové literály (konkrétne znakové hodnoty) píšeme tak, že znak uzavrieme do apostrofov:

```
char znak = 'a';
```

Java, narozdiel od iných programovacích jazykov/platforiem (C, C++, Pascal, Delphi, PHP, ...), uchováva znaky v originálnej verzii kódovania UNICODE, t.j. v premennej typu `char` môže byť jeden z 65536 znakov tohto kódovania. Vďaka tomu v Jave nemáme problém s národnými znakmi (slovenčina, ruština, švédčina, ázijské znaky, ...).

Niektoré znaky majú svoj historický význam, iné znaky naopak musíme zapisovať špeciálnym spôsobom. Java má niekoľko špeciálnych znakov, z ktorých najdôležitejšie sú:

- znak literál medzery: `' '`
- znak literál tabelátora: `'\t'`
- znak literál konca riadka: `'\n'`
- znak literál lomítka: `'\\'`
- znak literál úvodzovky: `'\"'`
- znak literál apostrofu: `'\''`

(Pre pokročilých.) Za kódovaniami sa skrýva dlhá historia. Pod kódovaním si môžeme predstaviť veľkú tabuľku, ktorá hovorí, aké písmeno reprezentuje `i`-ty znak kódovania:

- ASCII
  - 128 znakov (číselné kódy 0.127), anglická abeceda + špeciálne znaky
- rôzne kódovania s tabuľkou majúcou 256 znakov (prvých 128 je totožných s ASCII):
  - Windows-1250
  - ISO-8859-1
  - ISO-8859-2
- UNICODE
  - originálne UNICODE malo 65536 znakov, posledné rozšírenia môžu mať ešte viac znakov

Znaky môžeme tak, ako všetky jednoduché hodnoty navzájom porovnávať. V prípade znakov je toto porovnanie porovnaním toho, či sa daný znak nachádza v UNICODE tabuľke skôr. Keďže ASCII je na začiatku každého kódovania, veľmi užitočná vlastnosť je to, že cifry, malé písmená a aj veľké písmená anglickej abecedy idú pekne za sebou.

Častý je test na overenie toho, či v znakovej premennej je znak cifry:

```
if ((z >= '0') && (z <= '9')) {
        //...
}
```

Podobne môžeme testovať znaky anglickej abecedy ...

Reťazce

Premenné typu `char` nám dovoľujú uchovať jeden znak. S týmto si ale pri praktických problémoch veľmi nepomôžeme. Zvyčajne totiž potrebujeme uchovať postupnosť znakov. Na uchovanie postupnosti znakov slúžia objekty triedy `String`. Objekt triedy `String` uchovávajúci nejakú postupnosť znakov vytvoríme takto:

```
String s = new String("Ahoj Java");
```

Všimnime si, že ako parameter konštruktora uvádzame reťazcový literál (konkrétnu postupnosť znakov). Reťazcové literály vždy píšeme medzi úvodzovky. Ak v reťazcovom literály chceme použiť niektorý zo špeciálnych znakov, zapíšeme ho v takej forme, ako to je uvedené vyššie (apostrofy samozrejme vynecháme).

Keďže reťazce sú uložené v objektoch, platia pre ne všetky "objektové" pravidlá. Dôležité je si uvedomiť, že vo vyššie uvedenom príklade premenná `s` neobsahuje samotný reťazec, ale iba referencuje objekt triedy `String`, ktorý túto postupnosť znakov uchováva.

Tak ako všetky objekty, aj objekty triedy `String` majú zaujímavé metódy:

- `int length()` - vráti dĺžku reťazca (počet znakov)
- `char charAt(int index)` - vráti znak na zadanom indexe. Indexom nazývame pozíciu v reťazci. Jednotlivé znaky sú indexované od 0, t.j. prvý znak reťazca je na indexe 0, posledný znak na indexe `length()-1`.
- `String concat(String s)` - vráti referenciu na novovytvorený objekt triedy `String`, ktorého obsah vznikol spojením (zreťazením) postupnosti znakov tohto objektu a znakov objektu triedy `String` referencovaného z parametra `s`.

Pri rôznych metódach pracujúcich so znakmi potrebujeme postupne spracovať jednotlivé znaky reťazca. Ukážme si to na príklade metódy, ktorá spočíta počet výskytov znaku 'a' v zadanom reťazci:

```
public int pocetVyskytov(String s) {
        int vysledok = 0;
        for (int i=0; i<s.length(); i++) {
                if (s.charAt(i) == 'a') {
                        vysledok++;
                }
        }
        return vysledok;
}
```

Opäť pripomíname, že parameter `s` neobsahuje samotný reťazec, ale len referencuje objekt triedy `String`, ktorý máme "preskúmať".

Pri riešení úloh potrebujeme často zistiť, či postupnosti znakov v 2 (rôznych) objektoch triedy `String` sú rôzne, alebo rovnake. Na tento účel vieme použiť metódu `equals`.

```
String s = new String("Ahoj");
String r = new String("Ahoj");
if (s.equals(r)) {
       // príkazy, ktoré sa vykonajú ak je postupost písmen rovnaká
}
```

Veľmi častou chybou pri práci s reťazcami býva, že rovnosť reťazcov sa testuje pomocou operácie rovnosti:

```
if (s == r) {

}
```

My však už vieme, že tento postup je zlý práve preto, že premenné `s` a `r` sú premenné referenčného typu. Vyššie uvedený test len testuje, či premenné `r` a `s` referencujú ten istý objekt a nie to, či obsah (postupnosť znakov) je rovnaký.

Trieda `String` patrí medzi triedy, ktoré majú v Jave špeciálnu podporu, t.j. sú dovolené isté skrátené konštrukcie:

- namiesto:

```
String s = new String("Ahoj Java");
```

stačí napísať

```
String s = "Ahoj Java";
```

- namiesto:

```
String s = r.concat(t);
```

stačí napísať

```
String s = r + t;
```

Operátor `+` funguje ako operátor zreťazenia, ak jeden z jeho operandov je reťazec (referencia na objekt triedy `String`). Dokonca, ak druhý z operandov nie je reťazec, Java sa túto hodnotu pokúsi prerobiť na reťazec, ktorý bude použitý pri "zreťazovaní".

Môžeme tak písať aj takéto príkazy:

```
int c = 10;
String s = "V premennej c je cislo: " + c;
```

Pozrime sa, ako by vyzerala metóda, ktorá vráti referenciu na novovytvorený reťazec (objekt triedy `String`), ktorý obsahuje znaky zadaného reťazca v opačnom poradí. Pri riešení postupne vyberáme znaky reťazce referencovaného parametrom `s` zľava doprava. V každom kroku vytvárame nový reťazec, ktorý vznikne zreťazením aktuálneho znaku a doposiaľ vytvorenej obrátenej postupnosti.

```
public String obratRetazec(String s) {
        if (s == null)
                return null;

        String vysledok = "";
        for (int i=0; i<s.length(); i++) {
                vysledok = s.charAt(i) + vysledok;
        }

        return vysledok;
}
```

Zoznam metód objektov triedy `String` možno nájsť na stránke:
http://java.sun.com/javase/6/docs/api/java/lang/String.html#method_summary

StringBuilder

Predchádzajúce metódy "vytvárania" nových reťazcov majú jednu podstatnú nevýhodu: pri ich vykonávaní vznikne veľa "odpadu" vo forme dočasných reťazcov (objektov triedy `String`). Toto je prirodzené, keďže objekty triedy `String` majú tú vlastnosť, že ich obsah (uchovávanú postupnosť znakov) po vytvorení nejde už zmeniť. Aby sme sa vyhli zbytočnému vytváraniu dočasných reťazcov, môžeme pri budovaní obsahu objektov triedy `String` použiť objekty triedy `StringBuilder`.

Poslaním objektov triedy `StringBuilder` je podobne ako v prípade objektov triedy `String` uchovávať postupnosť znakov. Narozdiel od nich však objekty triedy `StringBuilder` dovoľujú meniť svoj obsah. Najčastejšie používané metódy objektov triedy `StringBuilder` sú:

- `append` - pridá nejakú postupnosť znakov na koniec uchovávaného reťazca (pozor, metóda `append` mení obsah objektu, narozdiel od metódy `concat` objektov triedy `String`, ktorá vytvorí nový objekt triedy `String`)
- `toString` - vráti referenciu na novovytvorený objekt triedy `String`, ktorého obsah bude rovnaký ako je aktuálny obsah objektu triedy `StringBuilder`.

Príklad:

```
public String obratRetazec(String s) {
        if (s == null)
                return null;

        StringBuilder vysledok = new StringBuilder();
        for (int i=s.length()-1; i>=0; i--) {
                vysledok.append(s.charAt(i));
        }

        return vysledok.toString();
}
```

<< Referencie a konštruktory | Obsah | Myšacie udalosti v JPAZe >>

Myšacie udalosti v JPAZe

V súčasnosti sme už zvyknutí, že každú rozumnú aplikáciu vieme nejako ovládať aj prostredníctvom myši. Kresliaca plocha JPAZu (objekty triedy `WinPane`) nie je v tomto smere výnimkou. Spracovávať myšacie udalosti (elementárne akcie vykonané myšou) kresliacej plochy vieme iba v triedach, ktoré rozširujú triedu `WinPane`. Ak v takejto triede vytvoríme nasledovnú metódu, JPAZ sa postará, aby sa vykonala vždy, keď sa klikne myšou do kresliacej plochy:

```
protected void onMouseClicked(int x, int y, MouseEvent detail) {
        // prikazy metody
}
```

Metóda `onMouseClicked` má aj 3 parametre. Prostredníctvom nich nám JPAZ povie bližšie informácie o tom, čo sa stalo. Konkrétne v parametri `x` dostaneme x-ovú súradnicu myši, v parametri `y` y-ovú súradnicu myši a nakoniec v parametri `detail` referenciu na objekt triedy `MouseEvent`. Objekt triedy `MouseEvent` je zaujímavý tým, že nám vie poskytnúť ďalšie informácie o tom, čo sa stalo.

Spomeňme pár najzaujímavejších metód objektov triedy `MouseEvent`:

- `boolean isAltDown()` - true, ak pri udalosti bol kláves `Alt` zatlačený
- `boolean isControlDown()` - true, ak pri udalosti bol kláves `Ctrl` zatlačený
- `boolean isShiftDown()` - true, ak pri udalosti bol kláves `Shift` zatlačený
- `int getButton()` - vráti kód tlačidla, ktorý udalosť spôsobil. Hodnoty týchto pre jednotlivé tlačidla kódov sú tieto:
    - `MouseEvent.BUTTON1`
    - `MouseEvent.BUTTON2`
    - `MouseEvent.BUTTON3`

Pozrime sa, ako by vyzerala metóda `onMouseClicked`, ak by sme chceli niečo vykonať, iba vtedy ak sa kliklo ľavým tlačidlom myši a zároveň bol počas kliknutia zatlačený kláves `Ctrl`:

```
protected void onMouseClicked(int x, int y, MouseEvent detail) {
        if ((detail.getButton() == MouseEvent.BUTTON1) && detail.isControlDown()) {

                // prikazy metody
        }
}
```

Okrem kliknutia do kresliacej plochy existujú aj ďalšie myšacie udalosti, ktoré vie JPAZ "obslúžiť" pomocou metód so "správnym" menom a parametrami:

- `onMouseClicked` - pri kliknutí do plochy (vykonáva sa až vtedy, keď bolo tlačidlo myši pustené)
- `onMousePressed` - pri zatlačení tlačidla myši
- `onMouseReleased` - pri uvoľnení tlačidla myši
- `onMouseMoved` - pri pohybe myši v stave, kedy žiadne jej tlačidlo nie je zatlačené
- `onMouseDragged` - pri pohybe myši v stave, kedy niektoré jej tlačidlo je zatlačené

Všetkých 5 spomenutých metód má rovnaké predpísané parametre ako `onMouseClicked`:

```
protected void onMouseClicked(int x, int y, MouseEvent detail)
protected void onMousePressed(int x, int y, MouseEvent detail)
```

```
protected void onMouseReleased(int x, int y, MouseEvent detail)
protected void onMouseMoved(int x, int y, MouseEvent detail)
protected void onMouseDragged(int x, int y, MouseEvent detail)
```

Kreslenie do kresliacej plochy

Zvyčajne, keď vytvoríme novú triedu, ktorá rozširuje triedu `WinPane`, potrebujeme do nej pridať pár metód, ktoré niečo nakreslia. Žial, samotná kresliaca plocha (aj keď je kresliaca) veľmi kresliť nevie. Ak teda chceme niečo do kresliacej plochy nakresliť, musíme si najprv vytvoriť korytnačku, ktorou to, čo treba, nakreslíme. Aby mohla kresliť, musíme ju najprv do plochy pridať. No a po skončení kreslenia, musíme korytnačku z kresliacej plochy "vyhodiť":

```
public void kresliacaMetodaPlochy {
        // Vytvorime korytnacku, ktoru pouzijeme na kreslenie
        Turtle k = new Turtle();
        // Pridame ju do tejto kresliacej plochy
        this.add(k);

        // Korytnacka k kresli ...

        // Korytnacku uz nepotrebujeme, tak ju "vyhodime"
        // z tejto kresliacej plochy
        this.remove(k);
}
```

<< Typ char a znakové reťazce | Obsah | Inštančné premenné >>

Inštančné premenné

Všetky premenné, s ktorými sme mali doposiaľ možnosť pracovať, boli tzv. lokálne - t.j. "žili" iba počas vykonávania metód. Neraz však potrebujeme, aby si objekt pamätal nejakú informáciu počas celého "svojho života". Na tento účel používame tzv. **inštančné** (objektové) premenné. Tieto premenné vznikajú spolu s objektom pri jeho vytvorení. Ak chceme definovať inštančnú premennú pre objekty nejakej triedy, stačí deklaráciu premennej napísať mimo tiel metód a pred deklaráciu pridať slovíčko `private`:

```
public class MojaPlocha extends WinPane {
        private int pocetBodiek = 0;

        // metody triedy ...
}
```

Hoci inštančnú premennú môžeme definovať na ktoromkoľvek mieste definície triedy, je dobrým zvykom umiestniť definície lokálnych premenných pred definície metód.

K inštančným premenným sa vo vnútri metód pristupuje podobne, ako keď voláme iné nami napísané metódy triedy - pred názov premennej napíšeme `this` a pridáme bodku:

```
public class MojaPlocha extends WinPane {
        private int pocetBodiek = 1;

        public void zvysPocetBodiek() {
               this.pocetBodiek = this.pocetBodiek + 1;
        }
}
```

V porovnaní s lokálnymi premennými majú inštančné premenné ešte jednu zvláštnosť. Kým lokálne premenné sú po vytvorení neinicializované a prv než s nimi začneme pracovať ich musíme príkazom priradenia inicializovať, pre inštančné premenné to neplatí. Inštančné premenné sú automaticky inicializované na tzv. "defaultnú" (prednastavenú?) hodnotu. Táto "defaultná" hodnota je prirodzene závislá od typu premennej:

- pre `int`, `byte`, `short`, `long`, `double` a `float` je to hodnota 0
- pre `boolean` je to `false`
- pre `char` je to znak s UNICODE kódom 0 (prvý znak UNICODE tabuľky)
- pre premenné referenčného typu (referencujúce napr. objekty tried `String`, `Turtle`, `WinPane`, ...) je to `null`

"Inicializačné metódy" (konštruktory)

Niekedy sa nám pri programovaní vynorí potreba vykonať nejaké príkazy hneď po tom, čo sa objekt nejakej triedy vytvorí (napr. chceme, aby vytvorená kresliaca plocha bola ihneď zelená, alebo aby jej graficky obsah už bol nejako predpripravený). Na tento účel vieme použiť špeciálnu "inicializačnú metódu" (v skutočnosti sa volá inak, ale o tom v neskorších prednáškach). Inicializačná metóda má tú vlastnosť, že sa vykoná vždy po tom, čo je vytvorený objekt danej triedy.

Pre inicializačnú metódu platia tieto pravidlá:

- nemá návratový typ
- volá sa presne tak, ako sa volá trieda
- nemá žiadne parametre

Inicializačnú metódu objektov triedy `Trieda` by sme definovali takto:

```
public Trieda() {
        // inicializacne prikazy
}
```

Polia a "poľové" algoritmy

Doposiaľ vždy, keď sme chceli pracovať s nejakým objektom, musela existovať premenná, ktorá tento objekt referencovala. Problém však vzniká v prípade, že potrebujeme pracovať s veľmi veľa objektami (napr. 100 korytnačiek). Je jasné, že vytvorene 100 premenných referencujúcich objekty triedy `Turtle` nie je práve ideálne riešenie. Analogicky, ak chceme realizovať nejaký výpočet, hodnoty uchovávame v premenných. Avšak, ak je dát veľa, použitie jednoduchých premenných (či už lokálnych alebo inštančných) neprichádza do úvahy. Riešením vyššie spomenutých problémov sú **polia**.

Polia sú špeciálne Java objekty, ktoré nám umožňujú uchovať veľa premenných rovnakého typu "pod jednou strechou". Každý "poľový" objekt sa skladá z dopredu určeného počtu políčok. Jednotlivé políčka sú usporiadané za sebou a očíslované. Číslovanie (indexovanie), tak ako to býva v Jave zvykom, začína od čísla 0. Každé políčko funguje ako samostatná premenná, t.j. uchováva hodnotu alebo referenciu v závislosti od typu políčok poľa.

Deklarácia premennej referencujúcej pole. Keďže polia sú objekty, na to, aby sme s nimi mohli aktívne pracovať, potrebujeme premennú referenčného typu, ktorá by referencovala "poľové" objekty. Premennú, ktorá referencuje polia ("poľové objekty") s políčkami typy `Typ`, deklarujeme takto:

```
Typ[] premennaReferencujucaPole;
```

Za `Typ` môžeme dosadiť ľubovoľný doposiaľ používaný typ premennej:

```
Turtle[] poleReferenciiNaKorytnacky;
String[] poleReferenciiNaRetazce;
int[] poleCisel;
```

Vo vyššie uvedenom príklade premenná `poleReferenciiNaKorytnacky` je premennou referenčného typu, ktorá je schopná referencovať polia ("poľové" objekty), ktorých políčka sú typu `Turtle` - to jest, každé políčko referencovaného poľa uchováva referenciu ("rodné číslo") na nejaký objekt triedy `Turtle`. Premenná `poleCisel` v tomto príklade je schopná referencovať pole ("poľový" objekt), ktorého každé políčko uchováva nejaké celé číslo (hodnotu typu `int`).

Vytvorenie poľa. Povedzme si ešte, ako pole ("poľový" objekt) vytvoríme. Ak chceme vytvoriť pole, ktoré bude mať `n` políčok typu `TypPolicka`, tak použijeme príkaz:

```
new TypPolicka[n];
```

V nasledujúcom príklade sa najprv vytvorí premenná `korytnacky`, ktorá je schopná referencovať polia s políčkami typu `Turtle`. Následne sa príkazom `new` vytvorí pole ("poľový" objekt), ktorý bude mať 10 políčok. Každé z týchto políčok bude schopné uchovávať referenciu na nejaký objekt triedy `Turtle`. Nakoniec sa referencia na vytvorené pole (objekt) uloží do premennej `korytnacky`:

```
Turtle[] korytnacky = new Turtle[10];
```

Políčka vytvoreného poľa fungujú ako klasické premenné. Aký je teda ich obsah po vytvorení poľa ? Pamätajme si, že políčka poľa sú automaticky inicializované "defaultnými" hodnotami (v závislosti od typu políčok), presne tak, ako je to u inštančných (objektových) premenných. V predchádzajúcom príklade budú políčka vytvoreného poľa inicializované hodnotami `null`, keďže typ `Turtle` je referenčným typom (políčka poľa sú referenčného typu).

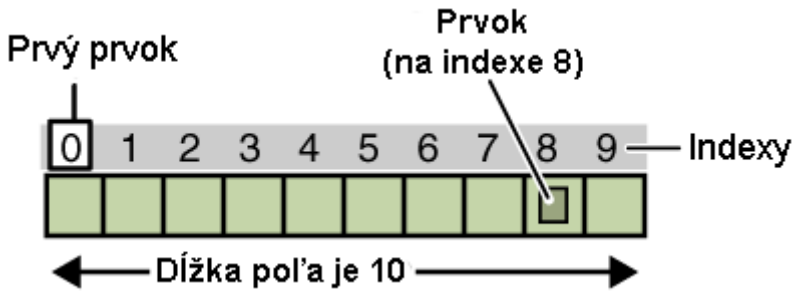
Prístup k políčkam (prvkom) poľa. Po vytvorení poľa chceme prirodzenie s políčkami poľa pracovať. Na prístup ku konkrétnemu políčku poľa používame hranaté zátvorky `[]`, medzi ktoré pridáme index políčka (jeho poradové číslo):

```
// Vytvoríme pole na 10 referencií na korytnačie objekty
Turtle[] korytnacky = new Turtle[10];
// Klasická referenčná premenná referencujúca jeden objekt korytnačky
Turtle korytnacka = new Turtle();
// Vyrobíme jednu korytnačku a necháme ju referencovať políčkom poľa na indexe 0
korytnacky[0] = new Turtle();
// Políčko na indexe 6 (siedme políčko poľa) necháme referencovať tú korytnačku,
// ktorá je referencovaná z premennej korytnacka
korytnacky[6] = korytnacka;
// Políčko na indexe 3 (4. políčko) referencuje tú korytnačku,
// ktorú referencovalo políčko s indexom 0 (1. políčko)
korytnacky[3] = korytnacky[0]

// Povieme korytnačke, ktorá je referencovaná z políčka na indexe 3, aby vykonala metódu
step
korytnacky[3].step(10);
```

Medzi `[]` môžeme napísať nielen konkrétnu hodnotu (celočíselný literál), ale aj ľubovoľný aritmetický výraz, ktorého výsledkom je celé číslo. Toto číslo sa použije ako index na prístup ku konkrétnemu políčku poľa. Keďže polia sú objekty, uvedomme si, že zápis `korytnacky[3]` v skutočnosti znamená: políčko na indexe 3 (4. políčko) poľa referencovaného z premennej `korytnacky`. (Pre lepšiu ilustráciu viď slidy k prednáške.)

Ilustračný obrázok podľa http://java.sun.com/docs/books/tutorial/java/nutsandbolts/arrays.html



Počet prvkov (políčok) poľa. Spomeňme si, že v prípade reťazcov (objektov triedy `String`) sme vedeli počet znakov v reťazci zistiť pomocou metódy `length()`. V prípade polí, ale niečo také nefunguje. Namiesto volania metódy nám Java ponúka špeciálnu konštrukciu **.length** (pozor, nie je to metóda a preto žiadne okrúhle zátvorky), pomocou ktorej vieme zistiť počet prvkov poľa.

Nasledujúci príklad spočíta, v koľkých políčkach je uložená hodnota `null`:

```
Turtle[] korytnacky = ...;

int pocetNullov = 0;
for (int i=0; i<korytnacky.length; i++)
        if (korytnacky[i] == null)
                pocetNullov++;
```

Pamätajme si tiež, že počet políčok poľa (hovoríme aj **dĺžku poľa**) po jeho vytvorení už nemôžeme zmeniť.

- **Pole po vytvorení**: Po vytvorení poľa je v políčkach poľa uložená tzv. "defaultná" hodnota podľa typu políčok poľa. Pamätajme, že ak typom políčok poľa je referenčný typ, tak po vytvorení poľa je v každom políčku hodnota `null`. Inými slovami, žiadne políčko neobsahuje ako svoju hodnotu "rodné číslo" objektu. Ak chceme mať v poli referencie na konkrétne objekty, musíme ich najprv vytvoriť resp. referencie na ne odniekiaľ získať. To je zrejmé, pretože ak je políčko poľa referenčného typu, obsahuje len referenciu ("rodné číslo") na nejaký objekt. Ak však je políčko poľa primitívneho typu (`int`, `double`, ...) obsahuje políčko poľa priamo hodnotu. "Defaultné" hodnoty pre tieto typy možno nájsť v prechádzajúcej prednáške. Pripomeňme, že po vytvorení poľa (cez `new`) sú v tomto prípade jednotlivé políčka poľa vyplnené touto "defaultnou" hodnotou.

Fragment programu zachycujúci prácu s poľom - vytvorenie a naplnenie poľa celých čísel:

```
int[] poleIntov = new int[100];
for (int i=0; i<poleIntov.length; i++)
        poleIntov[i] = 10 + 5 * i;
```

Inicializácia polí

V niektorých situáciach by sa celkom hodilo, keby vytvorené pole bolo rovno aj inicializované nejakými (pre naše potreby) rozumnými hodnotami (nie "defaultnými"). Java samozrejme aj na tento problém ponúka riešenie:

```
int[] cisla = {3, 4, 1, 5, 4, 8};
```

Tento zápis je totožný s týmito riadkami zdrojového kódu:

```
int[] cisla = new int[6];
cisla[0] = 3;
cisla[1] = 4;
cisla[2] = 1;
cisla[3] = 5;
cisla[4] = 4;
cisla[5] = 8;
```

Ako môžeme vidieť, vytvorenie pola a nastavenie hodnôt pre políčka môžeme skombinovať do jedného príkazu. Za znak `=` pridáme do kučeravých zátvoriek `{}` čiarkami oddelenú postupnosť hodnôt pre jednotlivé políčka. Navyše v tomto zozname hodnôt okrem literálov môžeme uviesť aj (aritmetické alebo logické) výrazy.

Príklady:

```
char[] znaky = {'a', 'r', 'x', 't', 'a'};
String[] retazce = {"Ahoj", "Java", "PAZ1a"};

int x = 1;
int[] cisla = {x, x+1, x+2};
```

Užitočné metódy na prácu s (jednorozmerným) poľom

Keďže práca s poľami je pri programovaní dosť častá, Java má už mnoho užitočných metód naprogramovaných (sú jej súčasťou).

Ak chceme **vypísať** obsah poľa zvyčajne použijeme for-cyklus, v ktorom vypíšeme prvky poľa "po jednom". Pomocou metódy `Arrays.toString` vieme nechať si vyrobiť reťazec, ktorý bude obsahovať čiarkami oddelený zoznam hodnôt v poli.

Príklad:

```
int[] pole = ...;

String s = Arrays.toString(pole);
System.out.println(s);
// alebo skrátene
System.out.println(Arrays.toString(pole));
```

Ďalšou častou činnosťou je **kopírovanie** obsahu políčok z jedného poľa do druhého poľa. Tu nám vie pomôcť metóda `System.arraycopy`. Táto metóda má 5 parametrov, v poradí:

- referencia na pole, ktorého obsah políčok ideme kopírovať
- index v poli, od ktorého ideme obsah políčok kopírovať
- referencia na pole, do ktorého ideme obsah políčok kopírovať
- index v poli, od ktorého ideme kopírované prvky ukladať
- počet kopírovaných prvkov

Pozrime sa na túto metódu v príklade:

```
int[] p1 = {1, 2, 3, 4, 5, 6}
int[] p2 = new int[5];
System.arraycopy(p1, 3, p2, 1, 2);
// po skončení bude v p2 obsah: {0, 4, 5, 0, 0}
```

Rôzne metódy (algoritmy) pracujúce s poľom sa môžu zdať na prvý pohľad veľmi zložité. V skutočnosti ide len o obmeny niekoľkých stratégii.

Zistenie, či všetky prvky poľa majú nejakú vlastnosť

Typickým problémom, ktorý sa často rieši, je overenie toho, či všetky prvky poľa (hodnoty alebo objekty referencované z políčok poľa) majú nejakú vlastnosť. Príkladmi vlastností môžu byť tieto:

- hodnota prvku je väčšia ako 100
- hodnota predchádzajúceho prvku v poli je menšia alebo rovná ako hodnota daného prvku
- referencovaná korytnačka je vo vzdialenosti menej ako 50 od nejakého význačného bodu
- hodnota na aktuálnom políčku sa ešte ani raz nevyskytla v políčkach s menším indexom ("naľavo")

Základná stratégia na riešenie tohto problému je táto:

- na začiatku veríme, že všetky prvky majú skúmanú vlastnosť ("sú dobré")
- prechádzame všetky prvky poľa
    - ak aktuálne políčko poľa nemá skúmanú vlastnosť ("je zlé"), tak vieme dať ihneď odpoveď, že nie je pravdou to, že všetky políčka poľa majú skúmanú vlasnosť. V tomto prípade poznáme odpoveď a teda nemá zmysel skúmať ďalšie prvky poľa
- ak sme prešli všetky prvky poľa a nenašli sme "zlý" prvok, tak vieme, že všetky prvky "sú dobré" a teda majú skúmanú vlastnosť

Túto stratégiu v Jave vieme vyjadriť nasledujúcim "pseudokódom". V ňom predpokladáme, že `maVlastnost` je niečo, čo nám povie, či políčko poľa má alebo nemá (`boolean`) skúmanú vlastnosť:

```
boolean vsetkyOK = true;

for (int i=0; i<pole.length; i++)
        if (!maVlastnost(pole[i])) {
                vsetkyOK = false;
                break; // nemusí byť, ale je to trochu časovo efektívnejšie
        }

// po skončení cyklu máme v premennej vsetkyOK informáciu, či všetky prvky poľa majú
skúmanú vlastnosť
```

V prípade, že zisťovanie robíme v samostatnej metóde, môžeme použiť tento variant:

```
public boolean test(...) {
        for (int i=0; i<pole.length; i++)
                if (!maVlastnost(pole[i]))
                        return false;

        return true;
}
```

Pozrime sa, ako by vyzerala aplikovanie tejto stratégie v prípade, že chceme zistiť či všetky prvky majú hodnotu väčšiu alebo rovnú ako zadaná dolná hraničná hodnota `hranica`:

```
public boolean vsetkyVacsieAko(int[] pole, int hranica) {
        for (int i=0; i<pole.length; i++)
                if (pole[i] < hranica)
                        return false;

        return true;
}
```

Túto stratégiu vieme použiť dokonca aj v prípade, keď nás nezaujíma, či všetky prvky poľa majú nejakú vlastnosť, ale či všetky neusporiadané dvojice (alebo aj k-tice) rôznych prvkov majú nejakú vlastnosť:

```
public boolean test(...) {
        // pre dvojice
        for (int i=0; i<pole.length; i++)
               for (int j=i+1; j<pole.length; j++)
                       if (!maVlastnost(pole[i], pole[j]))
                               return false;

        return true;
}
```

Samozrejme, ak by to nebolo v samostatnej metóde, použijeme variant s `break`-om a `boolean` premennou.

Počet prvkov poľa s nejakou vlastnosťou

Problém zistenia počtu prvkov poľa s nejakou vlastnosťou je veľmi podobný predchádzajúcemu problému. Na riešenie tohto problému využívame podobnú stratégiu. Namiesto `boolean` premennej použijeme počítadlo, v ktorom si budeme počítať, koľko prvkov so skúmanou vlastnosťou sme stretli pri prechode prvkami poľa. Pozrime sa na "schému" riešiacu tento problém:

```
int pocitadlo = 0;

for (int i=0; i<pole.length; i++)
        if (maVlastnost(pole[i]))
            pocitadlo++;
```

```
// po skončení cyklu máme v premennej pocitadlo informáciu, koľko prvkov poľa má skúmanú vlastnosť
```

Pozrime sa na príklad metódy, ktorá spočíta, koľko korytnačiek je bližšie k zadanému bodu ako 100:

```
private Turtle[] korytnacky = ...;

public int pocetBlizkychKorytnaciek(double x, double y) {
        int pocitadlo = 0;

        for (int i=0; i<this.korytnacky.length; i++)
                if (this.korytnacky[i].distanceTo(x, y) <= 100)
                        pocitadlo++;

        return pocitadlo;
}
```

Analogicky tento mechanizmus vieme použiť aj na spočítanie počtu dvojíc (k-tíc), ktoré majú nejakú vlastnosť.

Naj prvok poľa

Ďalším častým riešeným problémom je nájdenie "naj" prvku poľa, resp. zistenie jeho hodnoty. Pod pojmom "naj" prvok poľa myslíme taký prvok poľa, ktorý je spomedzi všetkých prvkov poľa nejakým spôsobom najlepší. Pár príkladov najlepšieho prvku poľa:

- prvok s najmenšou hodnotou (minimálny prvok v poli)
- prvok s najväčšou hodnotou (maximálny prvok v poli)
- prvok s najčastejšie sa opakujúcou hodnotou (najčastejšia hodnota v poli)
- korytnačka, ktorá má najmenšiu x-ovú súradnicu (najľavejšia korytnačka)

- korytnačka, ktorá je najbližsie (najďalej) k nejakému bodu

Základná stratégia na riešenie tohto problému je táto (metafora: zápas):

- v každom okamihu si pamätáme **doposiaľ najlepší nájdený prvok** (kandidáta na celkovo najlepší prvok)
- na začiatku si ako doposiaľ najlepší nájdený prvok vyberieme **prvý prvok** poľa
- postupne prechádzame všetky prvky poľa (metafora: doposiaľ nájdený najlepší prvok vyzýva všetky ostatné prvky poľa, ktoré ešte nebojovali v žiadnom "zápase")
    - v každom kroku **porovnávame** kvalitu doposiaľ nájdeného najlepšieho prvku s kvalitou aktuálneho prvku
        - ak je aktuálny prvok lepší, tak od tohto momentu bude on tým doposiaľ nájdeným najlepším prvkom (metafora: aktuálny držiteľ titulu prehral a titul získava ten, kto ho porazil)
- prvok, ktorý je na konci prechodu všetkými prvkami vybraný ako doposiaľ nájdený najlepší prvok je aj celkovo najlepší prvok

V nasledujúcom predpokladajme, že `jeLepsiAko(A, B)` nám povie, či `A` je lepší ako `B` (`A > B`). Algoritmus na nájdenie najlepšieho prvku poľa môžeme v Jave vyjadriť takto:

```
// v idxNaj si budeme pamätať index doposiaľ najlepšieho nájdeného prvku poľa
int idxNaj = 0;
for (int i=1; i<pole.length; i++)
        if (jeLepsiAko(pole[i], pole[idxNaj]))
                idxNaj = i;
```

```
// po skončení máme v idxNaj index najlepšieho prvku poľa
```

Ilustrujme si túto schému na probléme nájdenia najväčšej hodnoty v poli:

```
public double maximalnaHodnota(double[] pole) {
        int idxNaj = 0;
        for (int i=1; i<pole.length; i++)
                if (pole[i] > pole[idxNaj])
                        idxNaj = i;

        return pole[idxNaj];
}
```

Ilustrujme si tento mechanizmus ešte na nájdení korytnačky, ktorá má najmenšiu y-vú súradnicu:

```
private Turtle[] korytnacky = ...;

public Turtle najvyssiaKorytnacka() {
        int idxNaj = 0;
        for (int i=1; i<this.korytnacky.length; i++)
                if (this.korytnacky[i].getY() < this.korytnacky[idxNaj].getY())
                        idxNaj = i;

        return this.korytnacky[idxNaj];
}
```

Rovnako ako všetky skôr spomenuté "finty", aj táto úspešne funguje v prípade, kedy chceme nájsť nejakú najlepšiu dvojicu (alebo k-ticu) rôznych prvkov.

```
int idxNaj1 = 0;
int idxNaj2 = 1;

for (int i=0; i<pole.length; i++)
        for (int j=i+1; j<pole.length; j++)
                if (kvalita(i, j) > kvalita(idxNaj1, idxNaj2)) {
                        idxNaj1 = i;
                        idxNaj2 = j;
                }
```

<< "Inicializačné metódy" (konštruktory) | Obsah | Dvojrozmerné polia >>

Dvojrozmerné polia

Polia sme doposiaľ využívali na uloženie nejakého počtu hodnôt rovnakého typu (napr. pole čísel, či pole referencií na korytnačky). Jedna z predstáv o poli je tá, že je to objekt, ktorý sa skladá z nejakej postupnosti políčok schopných uchovávať hodnoty. Pri programovaní stolových hier, či pri komplikovanejšej matematike (sústavy rovníc) sa nám ale môže objaviť potreba niečoho, čo by uchovávalo dáta v dvoch rozmeroch - v tabuľke. Samozrejme 2-rozmerné polia dokážeme simulovať pomocou jednorozmerného, ale takáto práca nie je veľmi praktická. 2-rozmerné pole, nazývané aj matica, je taktiež Java objekt. Narozdiel o klasických jednorozmerných polí, sú políčka v 2-rozmernom poli prístupné na základe dvojice čísel: index riadka políčka a index stĺpca políčka. Pozrime sa ako deklarovať a ako vytvoriť "poľový" objekt pre 2-rozmerné polia.

```
// Deklarácia premennej schopnej referencovať 2-rozmerné pole
int[][] referenciaNa2DPole;

// Vytvorenie 2-rozmerného poľa (matice) s 10 riadkami a 15 stĺpcami
referenciaNa2DPole = new int[10][15];
```

Ako môžeme vidieť vo vyššie uvedenom príklad, 2-rozmerné polia sa nejako zásadné nelíšia od klasických jednorozmerných polí. Akurát pri deklarovaní premennej referencujúcej 2D pole namiesto jedného páru `[]` zátvoriek dáme dva páry. Podobne je to aj s vytvorením poľa ("poľového" objektu). Za `new` a typ políčka dáme dve dvojice hranatých zátvoriek. Do prvej z nich uvedieme počet riadkov vytváraného poľa a do druhej počet stĺpcov vytváraného poľa.

Aj 2-rozmerné polia podporujú skrátený príkaz kombinujúci vytvorenie poľa s definovaním hodnôt jednotlivých políčok poľa.

```
int[][] matica = {{3, 4, 5, 6}, {10, 11, 2, 3}, {1, 7, 3, 1}};

// je to isté ako:
int[][] matica = new int[3][4];
matica[0][0] = 3;
matica[0][1] = 4;
matica[0][2] = 5;
matica[0][3] = 6;
matica[1][0] = 10;
matica[1][1] = 11;
matica[1][2] = 2;
...
matica[2][3] = 1;
```

3	4	5	6
10	11	2	3
1	7	3	1

Na prístup k políčkam poľa využívame dvojicu indexov:

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

Ľahko sa dovtípime, že pri inicializácií 2D poľa píšeme do kučeravých zátvoriek postupnosť hodnôt pre jednotlivé riadku. Hodnoty pre jeden riadok sú tiež uzavreté vo vnorených kučeravých zátvorkách.

Finty pre 2-rozmerné polia

Pre 2D polia vieme s menšou obmenou využiť stratégie pre jednorozmerné polia predstavené v predošlej prednáške (všetky majú nejakú vlastnosť, či naj prvok). Druhý rozmer poľa so sebou prináša aj možnosť komplikovanejšieho pohybu v poli. Zoberme si implementáciu hry piškvorky, v ktorej je obsadenosť políčok hracej plochy uložená v nejakom 2-rozmernom poli. Aby sme zistili, či posledným ťahom nedošlo k dosiahnutiu výhernej konfigurácie, musíme overiť, či niekde v niektorom z 8 smerov nie je uložených 5 hracích kameňov rovnakej farby. V prípade hry piškvorky sa nám oplatí mať metódu, ktorá nám pre dané políčko (určené súradnicami `[riadok][stĺpec]`) povie, koľko kameňov rovnakej farby uložených za sebou stretneme, ak sa z daného políčka vyberieme niektorým s 8 smerov. Všimnime si, že v skutočnosti nám stačí skúmať len 4 smery (prečo ?):

- zľava doprava (rastie index stĺpca)
- zhora nadol (rastie index riadka)
- po uhlopriečke zľava doprava a zhora nadol (rastie aj index stĺpca aj index riadka)
- po uhlopriečke sprava doľava a zhora nadol (index stĺpca klesá, index riadka rastie)

V prvom prípade platí, že ak aktuálne políčko má súradnice `[r][s]`, tak jeho sused pri pohybe zľava doprava bude mať súradnice `[r+0][s+1]`. Analogicky, v prípade pohybu zhora nadol, bude mať sused súradnice `[r+1][s+0]`. V zostávajúcich 2 prípadoch bude sused na súradniciach `[r+1][s+1]`, resp. `[r+1][s-1]`. Súradnice suseda sa líšia stále o akúsi dvojicu hodnôt: pre `[r+1][s-1]` je to napríklad `[1, -1]`. Túto dvojicu môžeme v matematicko-fyzikálnej terminológii nazvať **vektorom posunutia**. Prirodzene, každý "priamočiary" pohyb v 2D poli vieme stále charakterizovať nejakým takýmto vektorom posunutia. Typicky ale potrebujeme skúmať viacero vektorov posunutia. V tejto situácii je preto veľmi výhodné mať jednotlivé vektory "posunutia" uložené v niečom "poľovom". Tým získame možnosť využiť výhody indexovania políčok. Keďže vektor má 2 zložky, veľmi výhodne sa javí použiť 2-rozmerné pole s 2 stĺpcami a `k` riadkami. Každý riadok z tejto tabuľke vie uchovávať vektor posunutia pre jeden z `k` smerov.

Pohyb v 2D poli môžeme zachytiť takto:

```
int[][] pole = ...;
...
int smer = ...;
while (existujePolicko(r, s)) {
       spracujPolicko(pole[r][s]);

       r = r + smery[smer][0];
       s = s + smery[smer][1];
}
```

Po vypočítaní súradníc nového políčka sa nám môže stať, že políčko s vypočítanými indexami neexistuje ("vypadli" sme mimo poľa). Preto je nutné pred spracovaním políčka overiť, či indexy, ktoré máme k dispozícii, sú naozaj platnými indexami do 2-rozmerného poľa.

<< Polia a "poľové" algoritmy | Obsah | Výnimky (odchytávanie) >>

Výnimky (odchytávanie)

Výnimky sú riadiaci mechanizmus, ktorý sa používa, keď nejaká operácia nedokáže prebehnúť štandardným spôsobom alebo nedokáže vrátiť očakávanú hodnotu. Uveďme si pár príkladov:

- Metóda, ktorá počíta priemer prvkov číselného poľa dostane na vstup pole dĺžky nula. Keďže vzorec na priemer je podiel súčtu prvkov a počtu prvkov, pokúsi sa deliť nulou, čím nastane nečakaná situácia alebo výnimočný stav.
- Vytvárate nové pole `Turtle[] korytnacky = new Turtle[-5];` a program sa dozvie, že vytvárate pole dĺžky -5, ale také pole vytvoriť nevie.
- Idete zapísať hodnotu na 15-ty prvok 10-prvkového poľa.
- Chcete pridať hodnotu do poľa, ktoré ste nevytvorili cez new

V starších programovacích jazykoch vám takáto situácia spôsobí okamžité ukončenie alebo zamrznutie celej aplikácie, čo je veľmi nepríjemné. Často program iba vypíše ničnehovoriacu hlášku ako "segmentation fault" alebo "fatal error", ktorá mu nepovie ani to, kde a ani aká výnimočná situácia nastala.

Aké výhody oproti tomuto má výnimka? Za prvé, výnimka má svoj názov, ktorý nám (po preklade z angličtiny) naznačí, aká udalosť sa stala.

Akých výnimiek by sme sa dočkali v prípade predchádzajúcich príkladov?

- `java.lang.ArithmeticException: / by zero` - výnimka pri aritmerickej operácii: delenie nulou
- `java.lang.NegativeArraySizeException` - výnimka zápornej veľkosti poľa
- `java.lang.ArrayIndexOutOfBoundsException: 15` - výnimka indexu poľa mimo hranice poľa: 15-ty prvok
- `java.lang.NullPointerException` - výnimka ktorá sa vyhodí ak namiesto objektu máme v premennej `null` a chceme cez túto premennú volať metódu.

Vytvorme si príklad metódy korytnačky, ktorá vráti `true` ak priemer prvých `k` prvkov poľa na vstupe je väčší ako 0, inak vráti `false`. Chceme aby táto metóda vždy vrátila nejakú hodnotu, to znamená, že by mala byť odolná voči všetkým zlým vstupom. Napíšme si najprv metódu, ktorá verí, že všetky vstupy budú pekné a nespôsobia výnimočný stav:

```
public boolean kladnyPriemer(int[] pole, int k) {
  int priemer = 0;
  int sucet = 0;
  for (int i = 0; i < k; i++) {
    sucet += pole[i];
  }
  priemer = sucet / k;
  if (priemer > 0) {
    return true;
  }
  else {
    return false;
  }
}
```

Čo by sa stalo v prípade, ak v premennej pole bude `null`? Telo for cyklu vyhodí výnimku `NullPointerException`, lebo chceme pristupovať do poľa, ktoré neexistuje. Vyrobíme si podmienku,

ktoré nám zabezpečí, že ak nastane tento prípad, tak sa v tomto prípade korektne vráti ako výstupná hodnota metódy **false**. (to znamená, že priemer prvých `k` prvkov nie je väčší ako nula).

```
public boolean kladnyPriemer(int[] pole, int k) {
  if (pole == null)
     return false;
  int priemer = 0;
  int sucet = 0;
  try {
    for (int i = 0; i < k; i++) {
      sucet += pole[i];
    }
    priemer = sucet / k;
    if (priemer > 0) {
      return true;
    }
    else {
      return false;
    }
  } catch (NullPointerException e) {
    return false; // priemer nie je vacsi ako nula lebo priemer neexistuje
  }
}
```

`null` v premennej pole však nie je jediné, čo môže postihnúť bezchybné fungovanie metódy.

Premenná `k` môže mať hodnotu nula. V tom prípade sa for cyklus nevykoná ani raz (lebo 0 < 0 je false) a program sa bude snažiť deliť nulou teda vyhodí `ArithmeticException`. Dodáme teda ďalšiu podmienku.

Ďalšia chyba vstupu môže byť, že pole môže mať dĺžku nula. Zasa pridáme podmienku.

Další problém môže byť, keď `k` je väčšie ako dĺžka poľa. V tomto prípade program vyhodí výnimku `ArrayIndexOutOfBoundsException` pri snahe pripočítať prvok poľa ktorý je prvý za posledným prvkom poľa. Zasa pridáme ďalšiu podmienku ktorá nám opraví `k` na dĺžku poľa.

Výsledný kód vyzerá nasledovne:

```
public boolean kladnyPriemer(int[] pole, int k) {
  if (pole == null || pole.length==0 || k<=0)
     return false;
  if (k > pole.length)
     k = pole.length;
  int priemer = 0;
  int sucet = 0;
  for (int i = 0; i < k; i++) {
    sucet += pole[i];
  }
  priemer = sucet / k;
  if (priemer > 0) {
    return true;
  }
  else {
    return false;
  }
}
```

Vyrobme si teraz nový program, ktorý bude vracať súčet čísiel, ktoré sa nachádzajú v reťazci. Budeme potrebovať reťazec rozdeliť na slová a každé slovo sa pokúsiť pretransformovať na `int`. Na to použijeme funkciu `Integer.parseInt(...)`.

```
String retazec = "235";
int cislo = new Integer.parseInt(reťazec); //do premennej cislo sa pridaí číslo 235
```

Samotný kód metódy vyzerá nasledovne.

```
public int sucetCisiel(String vstup) {
  int sucet = 0;
  vstup = vstup + " ";                          // aby sme zachytili aj posledné slovo
  String cislo = "";
  for(int i = 0; i < vstup.length(); i++) {
     if(vstup.charAt(i)==' ') {                 // prišli sam na koniec slova
        int hodnota = Integer.parseInt(cislo);
        sucet = sucet + hodnota;
        cislo = "";                             // pripravíme si reťazec pre nové číslo
     } else {                                   // sme na cifre, pridáme ju na koniec
čísla
        cislo = cislo + vstup.charAt(i);
     }
  }
  return sucet;
}
```

Tento program však môže pri zlom vstupe tiež vyhodiť výnimku, konkrétne `NumberFormatException`, a to vtedy, ak budeme mať viac medier medzi číslami, alebo ak použijeme vo vstupnom reťazci aj iné znaky ako cifry. V takomto prípade si už jednou podmienkou nepomôžeme a museli by sme robiť kadejaké testy, prechádzať reťazec a podobne.

Veľká výhoda výnimky je, že ju môžeme **odchytiť**, a ak je to možné, vysporiadať sa s týmto stavom v programe. Napríklad môžeme poprosiť používateľa nech zadá iný vstup alebo mu minimálne namiesto výnimky vypísať nejakú hlášku v ľudskej reči.

Odchytávanie výnimky robíme tak, že najprv si ohraničíme oblasť, kde očakávame, že by mohla nastať výnimka s ktorou sa chceme vysporiadať do bloku **try**, a za ním napíšeme blok **catch** ktorý sa aktivuje, iba ak nastala výnimka. Ak výnimka nenastane prebehnú všetky príkazy v bloku try akoby tam blok **try** ani nebol. Blokov **catch** môže byť jeden alebo viac - pre každý typ výnimky jeden. Syntax je teda nasledovná:

```
try {
  //príkazy, ktoré ak vyhodia výnimku skočíme na blok catch
} catch (typ_výnimky1 e) {
  vysporiadanie sa s prvým typom výnimky
} catch (typ_výnimky2 e) {
  vysporiadanie sa s druhým typom výnimky
}
```

Čo teda urobíme v našom príklade? Ak sa nám nepodarí stransformovať číslo v reťazci na číslo typu `int`, telo for cyklu vyhodí výnimku `NumberFormatException`. Vyrobíme si teda bloky **try** a **catch**, ktoré nám zabezpečia kód tým, že sa takáto výnimka odchytí, dané slovo budeme ignorovať a používateľovi vypíšeme hlášku, že mal nejaké chyby vo vstupe.

```
public int sucetCisiel(String vstup) {
  int sucet = 0;
  vstup = vstup + " ";
  String cislo = "";
  for(int i = 0; i < vstup.length(); i++) {
     if(vstup.charAt(i)==' ') {
        try {
            int hodnota = Integer.parseInt(cislo);
            sucet = sucet + hodnota;
        } catch (NumberFormatException e) {
            System.err.println("Slovo \"" + cislo + "\" neviem transformovať! Ignorujem
ho.");
        }
```

```
            cislo = "";
        } else {
            cislo = cislo + vstup.charAt(i);
        }
    }
    return sucet;
}
```

Ak nastane výnimka a nemáme príslušný **catch** blok, ktorý by odchytával správnu výnimku, program skončí s chybou. Aby to nebolo také jednoduché, tak za blokmi **catch** môže nasledovať ešte jeden blok **finally**. Podľa definície je to blok, ktorý sa vykoná vždy bez ohľadu na to, či výnimka vyhodená vo vnútri **try** bloku bola odchytená alebo nie. Používa sa obvykle na korektné ukončenie sieťového spojenia, na korektné zatvorenie súboru alebo odhlásenie z databázy. Ak výnimka nebola odchytená niektorým **catch** blokom, tak sa ešte vykoná blok **finally** a až potom program skončí s chybou.

```
try {
  //príkazy, ktoré ak vyhodia výnimku skočíme na blok catch
} catch (typ_výnimky1 e) {
  vysporiadanie sa s prvým typom výnimky
} catch (typ_výnimky2 e) {
  vysporiadanie sa s druhým typom výnimky
} finally {
  // príkazy ktoré sa vykonajú bez ohľadu na to, či bola výnimka odchytená niektorým
catch blokom
}
```

K výnimkám je ešte potrebné spomenúť, že Java pozná dva typy výnimiek:

- **RuntimeException** alebo behové výnimky, ktoré sa obvykle dajú opraviť zmenou programu. Doteraz spomínané výnimky sú všetky tohto typu.
- **Exception** alebo bežné výnimky, ktoré sú obvykle spôsobené chybou používateľa a zmenou programu sa im nevieme vyhnúť. Tieto výnimky sa musia odchytávať. Eclipse na to upozorňuje. S jednou takouto výnimkou, `FileNotFoundException`, sa stretneme pri otváraní súborov.

Práca so súbormi a adresármi operačného systému

Každý z vás už určite prišiel do styku s adresármi a súbormi. V jave používame triedu File ktorej objekty identifikujú buď jeden súbor alebo jeden adresár. Ako súbor tak aj adresár na lokálnom počítači sú jednoznačne určené svojou cestou. Cesta môže byť *úplná*, alebo *relatívna*.

Úplná cesta vo windowse je napríklad "C:\Windows\System32\shell32.dll" alebo "C:\Windows", v linuxe napríklad "/home/peter/heslo.txt" alebo "/etc".

Relatívna cesta je vždy vzhľadom k nejakému adresáru napríklad "System32\shell32.dll" je relatívna cesta vzhľadom k "C:\Windows".

Položky v ceste sú vo windowse oddelené spätnou lomkou "\" alebo obyčajnou lomkou "/" (niektorí o tom nevedia), v linuxe vždy obyčajnou lomkou. Ak budete chcieť používať spätné lomky dajte si pozor na to, že v reťazcoch spätná lomka označuje špeciálny znak, takže musí byť zdvojená, aby sa brala ako znak '\'.

```
// úplná cesta k adresáru s použitím spätných lomiek
File adresar = new File("C:\\Windows\\System32");
// úplná cesta k súboru s použitím obyčajných lomiek
File subor1 = new File("C:/Windows/system.ini");
// relatívna cesta k súboru vzhľadom k adresáru C:\Windows\System32
File subor2 = new File(adresar, "shell32.dll");
// relatívna cesta k súboru vzhľadom k aktuálnemu adresáru
File subor3 = new File("heslo.txt");
```

Ak spúšťate program z Eclipsu tak aktuálny adresár je adresár projektu, inak je to adresár z ktorého bol program spustený. Ak niekedy náhodou budete potrebovať zistiť meno aktuálneho adresára použite nasledovný kód:

```
String menoAktAdresara = System.getProperty("user.dir");
System.out.println("Nazov aktualneho adresara je: " + menoAktAdresara);
```

Cez metódy triedy File sa nevieme dostať k obsahu súboru iba k jeho metadátam, teda informáciám o tomto súbore. Metadáta o súboroch bežne vidíme v súborovom manažéri (napr. Explorer, Windows commander, Total commander, Krusader, mc a iných)). Metadáta obsahujú napríklad veľkosti súborov, obsahy adresárov či práva na čítanie/zápis/spúšťanie.

Veľmi dôležitou vecou, ktorú si je potrebné uvedomiť je, že daný súbor alebo adresár nemusí reálne existovať. Java pri kompilácii zdrojového kódu netestuje, či cesta špecifikovaná pri vytváraní objektu triedy File má aj spoj proťajšok v súborovom systéme. Predstavte si to ako adresu domu. Môžeme napísať na pohľadnicu adresu *Ján Javák, Kompilačná 8, Košice*, ale to ešte neznamená, že také mesto, ulica alebo Jano Javák bývajúci na tejto adrese existujú.

Prehľad niektorých užitočných metód znázorňuje nasledujúca tabuľka (neučte sa ju naspamäť).

metóda	činnosť
String getPath()	vráti úplný názov súboru alebo adresára
String getName()	vráti názov súboru alebo adresára (bez cesty)

boolean exists()	vráti true, ak súbor/adresár existuje
boolean isFile()	zistí, či inštancia zodpovedá súboru
boolean isDirectory()	zistí, či inštancia zodpovedá adresáru
long length()	vráti veľkosť súboru
createNewFile()	vytvorí súbor, ak neexistuje. Môže vyvolať výnimku **IOException** ak nemáme dostatočné práva alebo neexistuje adresár v ktorom by sa mal tento súbor vytvoriť.
mkdir()	vytvorí adresár, zodpovedajúci poslednej položke v ceste, nadadresár musí existovať
mkdirs()	vytvorí celú adresárovú štruktúru v ceste
renameTo(File)	premenuje súbor podľa inej inštancie triedy **File**
delete()	odstráni súbor
String[] list()	vráti pole názvov súborov/podadresárov v adresári. Vráti **null** ak nejde o existujúci adresár
File[] listFiles()	vráti pole inštancií triedy File zodpovedajúcich súborom/podadresárom v adresári. Vráti **null** ak nejde o existujúci adresár

Predstavte si, že chceme vypísať všetky súbory a podadresáre. Použijeme metódu list():

```
File adresár = new File("D:/MP3/Jackson");
String[] suboryAPodadresare = adresár.list();
if (suboryAPodadresare != null) {
  for(int i = 0; i < suboryAPodadresare.length; i++) {
    System.out.println(suboryAPodadresare[i]);
  }
}
```

Ak by sme však okrem mien chceli vedieť aj veľkosti všetkých mp3 súborov, už si s metódou list() navystačíme, lebo okrem názvu potrebujeme ku každej položke zistiť, či to je súbor, a akú má veľkosť. Použijeme teda listFiles().

```
File adresár = new File("D:/MP3/Jackson");
File[] súboryAPodadresáre = adresár.listFiles();
if (súboryAPodadresáre != null) {
  for(int i = 0; i < súboryAPodadresáre.length; i++) {
    File súborČiPodadresár = súboryAPodadresáre[i];
    if(súborČiPodadresár.isFile()) {
      String menoSúboru = súborČiPodadresár.getName();
      long veľkosť = súborČiPodadresár.length();
      if(menoSúboru.endsWith("mp3")) {
        System.out.println(menoSúboru + " " + veľkosť);
      }
    }
  }
}
```

Textové súbory

Doteraz sme pracovali iba s malým počtom vstupných dát. Keď už budeme písať väčšie programy, tie už budú pravdepodobne komunikovať s okolím aj inak ako čítaním textových políčok z vyskakovacieho okna. Najčastejšie prebieha výmena dát so súbormi v operačnom systéme, databázou alebo s inými počítačmi po sieti. Niekedy programy komunikujú aj s periférami ako tlačiareň a skener, špeciálnymi prístrojmi v nejakom podniku alebo robotmi.

Zápis do textového súboru - trieda PrintWriter

Trieda File nám umožňuje identifikovať adresár alebo súbor na disku. Ale v prípade, že chceme pracovať s obsahom súborov, potrebujeme využiť niečo iné. Najjednoduchší zápis do textového súboru je s použitím triedy PrintWriter.

Práca s textovými súbormi sa vždy skladá z 3 krokov.

1. otvorenie súboru, ktoré je súčasťou konštruktora nejakého čítača (napr. Scanner) alebo zapisovača (napr. PrintWriter).
2. práca s obsahom súboru, teda čítanie alebo zapisovanie
3. zatvorenie súboru

Na zapisovanie do súboru cez PrintWriter budeme používať metódy print(...) a println(...). Ich fungovanie je totožné s výpisom do konzoly cez `System.out.print(...)` a `System.out.println(...)`, akurát že výpis nie je do konzoly ale do otvoreného súboru na disku.

Uveďme si jednoduchý príklad aj s výsledným obsahom súboru.

```
File subor = new File("C:/text1.txt");
PrintWriter pw = null;
try {
  pw = new PrintWriter(subor); // otvoríme súbor C:/text1.txt na zápis. Ak v ňom niečo bolo, tak sa to vymaže.
  pw.print("Jano"); // napíšeme do súboru "Jano" a čakáme na ďalší zápis konci tohto riadku
  pw.println("Javák"); // napíšeme do súboru "Javák", ideme do nového riadku a čakáme
  pw.println(); // ideme do nového riadku a čakáme
  pw.println("0903 888 222"); // napíšeme do súboru "0903 888 222", ideme do nového riadku a čakáme
} catch (FileNotFoundException e) {
  System.out.println("Súbor " + subor.getName() + " neexistuje");
} finally {
  if(pw!=null)
     pw.close(); // uložíme a zavrieme súbor C:/text1.txt
}
```

Ak bol zápis úspešný výsledný súbor vyzerá nasledovne:

```
JanoJavák
0903 888 222
```

Určite ste zbadali bloky **try** a **catch**. V tomto prípade je ich použitie nutné, inak vám java váš kód neskompiluje. Výnimka `java.io.FileNotFoundException`, teda výnimka nenájdeného súboru, je totiž považovaná za bežnú výnimku, ktorá musí byť odchytená.

Ak súbor, ktorý otvárame cez PrintWriter, neexistuje, ale existuje adresár v ktorom má byť uložený, tak sa tento súbor vytvorí. Pokiaľ súbor už existoval, premaže sa a PrintWriter doňho zapisuje od prvého znaku.

Poznámka: Aby ste sa neupísali k smrti, tak Eclipse vám vie odchytávanie výnimiek napísať za vás. Označte si oblasť, ktorú chcete mať v **try** bloku, kliknite prvým tlačidlom myši a vyberte možnosť *Surround With > Try/catch Block*.

Čítanie textového vstupu - trieda Scanner

Trieda Scanner slúži na čítanie textových súborov, vstupu z konzoly alebo z reťazca. To, čo z toho bude čítať, mu povieme pri vytváraní novej inštancie Scannera. Uveďme si používané príklady otvárania textového vstupu Scannerom:

```
File subor = new File("C:\\vstup.txt");
Scanner scanerZoSuboru = new Scanner(subor);
Scanner scanerZKonzoly = new Scanner(System.in);
Scanner scanerZReťazca = new Scanner("Táto trieda sa mi páči.");
Scanner scanerZReťazca2 = new Scanner("C:\\vstup.txt"); // POZOR - ČASTÁ CHYBA!!! toto
nebude čítať súbor C:\vstup.txt ale reťazec "C:\vstup.txt".
```

`System.in` je štandardný vstup, teda obyčajne vstup z klávesnice. Naproti tomu existuje ešte štandardný výstup `System.out`, ktorý už poznáte v spojení s vypisovaním na konzolu cez metódy `print(...)` a `println()`. Java pozná ešte štandardný chybový výstup `System.err`, do ktorého sa dá písať rovnako cez metódy `print(...)` a `println()`. Eclipse konzola zobrazuje výstup na štandardný chybový výstup červeným písmom.

Po vytvorení už všetky inštancie fungujú rovnako, bez ohľadu na zdroj textu, ktorý čítajú. Jediný rozdiel je v Scanneri zo súboru, ktorý by sa mal po použití zatvoriť metódou close(), aby sa zatvoril aj súbor z ktorého sa čítalo.

Filozofia fungovania Scannera je cez použitie spríbuzdnených metód:

- `boolean hasNextXXX()` - vráti **true** ak je možné zo vstupu prečítať hodnotu typu XXX, inak vráti **false**
- `XXX getXXX()` - vráti hodnotu typu XXX zo vstupu, ak to je možné

Za `XXX` si môžete dosadiť primitívne typy ako napríklad int,double, boolean alebo ak nedosadíte nič a ide o `String`. Scanner metódou `getXXX()` číta po najbližší *oddeľovač*. Prednastaveným oddeľovačom je ľubovoľný znak, ktorý je takzvaným whitespace znakom, teda znakom, ktorý nie je vidieť. Typické whitespace znaky sú medzera `' '` a tabulátor `'\t'`.

Špeciálnym prípadom spríbuznených metód je boolean hasNextLine() a String nextLine(), ktoré za oddeľovač považujú (neviditeľný) znak konca riadku. Výsledkom metódy String nextLine() je teda jeden riadok vstupu od aktuálnej pozície.

Keď Scanner prečíta hodnotu cez niektorú `getXXX()` aktuálna pozícia sa presunie za túto hodnotu.

Prehľad niektorých metód Scannera ukazuje nasledujúca tabuľka:

metóda	činnosť
boolean hasNextLine()	vráti **true** ak je možné čítať ďalší riadok (t.j. nie sme na konci vstupu/súboru)
String nextLine()	vráti ďalší riadok od aktuálnej pozície
boolean hasNextInt()	vráti **true** ak je možné čítať ďalšie číslo typu **int** (t.j. dá sa skonvertovať na **int**)
int nextInt()	vráti ďalšiu hodnotu typu **int** od aktuálnej pozície

boolean hasNextDouble()	vráti **true** ak je možné čítať ďalšie číslo typu **double** (t.j. dá sa skonvertovať na **double**)
double nextDouble()	vráti ďalšiu hodnotu typu **double** od aktuálnej pozície
boolean hasNext()	vráti **true** ak je možné čítať ďalší reťazec
String next()	vráti reťazec od katuálnej pozície až po najbližší oddeľovač
useDelimiter(String)	nastaví nový oddeľovač - ak napríklad nastavíme oddeľovač na tabulátor `"\t"` tak medzera sa pri použití metódy next() bude brať ako bežný znak

Načítanie 5 čísiel z klávesnice môžeme spraviť nasledovne:

```
int[] pole = new int[5];
System.out.print("Zadajte 5 celých čísiel: ");
Scanner scanner = new Scanner(System.in);
for(int i = 0; i < 5; i++) {
  pole[i] = scanner.nextInt();
}
... // práca s naplneným poľom
```

Tieto čísla môžete zadať napríklad tak, že ich napíšete v jednom riadku oddelené medzerami alebo každé do nového riadku (aj nový riadok je whitespace znakom).

Načítanie 5 riadkov zo súboru môžeme spraviť nasledovne:

```
File súbor = new File("C:/vstup.txt");
String[] riadky = new String[5];
Scanner scanner = null;
try {
    scanner = new Scanner(súbor);
    for(int i = 0; i < 5; i++) {
      if(scanner.hasNextLine()) {
        riadky[i] = scanner.nextLine();
      } else {
        System.out.println("Chýba riadok!");
      }
    }
} catch (FileNotFoundException e) {
  System.out.println("Súbor " + subor.getName() + " neexistuje");
} finally {
  if(scanner!=null)
    scanner.close(); //nech sa stane čokoľvek, korektne uzavrieme súbor
}
```

Poďme si ešte ukázať načítanie do dvojrozmerného poľa zo súboru. Majme nasledovný súbor, ktorý obsahuje čísla ktoré by sme chceli dostať do políčok dvojrozmerného poľa:

```
5 84 -8 6
12 561 5 0
0 1 -1 0
```

Na prvý pohľad vidíme, že pôjde o dvojrozmerné pole celých čísiel veľkosti 3x4 ktoré si nazveme `matica`. Čítať zo súboru budeme po riadkoch, a každý riadok, ktorý získame metódou nextLine(), ktorý je typu String, budeme novým Scannerom čítať ako čísla oddelené whitespace oddeľovačom.

```
File súbor = new File("C:/matica.txt");
int[][] matica = new int[3][4];
Scanner scannerSuboru = null;
try {
    scannerSuboru = new Scanner(súbor);
    for(int i = 0; i < 3; i++) { // čítame 3 riadky
```

```
      String riadok = scannerSuboru.nextLine();
      Scanner scannerRiadku = new Scanner(riadok);
      for(int j = 0; j < 4; j++) { // čítame 4 čísla v každom riadku
         matica[i][j] = scannerRiadku.nextInt(); // na i-ty riadok a j-ty stĺpec do
dvojrozmerného poľa matica vložíme j-te číslo i-teho riadku zo súboru
      }
    }
} catch (FileNotFoundException e) {
  System.out.println("Súbor " + subor.getName() + " neexistuje");
} finally {
  if(scannerSuboru!=null)
    scannerSuboru.close(); //nech sa stane čokoľvek, korektne uzavrieme súbor
}
```

Čo však urobíme v prípade, že sa do toho súboru nepozrieme a nevieme, že koľko má matica v súbore riadkov a stĺpcov? Najprv si odhadneme maximálnu možnú veľkosť matice (v našom príklade to odhadneme na 6x6). Potom v tomto poli vyplníme iba toľko riadkov a stĺpcov, koľko ich bolo vo vstupnom súbore. Zároveň si musíme po načítaní uložiť to, koľko riadkov a stĺpcov reálne obsahuje dáta. Uložíme si to v premenných `pocetRiadkov` a `pocetStlpcov`. Počas napĺňania dvojrozmerného poľa si budeme spravovať premenné `r` a `s`, ktoré nám budú kopírovať našu aktuálnu pozíciu riadku a stĺpca.

```
File súbor = new File("C:/matica.txt");
int[][] matica = new int[6][6]; //horný odhad veľkosti matice v súbore
Scanner scannerSuboru = null;
int r = -1; // zatiaľ nie sme ani na nultom riadku, až potom keď ho pôjdeme naozaj čítať
zvýšime pozíciu na nulu.
int s = -1;
try {
    scannerSuboru = new Scanner(súbor);
    while(scannerSuboru.hasNextLine()) { // čítame pokiaľ ešte sú nejaké riadky v súbore
      r++; // ideme čítať ďalší riadok, zvýšime teda číslo riadku o 1
      String riadok = scannerSuboru.nextLine();
      Scanner scannerRiadku = new Scanner(riadok);
      s = -1; // zatiaľ nie sme ani na nultom stĺpci, až potom keď ho pôjdeme naozaj čítať
zvýšime pozíciu na nulu.
      while(scannerRiadku.hasNextInt()) { // čítame pokiaľ ešte sú nejaké čísla v tomto
riadku
         s++;  // ideme čítať ďalší stĺpec, zvýšime teda číslo stĺpca o 1
         matica[r][s] = scannerRiadku.nextInt(); // na pozíciu v danom riadku a stĺpci
vložíme do dvojrozmerného poľa matica číslo z rovnakej pozície v súbore
      }
    }
    pocetRiadkov = r + 1; // naposledy sme napĺňali r-tý riadok teda riadkov máme o 1 viac
(máme totiž aj nultý riadok)
    pocetStlpcov = s + 1; // naposledy sme napĺňali s-tý stĺpec teda stĺpcov máme o 1 viac
} catch (FileNotFoundException e) {
  System.out.println("Súbor " + subor.getName() + " neexistuje");
} finally {
  if(scannerSuboru!=null)
    scannerSuboru.close(); //nech sa stane čokoľvek, korektne uzavrieme súbor
}
```

Výsledkom tohto postupu v prípade rovnakého vstupného súboru ako v predchádzajúcom príklade budú:

- v premennej `pocetRiadkov` číslo 3
- v premennej `pocetStlpcov` číslo 4
- v dvojrozmernom poli matica tieto hodnoty:

5	84	-8	6	0	0
12	561	5	0	0	0
0	1	-1	0	0	0
0	0	0	0	0	0

0	0	0	0	0	0
0	0	0	0	0	0

Urobme si ešte výpis používanej časti tejto matice na konzolu:

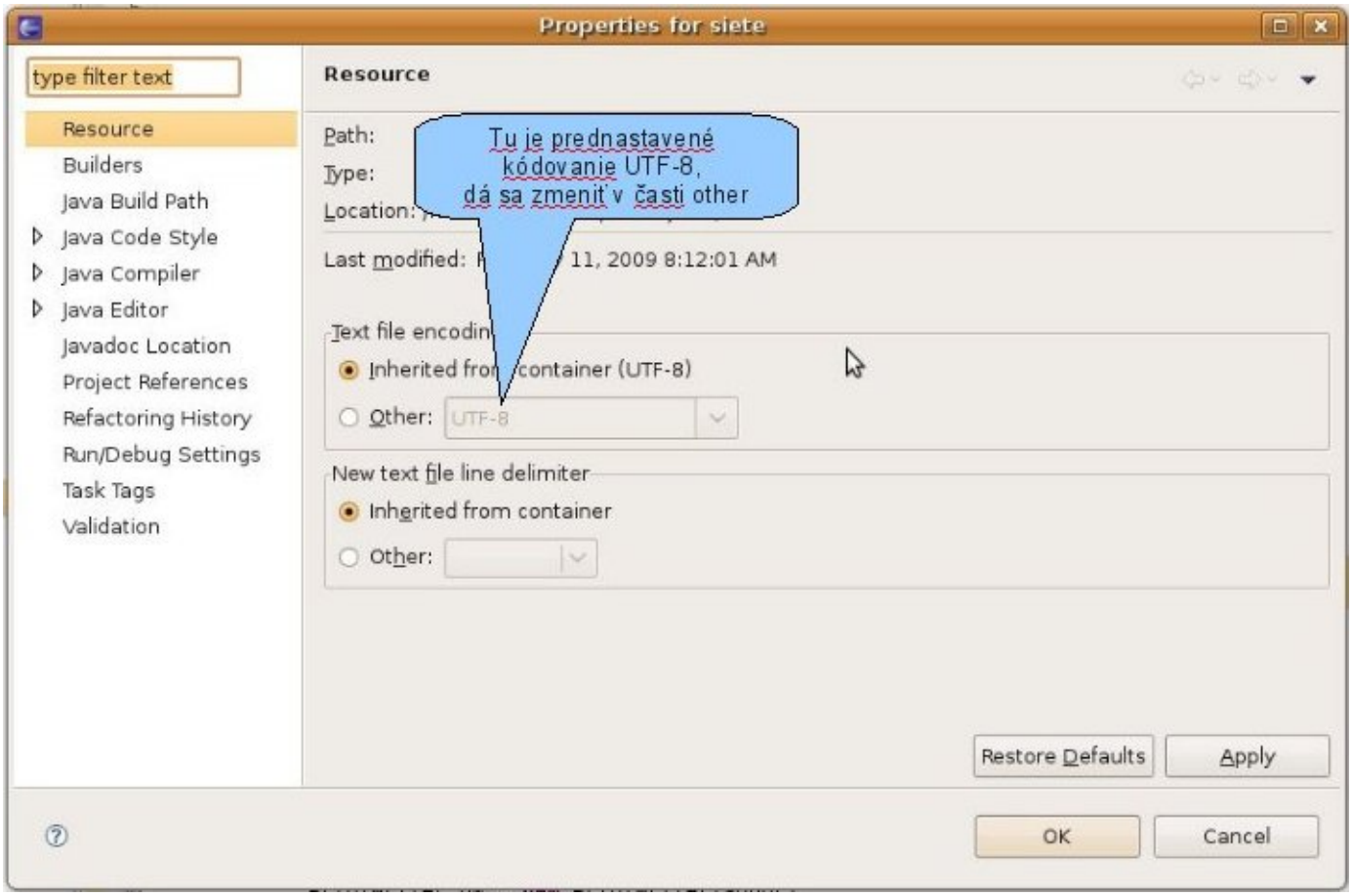
```
for(int i = 0; i < pocetRiadkov; i++) {
  for(int j = 0; j < pocetStlpcov; j++) {
     System.out.print(matica[i][j]+ " ");
  }
  System.out.println(); // po výpise všetkých stĺpcov jedného riadku skočíme na nový riadok
v konzole
}
```

Výsledkom je v našom príklade takýto výpis:

```
5 84 -8 6
12 561 5 0
0 1 -1 0
```

Práca s rôznymi znakovými kódovaniami súborov

Zaujímavými a niekedy užitočnými sú konštruktory tried PrintWriter a Scanner s dvoma parametrami, kde prvým parametrom je otváraný súbor a druhým kódovanie znakovej sady. Keď sa použije konštruktor iba s jedným parametrom nastaví sa znaková sada projektu. Pozrieť si ju môžete kliknutím pravým tlačidlom myši na projekt a vybratie možnosti *Properties*. Uvidíte podobné okno:



Bežný Slovák sa stretáva najmä so znakovou sadou windows-1250 a UTF-8. Po otvorení súboru je už práca s PrintWriterom alebo Scannerom úplne rovnaká. Použitie si ukážeme na príklade s PrintWriterom.

```
File subor = new File("C:/textUTF8.txt");
PrintWriter pw = null;
try {
  pw = new PrintWriter(subor,"UTF-8"); //otvoríme súbor a zapisovať budeme v kódovaní UTF-8
  pw.print("ľščťžýáíéúôä"); // niečo do súboru napíšeme
} catch (FileNotFoundException e) {
  System.out.println("Súbor " + subor.getName() + " neexistuje");
} finally {
  if (pw!=null)
    pw.close(); // uložíme a zavrieme súbor
}
```

Ak máte chuť použiť iné znakové sady kliknite si na stránku so zoznamom podporovaných kódovaní.

Vytvárame prvé triedy

Predstavte si, že si chcete vytvoriť program na pohodlné vyhľadávanie vo vašej zbierke DVD filmov. O každom DVD viete:

- názov filmu
- hercov, ktorý v ňom hrali
- žánre do ktorých spadá, predpokladáme že film môže mať viac žánrov (napr. "kriminálka a thriller" alebo "romantika, komédia a rodinný")
- dĺžku filmu
- vaše hodnotenie kvality filmu na stupnici od nula do desať.

Chceli by ste, aby váš program vedel:

- vložiť nové DVD
- vymazať DVD (napríklad sa poškodilo alebo stratilo)
- vypísať všetky filmy vo vašej zbierke
- vypísať tie filmy, ktoré zodpovedajú danému žánru (napr. komédie)
- vypísať tie filmy, ktoré sa dajú pozrieť do nejakého času (napr. < 90 minút)
- vypísať všetkých filmy, kde hral daný herec
- vypísať filmy, ktoré sú podľa vášho hodnotenia na stupnici od 7 do 10.

Všimnite si dobre toto zadanie. Ako snáď pri všetkých programoch aj pri tomto si potrebujeme identifikovať dve základné množiny požiadaviek:

- s **akými dátami** bude náš program pracovať
- **aké služby** má poskytovať resp. **akú funkcionalitu** má mať.

Tieto dve množiny požiadaviek sú úplne kľúčové. Pokiaľ si ich programátor neuvedomí, nemal by začať písať ani riadok kódu. Objektovo orientované programovanie zachováva tento princíp rozdelenia na dáta a funkcionalitu nie len pre celý program, ale aj pre jeho základné "súčiastky" - triedy a objekty.

V prvej polovici tejto prednášky sa budeme zaoberať prvou množinou požiadaviek - **dátami**

Trieda ako obal pre viac premenných

Keď máme ujasnenú prvú množinu požiadaviek, teda s akými dátami bude program pracovať, nasleduje prirodzene druhá otázka: *"V akých štruktúrach to budem uchovávať?"* Jednotlivé časti informácií by ste už asi vedeli uložiť vo vhodných premenných.

- Názov filmu bude zrejme **String**,
- meno jedného herca bude tiež typu **String**, keďže tých hercov je v jednom filme viac, asi by ste ich ukladali do poľa **String**ov,
- jeden žáner budeme reprezentovať tiež ako **String**, a keďže tých žánrov v jednom filme môže byť viac, asi by ste ich tiež ukladali do poľa **String**ov,
- dĺžku filmu uložíme asi do premennej typu **int**,
- a hodnotenie môže byť reprezentované napríklad premennou typu **double**.

Dvdčkám by sme vedeli vymyslieť aj ďalšie vlastnosti ako tvar, farba, hmotnosť, cena a podobne. Tieto vlastnosti však nie je potrebné uvažovať, keďže s týmito informáciami podľa zadania zjavne nepotrebujeme pracovať. Zapamätajme si, že uchovávame iba užitočné vlastnosti, teda také, ktoré potrebujeme na zabezpečenie požadovanej funkcionality.

Keďže ideme pracovať s potenciálne veľkým počtom DVD nosičov, je vhodné, aby sme vedeli nejako povedať, že jedna sada týchto premenných patrí jednému DVD a iná sada inému DVD. Chceme teda nejako reprezentovať to, že jedno DVD si bude o sebe pamätať názov filmu, hercov, žánre, dĺžku filmu a hodnotenie.

S podobnou situáciou sme sa už stretli. Keď sme chceli aby si kresliaca plocha pamätala veľa korytnačiek, rozšírili sme triedu `WinPane` a zadefinovali si v tejto novej triede inštančnú premennú referencujúcu pole korytnačiek. Keď sme chceli, aby si plocha pamätala dvojrozmerné pole pre piškôrky, opäť sme nejako rozšírili `WinPane`.

V prípade DVD je neprirodzené rozširovať napríklad `WinPane` na DVDPlochu alebo `Turtle` na DVDKorytnačku, ktorá by predstavovala jedno DVD, lebo v prípade DVD nepotrebujeme pôvodné vlastnosti plochy či korytnačky.

V jave, keď chceme nejakú triedu vybudovať úplne od začiatku vytvárame rozšírenie triedy Object, ktorá je najokliešteneišou triedou v jave. Napríklad aj pôvodná trieda `Turtle` a `WinPane` boli vytvorené rozšírením triedy Object, pretože všetky triedy sú vytvárané buď rozšírením triedy Object alebo rozšírením tried, ktorých pôvod má tiež korene v triede Object.

Vytvorme si triedu `Dvd`, ktorá obaľuje všetky premenné, ktoré potrebujeme na uchovanie informácií o jednom DVD.

```
public class Dvd extends Object {
  private String   nazovFilmu;
  private String[] herci;
  private String[] zanre;
  private int      dlzkaFilmu;
  private double   hodnotenie;
}
```

Ak vytvárame triedu ako rozšírenie triedy Object, nemusíme to expliticne písať. Nasledujúci zápis teda vyjadruje to isté ako predchádzajúci.

```
public class Dvd {
  private String   nazovFilmu;
  private String[] herci;
  private String[] zanre;
  private int      dlzkaFilmu;
  private double   hodnotenie;
}
```

Už vieme že tieto premenné, ktoré sú definované v triede, ale nie v jej metódach, sa nazývajú **inštančné premenné**.

Keď už máme vytvorenú triedu `Dvd` môžeme si vytvárať premenné typu `Dvd` podobne, ako sme si vytvárali nové korytnačky:

```
public class SkusanieDvd {
  public static void main(String[] args) {
    Dvd matrix = new Dvd();
    Dvd shawshank = new Dvd();
    ...
  }
}
```

Priamy prístup k inštančným premenným objektov

Teraz by sme potrebovali o každom z týchto filmov uložiť informácie, ktoré si o filmoch chceme ukladať - teda názov filmu, hercov, žánre, dĺžku filmu a hodnotenie. Chceme teda **naplniť inštančné premenné** vytvorených objektov typu `Dvd`.

Jeden zo spôsobov prístupu k inštančným premenným, a zároveň aj najmenej odporúčaný, je priamy prístup *z vonku* cez premennú objektu. Na to aby sme to umožnili, musíme im vymazať príznak **private**.

Budeme používať "bodkovú" notáciu. Podobne, ako keď voláme metódy na objektoch, môžeme za istých okolností (určite nesmú byť **private**) pristupovať cez bodku aj k inštančným premenným objektov.

```
public class SkusanieDvd {
  public static void main(String[] args) {
    Dvd matrix = new Dvd();
    ...
    matrix.nazovFilmu = "The Matrix";
    matrix.herci = new String[3]; // inštančná premenná herci mala zatiaľ v sebe null
    matrix.herci[0] = "Keanu Reeves";
    matrix.herci[1] = "Laurence Fishburne";
    matrix.herci[2] = "Carrie-Anne Moss";
    matrix.zanre = new String[2]; // inštančná premenná zanre mala zatiaľ v sebe null
    matrix.zanre[0] = "Akčný";
    matrix.zanre[1] = "Sci-Fi";
    matrix.dlzkaFilmu = 136;
    matrix.hodnotenie = 8.7;
    ...
    System.out.println("Dĺžka filmu " + matrix.nazovFilmu + " je " + matrix.dlzkaFilmu+
" minút");
    ...
  }
}
```

```
Dĺžka filmu The Matrix je 136 minút
```

Použitie getterov a setterov

Predchádzajúci spôsob nie je odporúčaný, lebo porušuje princíp ***zapúzdrenosti***, v ktorom sa objekty autonómne starajú o hodnoty svojich vlastných inštančných premenných. Pri priamom prístupe si objekty nevedia samy ochrániť hodnoty svojich inštančných premenných, čo môže ohroziť očakávanú funkcionalitu ďalších metód.

Povedzme, že naša trieda `Dvd`, by mala metódu, ktorá pri výpise informácií o filme bude kresliť graficky ohodnotenie filmu napríklad počtom hviezdičiek. Keďže vieme, že hodnotenia sú v škále 0 až 10, prispôsobíme tomu veľkosť hviezdičiek a ich umiestnenie na ploche. A teraz si predstavte, že pri pridávaní nového filmu sa používateľ vášho programu preklepne a zadá namiesto desiatich sto hviezdičiek, alebo nejaký záporný počet.

Veľmi elegantne sa dá takéto prípady ošetriť tak, že hodnotu budeme nastavovať pomocou nastavovacej medódy - **setter-a** (z angličtiny *setter* = nastavovač, ale slovenský ekvivalent nikto nepoužíva). V tejto metóde si môžeme ošetriť správnosť nastavovanej hodnoty.

```
public class Dvd {
  String   nazovFilmu;
  String[] herci;
  String[] zanre;
```

```
  int      dlzkaFilmu;
  double   hodnotenie;
  ...
  void setHodnotenie(double hodnotenie) {
    if (hodnotenie < 0 || hodnotenie > 10) {
      System.err.println("Hodnotenie " + hodnotenie + " je mimo povolený rozsah
[0,10]");
    }
    else {
      this.hodnotenie = hodnotenie;
    }
  }
  ...
}
```

Potom keď napíšeme nasledujúci príkaz, `hodnotenie` nám nedovolí zmeniť:

```
matrix.setHodnotenie(20.0); // hodnotenie sa nezmení
```

```
Hodnotenie 20.0 je mimo povolený rozsah [0,10]
```

Ochrana však v tomto prípade nie je dokonalá, lebo stále je možné meniť hodnotu priamo. Aby sme tomu zabránili, môžeme premennú `hodnotenie` nastaviť ako *súkromnú* napísaním slovíčka **private** pred jej deklaráciu.

```
public class Dvd {
  ...
  private double   hodnotenie;
  ...
}
```

Teraz keby sme chceli zmeniť hodnotu z vonku priamo, nedovolilo by nám to. Eclipse nám nasledujúci riadok podčiarkne červenou čiarou s chybou "The field Dvd.hodnotenie is not visible"

```
matrix.hodnotenie = 5.0;
```

Súkromné premenné môžu vidieť iba metódy v danej triede. Po nastavení hodnotenia, ako súkromnej premennej, nám teda nedovolí ani jej čítanie z vonku:

```
System.out.println("Hodnotenie je: " + matrix.hodnotenie); //Eclipse vypisuje chybu "The
field Dvd.hodnotenie is not visible"
```

Na čítanie použijeme teda ďalšiu metódu vo vnútri triedy a to takzvaný **getter** (z angličtiny "getter" = odovzdávač, opäť slovenský ekvivalent nik nepoužíva).

```
public class Dvd {
  ...
  private double   hodnotenie;
  ...
  double getHodnotenie() {
    return hodnotenie;
  }
}
```

Zavolanie tohto gettera je potom nasledovné:

```
System.out.println("Hodnotenie je: " + matrix.getHodnotenie());
```

```
Hodnotenie je 8.7
```

Systém getterov a setterov je veľmi často využívaný. Aby ste sa nebodaj neupísali k smrti, keď máte veľa inštančných premenných, Eclipse umožňuje gettery a settery vygenerovať cez menu *Source -> Generate Getters and Setters*, kde si môžete špecifikovať, pre ktoré inštančné premenné sa majú gettery a settery vygenerovať.

Našej triede `Dvd` budeme odteraz chrániť jej inštančné premenné:

```
public class Dvd {
  private String   nazovFilmu;
  private String[] herci;
  private String[] zanre;
  private int      dlzkaFilmu;
  private double   hodnotenie;

  String getNazovFilmu() {
    return nazovFilmu;
  }

  void setNazovFilmu(String nazovFilmu) {
    this.nazovFilmu = nazovFilmu;
  }

  String[] getHerci() {
    return herci;
  }

  void setHerci(String[] herci) {
    this.herci = herci;
  }

  String[] getZanre() {
    return zanre;
  }

  void setZanre(String[] zanre) {
    this.zanre = zanre;
  }

  int getDlzkaFilmu() {
    return dlzkaFilmu;
  }

  void setDlzkaFilmu(int dlzkaFilmu) {
    if (dlzkaFilmu <= 0) {
      System.err.println("Dĺžka filmu musí byť kladné číslo");
    }
    else {
      this.dlzkaFilmu = dlzkaFilmu;
    }
  }

  double getHodnotenie() {
    return hodnotenie;
  }

  void setHodnotenie(double hodnotenie) {
    if (hodnotenie < 0 || hodnotenie > 10) {
      System.err.println("Hodnotenie " + hodnotenie + " je mimo povolený rozsah
[0,10]");
    }
    else {
      this.hodnotenie = hodnotenie;
    }
  }
}
```

Naplnenie nového objektu typu `Dvd` a jeho používanie môže teraz prebiehať nasledovne:

```
public class SkusanieDvd {
  public static void main(String[] args) {
    ...
    Dvd shawshank = new Dvd();
    shawshank.setNazovFilmu("The Shawshank Redemption");
    shawshank.setHerci(new String[]{"Tim Robbins","Morgan Freeman"});
    shawshank.setZanre(new String[]{"Dráma"})
    shawshank.setDlzkaFilmu(142);
    shawshank.setHodnotenie(9.2);
    ...
    System.out.println("Dĺžka filmu " + shawshank.getNazovFilmu() + " je " +
shawshank.getDlzkaFilmu());
    ...
  }
}
```

```
Dĺžka filmu The Shawshank Redemption je 142
```

Konštruktory

Napĺňanie inštančných premenných priamo pri vytváraní objektu nám umožňuje konštruktor. Konštruktor je špeciálny typ metódy, ktorá sa volá iba pri vytváraní (konštruovaní) nového objektu. Keď už objekt existuje, konštruktor preňho už nevoláme. Výsledkom zavolania konštruktora je nový objekt, ktorý môžeme typicky priradiť do nejakej premennej príslušného typu.

Volanie konštruktora sa nachádza za kľúčovým slovíčkom **new**. Konštruktor môže mať žiaden alebo viac parametrov.

```
Dvd matrix = new Dvd(); // voláme konštruktor bez parametrov
File adresar = new File("C:/Windows"); // voláme konštruktor s jedným parametrom
File subor = new File(adresar, "system.ini"); // voláme konštruktor s dvoma parametrami
Scanner sc = new Scanner(subor);// voláme konštruktor s jedným parametrom
```

Každá trieda má aspoň jeden konštruktor. Môže ich mať aj viac, ale musia sa líšiť počtom parametrov alebo typmi parametrov. Ak sa metódy alebo konštruktory volajú rovnako a líšia sa iba parametrami hovoríme o **preťažovaní**. V predchádzajúcom príklade sme volali až dva rôzne konštruktory triedy File, ktoré sa odlišovali počtom parametrov.

Ak v triede nie je napísaný žiaden koštruktor, použije sa skrytý, takzvaný **implicitný konštruktor**. Ide o koštruktor, ktorý nemá žiadne vstupné parametre, ani žiadne príkazy v tele. Prázdny konštruktor sa v prípade, že neexistujú iné konštruktory triedy, písať nemusí. POZOR! Ak existuje v triede nejaký konštruktor s parametrami, implicitný konštruktor sa automaticky nedoplní a ak ho chceme používať musíme si ho dopísať sami. Ak by sme chceli napísať prázdny konštruktor, vyzeral by takto:

```
public class Dvd {
  ...
  public Dvd() {
    //prázdny konštruktor
  }
  ...
}
```

Konštruktor je špeciálny typ metódy. Od normálnych metód sa líši nasledovne:

- meno konštruktora musí byť rovnaké ako meno triedy - v našom prípade `Dvd`
- nepíšeme mu návratový typ, teda hlavička konštruktora nezačína slovom **void** ani nejakým konkrétnym typom
- telo konštruktora teda neobsahuje žiadne vrátenie hodnoty cez **return**.

Prvou úlohou každého, aj prázdneho, konštruktora je vytvoriť objekt, jeho inštančné premenné a naplniť inštančné premenné preddefinovanými hodnotami.

Preddefinovaná hodnota pre triedové premenné je `null`. Premenné primitívnych typov `null` obsahovať nemôžu, takže majú určené nasledovné hodnoty.

- Všetky číselné primitívne typy majú preddefinovanú nulu
- `boolean` premenné majú preddefinovaný `false`
- `char` má preddefinovanú hodnotu '\u0000', ktorá je neviditeľným (NUL) znakom, ktorý sa v praxi nevyužíva.

V našom prípade, keď vytvoríme nový objekt triedy `Dvd` cez prázdny konštruktor, máme v premenných `nazovFilmu`, `herci` a `zanre` hodnotu **null**, v premennej `dlzkaFilmu` hodnotu 0 a v premennej `hodnotenie` hodnotu 0.0

Po vytvorení objektu môžeme konštruktor použiť na vhodnejšie nastavenie inštančných premenných. Využijeme na to niektorý konštruktor s parametrami.

```
public class Dvd {
  ...
  public Dvd(String nazovFilmu, String[] herci, String[] zanre, int dlzkaFilmu, double
hodnotenie) { // tento konštruktor nám nastaví všetky inštančné premenné
    this.nazovFilmu = nazovFilmu;
    this.herci = herci;
    this.zanre = zanre;
    this.dlzkaFilmu = dlzkaFilmu;
    this.hodnotenie = hodnotenie;
  }

  public Dvd(String nazovFilmu, int dlzkaFilmu, double hodnotenie) { //nastavíme iba 3
inštančné premenné a zvyšok necháme na settery
    this.nazovFilmu = nazovFilmu;
    this.dlzkaFilmu = dlzkaFilmu;
    this.hodnotenie = hodnotenie;
  }

  public Dvd() {
    // doplníme prázdny konštruktor, aby sme mohli naďalej volať aj konštruktor bez
parametrov aj keď sme vytvorili konštruktory s parametrami
  }
  ...
}
```

Nový objekt triedy DVD potom vytvoríme volaním ľubovoľného (preťaženého) konštruktora.

```
public class SkusanieDvd {
  public static void main(String[] args) {
    ...
    String[] herci = new String[3]; // vyrobíme si pomocnú lokálnu premennú
    herci[0] = "Eva Vejmělková";
    herci[1] = "Jiří Bábek";
    herci[2] = "Robo Grigorov";
    String[] zanre = new String[3]; // vyrobíme si pomocnú lokálnu premennú
    zanre[0] = "Dráma";
    zanre[1] = "Romantika";
```

```
    zanre[2] = "Muzikál";
    Dvd fontana = new Dvd("Fontána pre Zuzanu",herci,zanre,81,6.3); //vytvoríme nový
objekt s pomocou konštruktora s piatimi parametrami

    Dvd pacho = new Dvd("Pacho, hybský zbojník",91,8.5); //ak použijeme konštruktor s
troma parametrami, potrebujeme ešte nastaviť hercov a žánre
    herci = new String[2]; // vytvoríme nové pole do pomocnej premennej herci
    herci[0] = "Jozef Kroner";
    herci[1] = "Marián Labuda";
    pacho.setHerci(herci);
    zanre = new String[1]; // vytvoríme nové pole do pomocnej premennej zanre
    zanre[0] = "Komédia";
    pacho.setZanre(zanre);
    ...
    System.out.println("Herci vo filme " + fontana.getNazovFilmu() + " sú:");
    for (int i = 0; i < fontana.getHerci().length; i++) {
      System.out.println(fontana.getHerci()[i]);
    }
    ...
  }
}
```

```
Herci vo filme Fontána pre Zuzanu sú:
Eva Vejmělková
Jiří Bábek
Robo Grigorov
```

Ak by ste potrebovali, tak konštruktor bez parametrov nemusí byť vždy prázdny. Nasledujúci konštruktor nastaví automaticky hodnotenie 5.0 pre každý objekt triedy `Dvd`, ktorý bol vytvorený cez konštruktor bez parametrov.

POZOR! Do triedy môžeme napísať iba jeden konštruktor s rovnakým počtom, typmi a poradím typov parametrov. Takže nemôžete mať v triede aj prázdny konštruktor bez parametrov aj konštruktor bez parametrov, ktorý má v sebe nejaké príkazy.

```
public class Dvd {
  ...
  public Dvd() { //prednastavujeme filmom hodnotenie 5.0
    this.hodnotenie = 5.0;
  }
  ...
  public Dvd() { //tento konštruktor tu už byť nesmie
  }
}
```

Aj konštruktor sa bežne používa na ochranu hodnôt inštančných premenných. Prepíšeme si konštruktor na taký, ktorý robí rovnaké kontroly, ako sme písali v setteroch. V prípade, že vstupné hodnoty nebudú spĺňať požadované podmienky, nastavíme im nejaké preddefinované hodnoty.

```
public class Dvd {
  ...
  public Dvd(String nazovFilmu, String[] herci, String[] zanre, int dlzkaFilmu, double
hodnotenie) { // tento konštruktor nám nastaví všetky inštančné premenné
    this.nazovFilmu = nazovFilmu;
    this.herci = herci;
    this.zanre = zanre;

    if (dlzkaFilmu <= 0) {
      this.dlzkaFilmu = 0; //preddefinovaná dĺžka bude nula
    }
    else {
```

```
      this.dlzkaFilmu = dlzkaFilmu;
    }

    if (hodnotenie < 0 || hodnotenie > 10) {
      this.hodnotenie = 5.0; //preddefinované hodnotenie bude 5.0
    }
    else {
      this.hodnotenie = hodnotenie;
    }
  }
...
}
```

Keďže aj konštruktor je riadny kus kódu, ktorý sa nám písať nechce, Eclipse prichádza s generátorom koštruktora. Koštruktor vygenerujete cez menu *Source -> Generate Constructor using fields*.

Hierarchia konštruktorov

V kapitolke o koštruktoroch sme povedali, že prvou úlohou každého konštruktora je vytvoriť objekt, jeho inštančné premenné a naplniť inštančné premenné prednastavenými hodnotami. Vytváranie objektu reálne vykonáva konštruktor triedy Object a inštančné premenné sa doplňujú postupne volaním viacerých konštruktorov. Ako to vlastne funguje?

Musíme si uvedomiť, že tým, že naša trieda rozširuje nejakú inú triedu, tak okrem nových vecí musí vedieť aj všetko to, čo pôvodná trieda. Z toho dôvodu sa ako prvá vec pri vytváraní nového objektu vyrobí objekt, ktorý vie všetko to, čo by mu umožnila pôvodná trieda a potom naňho nabalíme schopnosti, ktoré má od našej triedy.

Keďže všetky triedy pôvodne pochádzajú z triedy Object, tak každý nový objekt najprv vznikne ako inštancia, ktorá má všetky schopnosti, ktoré mu poskytuje trieda Object a až potom sa na tieto schopnosti nabaľujú schopnosti (inštančné premenné a metódy), ktoré poskytujú rozširujúce triedy.

Prvá vec, čo urobí každý konštruktor je, že zavolá konštruktor rodičovskej triedy (t.j. z triedy z ktorej sa rozširuje). Tento rodičovský konštruktor zavolá konštruktor svojho rodiča a tak ďalej až kým sa nedostaneme k zavolaniu konštruktora triedy Object.

Ak sa na začiatku konštruktora neuvedie volanie rodičovského konštruktora, toto volanie sa uskutočňuje automaticky. Automatické volanie používa rodičovský konštruktor bez parametrov.

Pozrime sa na to, čo sa deje v kóde. Rodičovská trieda sa v jave označuje slovom **super**. Keď si vygenerujete konštruktor cez generátor konštruktora v Eclipse, ako prvý príkaz v konštruktore je `super();`. Ak je tento príkaz uvedený, MUSÍ byť uvedený ako prvý príkaz v konštruktore. To je volanie rodičovského konštruktora - neznamená to, že rodičovská trieda sa volá `super`.

Ak nechcete aby sa volal konštruktor rodičovskej triedy bez parametrov, môžete zavolať ľubovoľný jeden konštruktor s parametrami z rodičovskej triedy.

Je potrebné dodať, že niekedy sa konštruktor bez parametrov volať nedá a musíme volať rodičovský konštruktor s parametrami cez **super(...)**. Je to vtedy, keď rodičovská trieda takýto konštruktor nemá a zároveň má nejaký konštruktor s parametrami t.j. implicitný konštruktor sa jej automaticky nevygeneroval.

Použitie si ukážeme na triede `DvdNaPredaj`, ktorá rozširuje našu triedu `Dvd` pridaním ceny.

```
public class DvdNaPredaj extends Dvd {
  double cena;

  public DvdNaPredaj(String nazovFilmu, String[] herci, String[] zanre, int dlzkaFilmu,
double hodnotenie, double cena) { // tento konštruktor nám nastaví všetky inštančné
premenné
    super(nazovFilmu, herci, zanre, dlzkaFilmu, hodnotenie); // zavolanie konštruktora
triedy Dvd s 5 parametrami
    this.cena = cena;
  }

  public DvdNaPredaj(String nazovFilmu, double cena) { //nastavíme iba 2 inštančné
premenné
    super(); //použijeme rodičovský konštruktor bez parametrov- tento riadok môžeme
vymazať a bude to fungovať rovnako
    this.nazovFilmu = nazovFilmu;
    this.cena = cena;
  }

  public DvdNaPredaj() { // v tomto konštruktore sa iba zavolá konštruktor triedy Dvd
bez parametrov
  }
  ...
}
```

Zosumarizujme, čo sa reálne deje pri volaní konštruktora:

1. zavolá sa rodičovský konštruktor, ktorý zabezpečí vznik objektu, ktorý má schopnosti získané od rodičovskej triedy - ak nie je uvedený, použije sa konštruktor bez parametrov.
2. vytvoria sa inštančné premenné špecifikované v aktuálnej triede a naplnia sa prednastavenými hodnotami (premenné tried majú null, čísla sú nuly, boolean sú false a char sú '\u0000') a pridajú sa do objektu
3. spúšťajú sa postupne príkazy v tele konštruktora.
4. keď sa vykonajú všetky príkazy konštruktora, objekt je vytvorený a vráti sa referencia na tento novovytvorený objekt - obvykle sa táto referencia priradí do premennej, ktorá je rovnakého typu ako trieda, ktorej konštruktor sa vykonával.

Vytvárame funkčné triedy

Predchádzajúce kapitoly nám rôznorodým spôsobom pomohli vytvoriť objekty typu `Dvd` a nastaviť im hodnoty inštančných premenných. Po vytvorení týchto objektov sme si uložili referencie na ne do premenných typu `Dvd`.

```
public class SkusanieDvd {
  public static void main(String[] args) {
    ...
   Dvd matrix = ...
   ...
   Dvd shawshank = ...
   ...
   Dvd fontana = ...
   ...
   Dvd pacho = ...
   ...
  }
}
```

Mať každé DVD v jednej premennej je veľmi nepraktické, preto by sme potrebovali dať tieto objekty radšej do poľa Dvdčiek.

```
public class SkusanieDvd {
  public static void main(String[] args) {
    ...
   Dvd[] filmy = new Dvd[4];
   filmy[0] = matrix;
   filmy[1] = shawshank;
   filmy[2] = fontana;
   filmy[3] = pacho;
   ...
  }
}
```

Teraz by sme už vedeli napríklad jednoduchým cyklom vypísať všetky filmy v našej zbierke. Pokiaľ by nám išlo iba o uloženie dát tak sme hotoví. Zadanie sa však skladá z dvoch množín požiadaviek:

- s **akými dátami** bude náš program pracovať
- **aké služby** má poskytovať resp. **akú funkcionalitu** má mať.

Pozrime sa teraz na druhú množinu požiadaviek - funkcionalitu. V našom zadaní sú nasledovné požiadavky na funkcionalitu.

- vložiť nové DVD
- vymazať DVD (napríklad sa poškodilo alebo stratilo)
- vypísať všetky filmy vo vašej zbierke
- vypísať tie filmy, ktoré zodpovedajú danému žánru (napr. komédie)
- vypísať tie filmy, ktoré sa dajú pozrieť do nejakého času (napr. < 90 minút)
- vypísať všetkých filmy, kde hral daný herec
- vypísať filmy, ktoré sú podľa vášho hodnotenia na stupnici od 7 do 10.

Ako pretaviť tieto požiadavky do kódu? **Z každej požiadavky na funkcionalitu programu sa stane metóda**, ktorú zavoláme, keď budeme chcieť vykonať niektorú z požadovaných akcií.

V Jave musí každá metóda *bývať* v nejakej triede. Už sme sa naučili, že je dobrým zvykom dávať **main** metódu do samostatnej triedy (nazvanej napríklad `Spustac`), kde sa nič iné ako metóda **main** nenachádza. Takže do tejto triedy ich nedáme.

Do triedy `Dvd` je ich tiež nevhodné dávať z toho dôvodu, že ľubovoľné Dvd by malo schopnosti spravovať všetky ostatné DVD-čka, vznikla by anarchia. Naviac `Dvd` nemá potrebné dáta z iných DVD-čiek a museli by sme mu ich stále dodať. Spomínané požiadavky funkcionality nášho programu nie sú požiadavky, ktoré by malo spĺňať každé DVD-čko, ale mal by ich spĺňať akýsi správca či inteligentný zoznam DVD-čiek.

Zapúzdrenosť

Zapúzdrenosť je základné pravidlo OOP, ktoré by sa malo dodržiavať. Zapúzdranosť nám hovorí, že všetky dôležité dáta by mali mať svojho *zodpovedného správcu*, ktorý má plnú kontrolu nad obsahom týchto dát.

Inštančné premenné majú svojho *správcu* - triedu, ktorá jediná má právo pristupovať k svojim premenným pomocou svojich metód. Lepšie povedané objekty, t.j. inštancie tried, sú zodpovedné za obsah svojich premenných a iba oni by mali mať v dobrom návrhu programu plnú kontrolu nad zmenami obsahu svojich inštančných premnných.

V našom príklade predstavuje dôležité dáta pole DVD-čiek. Vyrobíme teda tomuto poľu jeho správcu, triedu `ZoznamDvd`, ktorá bude zodpovedná za všetky zmeny v tomto poli a tiež bude poskytovať vyhľadávacie služby nad týmto poľom.

Keďže všetky metódy zo zadania pracujú hlavne s poľom DVD-čiek, všetky umiestnime do triedy, ktorá toto pole spravuje. Na začiatok si napíšeme prvý nástrel hlavičiek týchto metód:

```
public class ZoznamDvd {
  private Dvd[] filmy;

  public ZoznamDvd() {
    filmy = new Dvd[0]; //na začiatku máme nula filmov, ale máme istotu, že v premnnej
filmy nebude null
  }

  public void vlozNoveDvd(Dvd dvd) {
  }

  public void vymazDvd(Dvd dvd) {
  }

  public void vypisVsetko() {
  }

  public void vypisPodlaZanru(String zaner) {
  }

  public void vypisPodlaCasu(int maximalnyCas) {
  }

  public void vypisPodlaHerca(String menoHerca) {
  }

  public void vypisPodlaHodnotenia(double odHodnota, double doHodnota) {
  }
}
```

Poďme si doplniť telá našich metód. Prvou metódou je `vlozNoveDvd`. Budeme klasicky nafukovať pole tak, že vytvoríme väčšie pole do ktorého pôvodné skopírujeme a pridáme nový prvok.

```
public class ZoznamDvd {
  ...
  public void vlozNoveDvd(Dvd dvd) {
    Dvd[] novePole = new Dvd[filmy.length+1];
    System.arraycopy(filmy,0,novePole,0,filmy.length);
    filmy = novePole;
    filmy[filmy.length-1] = dvd;
  }
  ...
}
```

Metódae `vymazDvd` má opačný postup, teda zmenšuje pole.

```
public class ZoznamDvd {
  ...
  public void vymazDvd(Dvd dvd) {
    int indexDvd = -1;
    for (int i = 0; i < filmy.length; i++) {
      if (dvd.equals(filmy[i])) {
        indexDvd = i;
        break;
      }
    }
    if (indexDvd >= 0) { // ak sme našli dané DVD-čko
      Dvd[] novePole = new Dvd[filmy.length-1];
      System.arraycopy(filmy, 0, novePole, 0, indexDvd); //skopírujeme všetky prvky
naľavo od indexDvd
      System.arraycopy(filmy, indexDvd+1, novePole, indexDvd, novePole.length -
indexDvd); //skopírujeme všetky prvky napravo od indexDvd
      filmy = novePole;
    }
  }
  ...
}
```

Na tejto metóde je blbé to, že má na vstupe referenciu na inštanciu `Dvd`, ktorá sa má vymazať. My však od používateľa nášho programu dostaneme skôr názov filmu. Vyrobme si teda inú verziu tejto metódy, ktorá má na vstupe názov filmu. Budeme mať teda dve metódy s rovnakým názvom t.j. preťažené metódy.

Preťažené metódy

Preťažené metódy sa od seba musia líšiť

- počtom parametrov,
- alebo ak majú rovnaký počet parametrov, musí sa líšiť typ aspoň jedného z parametrov.

Pozor! Neplatí, že stačí že sa líšia návratové typy metód. Musia sa líšiť typy vstupných premenných. Napríklad **nasledovné sa nedá skompilovať**, lebo nie je možné na základe typov vstupných parametrov rozhodnúť, ktorá z týchto metód sa má zavolať:

```
...
public int vypocet(int vstup1, double vstup2){
  ...
}

public double vypocet (int prva, double druha) {
  ...
}
...
```

Preťažené metódy sa môžu líšiť aj poradím parametrov ak sa na nejakej pozícii líšia typy, v praxi sa to však neodporúča.

Vráťme sa k nášmu príkladu korektne preťaženej metódy `vymazDvd`. Na skracovanie poľa využijeme už hotovú metódu s rovnakým názvom, ale iným typom parametra.

```
public class ZoznamDvd {
  ...
  public void vymazDvd(String nazovFilmu) {
    for (int i = 0; i < filmy.length; i++) {
      if (nazovFilmu.equals(filmy[i].getNazovFilmu())) {
        this.vymazDvd(filmy[i]);
      }
    }
  }
  ...
}
```

Nasleduje plejáda metód ktoré majú vypisovať filmy, ktoré spĺňajú nejakú podmienku. Povedzme, že pri výpise chceme o každom DVD-čku vypísať nielen názov filmu, ale aj ostatné parametre: dĺžku, žánre, hercov aj hodnotenie. Opäť si spomenieme na princíp zapúzdrenosti. O premenné `nazovFilmu`, `dlzkaFilmu` a tak ďalej sa starajú objekty triedy `Dvd`. Naučíme teda triedu `Dvd` nech nám vráti reťazec, ktorý budeme chcieť vypísať pri výpise DVD-čiek.

```
public class Dvd {
...
  String retazecPreVypis() {
    String vystup = nazovFilmu + "\n";
    vystup = vystup + dlzkaFilmu + " minút\n";
    vystup = vystup + "žánre: ";
    for (int i = 0; i < zanre.length; i++) {
      vystup = vystup + zanre[i] + ", ";
    }
    vystup = vystup + "\n" + "herci: ";
    for (int i = 0; i < herci.length; i++) {
      vystup = vystup + herci[i] + ", ";
    }
    vystup = vystup + "hodnotenie: " + hodnotenie + "\n";
    return vystup;
  }
...
}
```

poznámka pre optimalizáciu: namiesto zreťazovania je vhodnejšie používať triedu StringBuilder.

Keď už sme triedu `Dvd` naučili vytvárať výpis o sebe, metóda na výpis všetkých DVD-čiek je už malina.

```
public class ZoznamDvd {
  ...
  public void vypisVsetko() {
    for (int i=0; i < filmy.length; i++) {
      System.out.println(filmy[i].retazecPreVypis());
    }
  }
  ...
}
```

V metóde `vypisPodlaZanru` by sme o každom DVD-čku potrebovali zistiť, či daný žáner obsahuje. Túto robotu opäť necháme na triedu `Dvd` keďže ide o jej pole.

```
public class Dvd {
...
  boolean mamZaner(String zaner) {
    for (int i = 0; i < zanre.length; i++) {
```

```
      if (zaner.equals(zanre[i]))
        return true;
      }
    return false;
  }
...
}
```

V triede `ZoznamDvd` to potom bude vyzerať nasledovne:

```
public class ZoznamDvd {
  ...
  public void vypisPodlaZanru(String zaner) {
    for (int i=0; i < filmy.length; i++) {
      if (filmy[i].mamZaner(zaner))
        System.out.println(filmy[i].retazecPreVypis());
    }
  }
  ...
}
```

Posledné tri metódy zo zadania sú už iba ľahkým cvičením využívajúcim naučené veci, ktoré určite zvládnete aj sami.

Dedičnosť, t.j. rozširovanie tried

Stalo sa, čo ste nepredpokladali. Vaša rodina bola nadšená vašim zoznamom DVD-čiek a chce od Vás, aby ste evidovali aj všetky filmy na videopáskach a v počítači. Samozrejme centrálne v jednom programe. Navyše im už nestačia informácie ako sa film volá, kto v ňom hrá, aké má žánre, dĺžku a hodnotenie. Chcú vedieť, kde film hľadať. Ale ono to je závislé na médiu, kde je film uložený:

- DVD
    - Máme ich očíslované - rodičia sa snažili urobiť poriadok, na každé DVD-čko nalepili čísielko a vedú si ošuntelý a niekoľkokrát kávou poliaty zoznam. Samozrejme, niečo v tom zozname nájsť je horor, lebo filmy sú zotriedené podľa čísla (v akom poradí sa nakupovali) a nie podľa abecedy.
- Pásky
    - tiež majú číslo
    - no keďže ide o nahrávky z TV, tak potrebujeme evidovať aj od ktorej minúty na páske daný film začína
- Súbory v počítačoch
    - treba evidovať, v ktorom počítači je uložený
    - treba evidovať aj úplnú cestu k súboru kde je uložený
    - dôležitá je aj veľkosť súboru

Filmy na všetkých médiách majú nejaké spoločné dáta a funkcionalitu. Film na ľubovoľnom médiu si zachováva pôvodné dáta:

- názov filmu
- hercov, ktorý v ňom hrali
- žánre, do ktorých spadá. Predpokladáme že film môže mať viac žánrov (napr. "kriminálka a thriller" alebo "romantika, komédia a rodinný")
- dĺžku filmu
- vaše hodnotenie kvality filmu na stupnici od nula do desať.

Rovnaké budú aj metódy spoločné pre filmy na všetkých médiách - `mamZaner`, `mamHerca`, `retazecPreVypisFilmu`

Rozdielne dáta sa týkajú identifikácie média, na ktorom sa film nachádza (očíslovanie, identifikácia počítača a cesty) a tiež niektoré doplňujúce údaje (začiatočná minúta, veľkosť súboru).

Rozdielne bude aj chovanie niektorých funkčných schopností (výpis umiestnenia filmu, uloženie do súboru, dodatočné informácie)

Dobrý programátor sa snaží každú metódu písať v celom programe iba na jednom mieste, v jednej jedinej kópii. Dôvod je zrejmý. Programátor pamätá na to, že každá metóda má veľkú šancu na svoju zmenu z rôznych dôvodov (prídu dáta s ktorými sa pred tým nerátalo, zákazník chce zmenu chovania metódy, Y2K,...) a v prípade jej zmeny to robí len na jednom mieste. Výhody sú tieto:

- netreba veľa písať
- neprihodí sa to, že sme niektorú kópiu metódy zabudli prepísať (niekedy sa to veľmi ťažko odhaľuje, lebo program v 99% prípadov funguje dobre)

- pri neskorom odhalení zabudnutia zmeny v niektorej z kópií sa môžu dáta stať nekonzistentné, čo je často spojené s ťažkým a náročným opravovaním, ak je vôbec oprava možná

Z toho vyplýva, že nie je dobré, keby sme našu situáciu s troma druhmi médií pre filmy vyrobili 3 triedy (`FilmNaDvd`, `FilmNaPaske`, `FilmVPocitaci`) a každá z nich by zisťovala napríklad prítomnosť herca vo filme svojou metódou s rovnakým telom. Mohlo by sa nám totiž stať, že zrazu niekto zahlási, že meno herca nestačí, lebo o hercoch chceme uchovávať aj národnosť, rasu, ocenenia a podobne. To by viedlo k tomu, že hercov nebudeme uchovávať ako pole `String`-ov ale ako pole typu `Herec[]`. Nasledovalo by trojnásobné prepisovanie typu inštančnej premennej `herci` a trojnásobné prepisovanie metódy `mamHerca` (a to náš program je ešte malý).

Využijeme **dedičnosť**, ktorú ste doteraz poznali pod názvom **rozširovanie tried**. Vyrobíme si triedu pre premenné a metódy, ktoré sú spoločné pre všetky typy médií. Následne vytvoríme triedy, ktoré rozširujú túto triedu o ďalšie vlastnosti. Táto spoločná trieda sa bude nazývať `Film` a z nej rozšírené (oddedené) triedy sa budú volať `FilmNaDvd`, `FilmNaPaske` a `FilmVPocitaci`. Tieto triedy budú môcť používať všetky metódy rodičovskej (rozširovanej) triedy `Film`. Na podobné správanie sme už zvyknutí pri rozširovaní `Turtle` alebo `WinPane`.

```
public class Film {

        private String nazovFilmu;
        private String[] herci;
        private String[] zanre;
        private int dlzkaFilmu;
        private double hodnotenie;

        public boolean mamHerca(String herec) {
                ...
        }

        public boolean mamZaner(String zaner){
                ...
        }

        public String retazecPreVypis() {
                ...
        }

   //nasledujú gettery a settery
}
```

```
public class FilmNaDvd extends Film {
        private int cisloDvdcka;

        public String dajUmiestnenie() {
                return "DVD číslo "+cisloDvdcka;
        }

...
}
```

```
public class FilmNaPaske extends Film {
        private int cisloPasky;
        private int zaciatok;

        public String dajUmiestnenie() {
                return "Páska číslo "+cisloPasky+ ", "+ zaciatok+". minúta";
        }
```

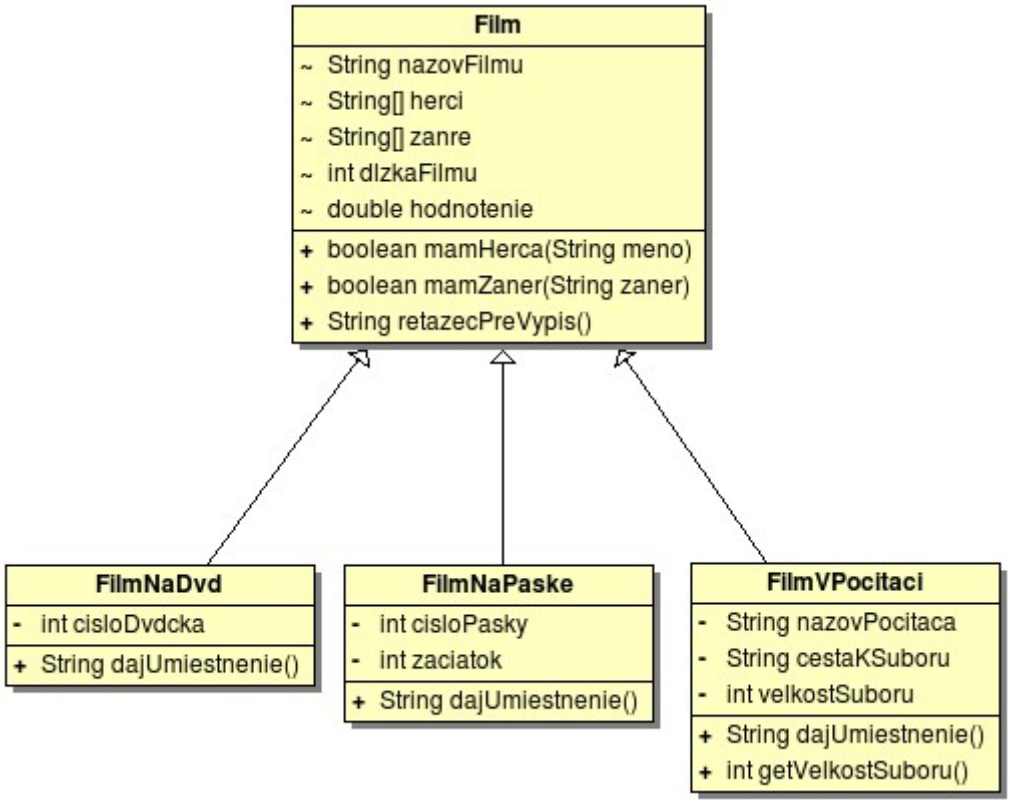
```
...
}
```

```
public class FilmVPocitaci extends Film {
        private String nazovPocitaca;
        private String cestaKSuboru;
        private int    velkostSuboru;

        public String dajUmiestnenie() {
                String vysledok ="Počítač "+nazovPocitaca + "\n";
                return vysledok = vysledok + cestaKSuboru;
        }

        public int getVelkostSuboru(){
                return velkostSuboru;
        }
...
}
```

Pre celkový obraz sa používa takzvaný **triedový diagram**, ktorý znázorňuje, ktorá trieda je rozšírením ktorej (resp. ktorá trieda dedí od ktorej).



Keď máme už tieto triedy naprogramované, môžeme prispúpiť k renovácii zoznamu filmov. Keďže máme médiá troch druhov zrejme by ste očakávali, že vytvoríme tri polia:

```
public class ZoznamFilmov {
        private FilmNaDvd[]     filmyDvd;
        private FilmNaPaske[]   filmyPasky;
        private FilmVPocitaci[] filmyPocitac;
...
}
```

a potom pri výpise, ale aj pri všetkých ostatných metódach budeme používať 3 cykly:

```
public class ZoznamFilmov {
...
```

```
        public void vypisVsetko() {
                for (int i = 0; i < filmyDvd.length; i++) {
                        System.out.println(filmyDvd[i].retazecPreVypis());
                }
                for (int i = 0; i < filmyPasky.length; i++) {
                        System.out.println(filmyPasky[i].retazecPreVypis());
                }
                for (int i = 0; i < filmyPocitac.length; i++) {
                        System.out.println(filmyPocitac[i].retazecPreVypis());
                }
        }
...
}
```

Takýto prístup našťastie nie je potrebný, pretože platí, že **premenná majúca typ nadradenej triedy** (v našom prípade `Film`) **dokáže referencovať aj objekt, ktorý je inštanciou ľubovoľnej triedy, ktorá je jej potomkom** (v našom prípade napr. `FilmNaDvd`). Inými slovami z premennej typu `Film` vieme referencovať objekty typu `Film`, `FilmNaDvd`, `FilmNaPaske` alebo `FilmVPocitaci`:

```
Film matrix = new Film();
Film matrix2 = new FilmNaPaske();
```

Cez premenné `matrix` a `matrix2` však môžeme volať iba tie metódy, ktoré sú definované v triede `Film` alebo jej predkoch. Nasledujúce teda nie je možné:

```
String s = matrix2.dajUmiestnenie();
```

Teda **od premennej typu `Film` môžeme požadovať iba to, čo umožňuje trieda `Film`**. Keďže pri dedičnosti platí, že triedy vedia všetko to, čo zdedili od predkov, tak aj objekty tried `FilmNaDvd`, `FilmNaPaske` alebo `FilmVPocitaci` vedia všetko to, čo by sme mohli požadovať od `Film`-u.

Treba si dať pozor na to, že **opačné pravidlo neplatí**, t.j. z premennej, ktorá je typu potomka (napr. `FilmNaDvd`) nevieme referencovať objekt typu predka (napr. `Film`). Je to logické keď si uvedomíte, že cez premennú typu `FilmNaDvd` očakávame možnosť volania ľubovoľnej metódy triedy `FilmNaDvd`, a to objekt triedy `Film` poskytnúť nevie.

Budeme si teda uchovávať iba jedno pole filmov a môžeme z neho referencovať všetky typy médií. Výpis filmov už teda bude iba cez jedno pole.

```
public class ZoznamFilmov {
        private Film[] filmy;
...
        public void vypisVsetko() {
                for (int i = 0; i < filmy.length; i++) {
                        System.out.println(filmy[i].retazecPreVypis());
                }
        }
...
}
```

Polymorfizmus

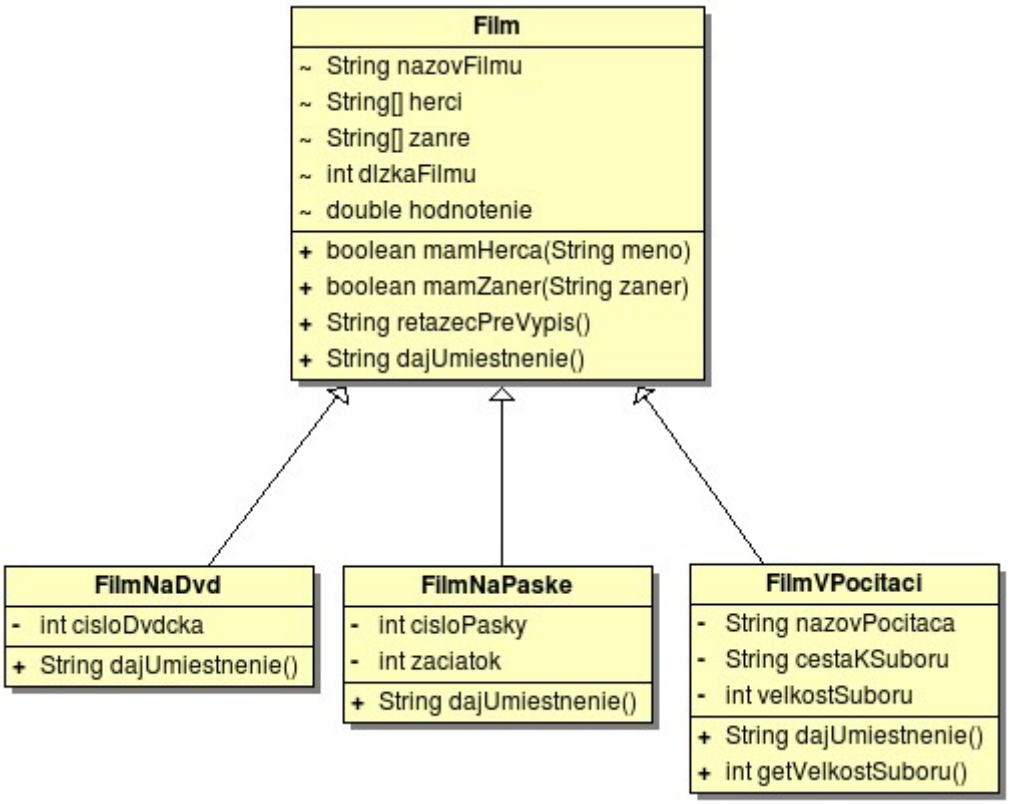
No hej, poviete si, ale ako my teraz budeme volať metódy `dajUmiestnenie()` ? Odpoveďou je **polymorfizmus**, alebo po slovensky viactvarovosť (slovenský ekvivalent nikto nepoužíva). Ide vlastne o dôsledok mechanizmu volania metód na objektoch. Keď cez niektorú premennú referencujúcu daný objekt zavoláme nejakú metódu, postupuje sa nasledovne:

- Ak je v triede tohto objektu takáto metóda definovaná, zavolá sa
- Ak nie, pozrieme sa do jej rodičovskej triedy. Ak sa tam táto metóda nachádza, vykoná sa, inak postupujeme v hierarchii vyššie, až kým na túto metódu nenarazíme a vykonáme ju.

Toto pravidlo platí bez ohľadu na to akého typu bola premenná referencujúca daný objekt. V predchádzajúcej kapitolke sme však povedali, že vieme volať iba metódy, ktoré nám umožní typ premennej. Dopíšeme si teda metódu `dajUmiestnenie()` aj do triedy `Film`. Reálne sa však táto metóda v našom programe nikdy nezavolá, keďže v poli `filmy` nebudeme mať objekty typu `Film`, ale iba objekty jej potomkov. Hovoríme tomu, že metódy v potomkoch **prekrývajú túto metódu**.

```
public class Film {
...
        //túto metódu potomkovia prekryjú
        public String dajUmiestnenie() {
                return "nemá umiestnenie";
        }
...
}
```

Situácia sa pridaním tejto metódy zmenila nasledovne:



Teraz si môžeme v triede `ZoznamFilmov` urobiť akýsi výpis filmov s ich umiestneniami. Každý objekt si zavolá metódu `dajUmiestnenie()` zo svojej triedy.

```
public class ZoznamFilmov {
        private Film[] filmy;
...
        public void vypisFilmySUmiestnenim() {
                for (int i = 0; i < filmy.length; i++) {
                        System.out.println(filmy[i].getNazovFilmu());
                        System.out.println(filmy[i].dajUmiestnenie());
                }
        }
...
}
```

Malý bonus nakoniec: Ak by sme chceli aby aj metóda `retazecPreVypis()` vrátila okrem pôvodných informácií o filme aj informáciu o umiestnení, môžeme ***v triede*** `Film` modifikovať túto metódu tak, že v nej zavoláme metódu `dajUmiestnenie()` nasledovne:

```
public String retazecPreVypis() {
                String s = nazovFilmu + "\n";
                for (int i = 0 ; i < herci.length; i++) {
                        s = s + herci[i]+ " ";
                }
                s = s+ "\n";
                for (int i = 0 ; i < zanre.length; i++) {
                        s = s + zanre[i]+ " ";
                }
                s = s+ "\n";
                s = s+ "Hodnotenie: "+ hodnotenie + "\n";
                s = s+ "Dlzka filmu: "+ dlzkaFilmu + "\n";
                s = s+ dajUmiestnenie() + "\n"; // tu sa zavolá dajUmiestnenie() podľa
typu objektu
                return s;
        }
```

Prečo je to tak? Lebo platí postup pri volaní metódy spomínaný za začiatku tejto kapitolky. Dá sa to predstaviť aj tak, že objekt si osvojí všetky metódy, ktoré dostal od svojej triedy, a z tried predkov si pozbiera postupne iba tie metódy, ktoré zatiaľ nemá. Teda ak sú nejaké metódy prekryté, použije tie, ktoré má v hierarchii dedenia k sebe bližšie.

V tomto prípade máme napríklad objekt triedy `FilmNaDvd`, tak tento objekt má metódu `dajUmiestnenie()` z triedy `FilmNaDvd` a metódu `retazecPreVypis()` z triedy `Film` lebo túto metódu potomkovia nemajú. Takže, keď táto osvojená metóda `retazecPreVypis()` zavolá `dajUmiestnenie()`, použije tú prekrytú metódu, ktorú si osvojil z triedy `FilmNaDvd` a nie metódu z triedy `Film`.

Finty pre komunikáciu s rodičovskou triedou

Niekedy sa stane, že z nejakého dôvodu chceme z oddedenej triedy (napr. `FilmNaDvd`) zavolať metódu z rodičovskej triedy (v tomto prípade `Film`) ktorú sme si v oddedenej triede prekryli. V našom príklade sme prekryli metódu `dajUmiestnenie()`. Ak chceme aj tak zavolať pôvodnú metódu použijeme slovíčko `super`, ktoré označuje rodičovskú triedu.

```
public class FilmNaDvd {
...
  public String dajPovodneUmiestnenie() {
     return "povodne:" + super.dajUmiestnenie();
  }
...
}
```

Abstraktné triedy

Abstraktné triedy úzko súvisia s dedičnosťou a polymorfizmom. V minulej prednáške sme vytvorili hierarchiu tried v ktorej od triedy `Film` dedili triedy `FilmNaDvd`, `FilmNaPaske` a `FilmVPocitaci`. Povedali sme si, že dokážeme z premennej triedy `Film` referencovať objekty tried `Film`, `FilmNaDvd`, `FilmNaPaske` alebo `FilmVPocitaci`. Reálne sme však objekty triedy `Film` nevytvárali. Ak by sme chceli zabrániť tomu, aby sme nemohli vytvárať objekty triedy `Film` mohli by sme túto triedu zmeniť na abstraktnú. Zmena triedy na abstraktnú spočíva v pridaní slovíčka **`abstract`** pred definíciu triedy:

```
public abstract class Film {

//pôvodné telo triedy

}
```

Touto jednoduchou zmenou sa nám funkcionalita nášho programu nijako nezmenila. Naďalej môžeme mať premenné typu `Film` a môžeme z nej referencovať objekty tried `FilmNaDvd`, `FilmNaPaske` a `FilmVPocitaci`. Ak by sme však chceli vytvoriť nový objekt triedy `Film` nepovolilo by nám to:

```
Film matrix = new Film(); //Eclipse vypisuje chybu "Cannot instantiate the type Film"
```

Ak chceme, aby každý film mal svoje médium, na ktorom je uložený, tak zmenou triedy `Film` na abstraktnú si môžeme zabezpečiť, že Java nás upozorní, keď na takéto pravidlo zabudneme.

Iný príklad by mohol byť, že by sme mali triedu `Clovek`, od nej oddedené triedy `Muz` a `Zena` a chceli by ste mať pole ľudí, ale každý by musel byť buď muž alebo žena. Človek je totiž v takomto kontexte abstraktný a nemá zmysel mať človeka, ktorý by nebol ani muž ani žena.

Abstraktné metódy

Abstraktné metódy majú tri hlavné vlastnosti:

- sú to metódy bez tela
- môžu sa vyskytovať iba v abstraktnej triede
- neabstraktné triedy, ktoré dedia od takejto abstraktnej triedy musia túto metódu prekryť

V našom príklade s filmami by sme si mohli zmeniť metódu `dajUmiestnenie()` v abstraktnej triede `Film`. Tým by sme prinútili všetky triedy, ktoré dedia od triedy `Film`, aby mali vlastnú metódu `dajUmiestnenie()`. V triede `Film` teda zapíšeme metódu bez tela spolu so slovíčkom **`abstract`**:

```
public abstract class Film {
...

        public abstract String dajUmiestnenie(); // na konci bodkočiarka a žiade
kučeravé zátvorky {}

...
}
```

Ak by sme potom vytvorili triedu `FilmNaUSB`, ktorá by dedila od triedy `Film`, ale nemala by metódu `dajUmiestnenie()`, Eclipse by protestoval (inherited znamená oddedená):

```
public class FilmNaUSB extends Film { //hlási chybu "The type FilmNaUSB must implement
the inherited abstract method Film.dajUmiestnenie()"

...
}
```

Takto si pomocou mechanizmu abstraktných metód a tried vieme uchrániť funkcionalitu nášho zoznamu filmov lebo vieme, že ak máme pole filmov tak sú v nich iba objekty, ktorých triedy dedia od triedy `Film` a všetky majú funkčnú metódu `dajUmiestnenie()`. Teda bez obáv môžeme volať napríklad nasledujúcu metódu triedy `Film`, ktorá vráti názvy filmov a ich umiestnenia.

```
public class ZoznamFilmov {
...
        public void vypisUmiestnenia() {
                for (int i = 0; i < filmy.length; i++) {
                        System.out.print(filmy[i].getNazovFilmu()+": ");
                        System.out.println(filmy[i].dajUmiestnenie());
                }
        }
...
}
```

Interface-y, t.j. rozhrania

Keď sme rozprávali o zapúzdrení, zdôrazňovali sme, že každé rozumné zadanie sa skladá z dvoch množín požiadaviek.

- s **akými dátami** bude náš program pracovať
- **aké služby** má poskytovať resp. **akú funkcionalitu** má mať.

To, ako budeme dáta reprezentovať t.j. v akých štruktúrach, je v požiadavkach zamlčané. Záleží to od konkrétnej voľby programátora. V zadaní je dôležitejšie, že dáta sme schopní ***nejako*** reprezentovať. Keďže samotná reprezentácia dát je v privátnych premenných, tak vkladanie a čítanie dát z požiadaviek realizujeme cez metódy (napr. cez settery a gettery). Cez metódy realizujeme aj požadovanú funkcionalitu z druhej množiny požiadaviek.

V čase špecifikovania zadania pre našu triedu, ktorá bude reprezentovať výsledné funkčné riešenie, teda špecifikujeme hlavičky metód a popisujeme čo majú tieto metódy s daným vstupom spraviť, prípadne čo majú vrátiť.

Požiadavky sú kontraktom medzi zadávateľom a riešiteľom, ktorý sa riešiteľ zaväzuje splniť. Zadávateľa nezaujíma ako je to urobené vo vnútri riešenia. Zadávateľa zaujíma iba to, či bol dodržaný kontrakt, teda či metódy robia to, čo sa dohodlo. Riešenie je preňho čiernou skrinkou.

Kontrakt sa v Jave zapisuje vo forme `interface`-u. Triedy, ktoré spĺňajú požadovanú funkcionalitu `interface`-u **implementujú tento** `interface`.

Napriek tomu však v návrhu existujú triedy, ktoré sú skôr nositeľmi dát a funkcionalita je skôr okrajová, ale i naopak: v návrhu sa objavujú triedy, ktoré majú minimálny počet inštančných premenných, ale sú skôr zamerané na funkcionalitu.

V našom príklade je nositeľom dát `Film` (a potomkovia), a do druhej skupiny patrí `ZoznamFilmov` so svojou bohatou funkcionalitou na vyhľadávanie filmov, pridávanie, atď.

Triedy zamerané na funkcionalitu (často nazývané *služby*) ju často môžu riešiť viacerými spôsobmi. Vezmime si náš `ZoznamFilmov`: v terajšej podobe reprezentuje filmy, ktoré sú načítavané z textového súboru. Pokojne sa však môže stať, že v neskorších fázach budeme chcieť načítavať súbory z relačnej databázy alebo dokonca z centrálneho súboru umiestneného na internete. Základná množina operácií, ktoré nám zoznam filmov poskytne, je rovnaká bez ohľadu na to, odkiaľ prichádzajú dáta. To opäť súvisí s tým, že pre používateľa sa zoznam filmov bude správať ako čierna skrinka: nebude ho zaujímať *ako* a odkiaľ bude zoznam získavať dáta, dôležité je, že ich dostane.

A práve túto flexibilitu vieme vyjadriť pomocou interfejsov. Aby sme ich vedeli demonštrovať na príklade filmu, budeme musieť `ZoznamFilmov` intenzívne prekopať. Zabudnime na starú verziu a navrhnime novú. (To nie je nič výnimočné: vo vývoji softvéru sa často stáva, že kód je už neudržateľný a je nutné ho navrhnúť nanovo a od podlahy, samozrejme s využítím skúseností z predošlých verzií.)

Najprv sa zamyslime nad operáciami, ktoré nám `ZoznamFilmov` bude ponúkať (prihliadnime pri tom na starú verziu):

- vypíš všetky filmy
- vypíš filmy podľa žánru
- nájdi film podľa názvu
- vlož film do zoznamu
- vymaž film podľa názvu

Uvedené operácie vieme priamo zapísať v podobe interfejsu: každej požiadavke zodpovedá jedna metóda:

```
public interface ZoznamFilmov {
        public void vypisVsetko();

        public void vypisPodlaZanru(String zaner);

        public Film nájdiFilm(String nazov);

        public void vlozFilm(Film film);

        public void vymazFilm(String nazov);

}
```

Tento interfejs naozaj udáva ***čo*** chceme vedieť od zoznamu filmov. Nevraví nič o tom, ***ako*** sa správa vo vnútri, kam ukladáme jednotlivé filmy atď. To však nie je jeho úlohou. Používateľa zoznamu filmov vôbec nezaujímajú "črevá", teda vnútorná implementácia. Dôležité je, aby sa naplnil kontrakt a získal požadované informácie.

Pri syntaxi je dôležité **neuvádzať** žiaden kód do metód. V interfejsi naozaj udávame len hlavičky metód, nepatrí tam žiaden kód!

Zlé príklady sú:
- ~~`void vypisVsetko() {}`~~
- ~~`Film nájdiFilm() { return filmy[0] }`~~

Do interfejsu nemožno uvádzať ani žiadne inštančné premenné! Naozaj, interfejs je len predloha pre funkcionalitu, a inštančná premenná patrí medzi implementačné detaily.

Inak povedané, interfejsy zodpovedajú abstraktným triedam, ktoré
- nemajú žiadne inštančné premenné
- všetky ich metódy sú abstraktné.

Implementácie rozhraní

Samozrejme, musí existovať niekto, kto určí, ***ako*** sa bude správať trieda, ktorá implementuje požadovanú funkcionalitu. Pri úvahách o spôsobe implementácie sa treba zamýšľať nad viacerými hľadiskami: pamäťovou náročnosťou, časovou zložitosťou, alebo dokonca náročnosťou programovania.

V prípade zoznamu filmov je potrebné zvážiť napr.:
- odkiaľ budeme načítavať dáta? Zo súboru? Z databázy? Z internetu?
- v akej dátovej štruktúre ich budeme ukladať? V poli? V zozname?

Úvaha nemusí byť jednoduchá, ale bez ohľadu na zvolený spôsob, je najdôležitejšie dodržaž kontrakt stanovený interfejsom. Používateľa našej triedy vôbec nezaujíma, ktorý spôsob sme si zvolili, ak dostane to,

čo chce, bude spokojný. V prípade potreby môžeme dokonca kompletne prekopať kód v "črevách" a používateľ si nič nevšimne. Práve v tom spočíva sila a výhoda interfejsov.

Pre jednoduchosť sa rozhodneme demonštrovať spôsob, kde budeme filmy načítavať zo súboru a ukladať ich do poľa (teda robiť presne to, čo naša pôvodná verzia `ZoznamuFilmov`:

```
public class SuborovyZoznamFilmov implements ZoznamFilmov {
        Film[] filmy = new Film[0];

        public void vypisVsetko() {

        }

        public void vypisPodlaZanru(String zaner) {

        }

        public Film nájdiFilm(String nazov) {
                return null;
        }

        public void vlozFilm(Film film) {

        }

        public void vymazFilm(String nazov) {

        }

}
```

Ak chceme deklarovať, že trieda implementuje interfejs, uvedieme to kľúčovým slovom `implements`. Je to veľmi podobné ako dedičnosť (ak trieda `Lev` dedí od triedy `Zviera`, píšeme `class Lev extends Zviera`), ale vzhľadom na špecifické vlastnosti interfejsov zvolili dizajnéri svojskú terminológiu a špeciálne kľúčové slovo. Žiadny Java programátor nepovie, že `SuborovyZoznamFilmov` dedí od interfejsu `ZoznamFilmov`. Všetci hovoria, že `SuborovyZoznamFilmov` implementuje interfejs `ZoznamFilmov` a teda

```
public class SuborovyZoznamFilmov implements ZoznamFilmov {
...
}
```

Do triedy sme zároveň uviedli premennú `filmy`, ktorá bude obsahovať pole filmov.

Ak trieda implementuje interfejs, **musí** prekryť všetky jeho metódy. Toto je prirodzené: ak raz interfejs reprezentuje kontrakt, podľa ktorého jedna zo strán musí poskytnúť príslušnú funkcionalitu, tak trieda, ktorá sa rozhodla podieľať na tomto kontrakte, je povinná uviesť spôsob, akým ho naplní (tento spôsob je reprezentovaný kódom v metódach). Ak trieda `SuborovyZoznamFilmov` napĺňa kontrakt zoznamu filmov, nie je mysliteľné, aby sa v nej vynechala implementácia niektorej z metód, pretože to by používateľa tejto triedy zmiatlo a zabránilo realizácii kontraktu.

V predošlom príklade sme teda implementovali všetky metódy, hoci naša implementácia je veľmi biedna a v praxi by táto trieda bola nepoužiteľná. Ale môžeme to skúsiť:

```
public class SuborovyZoznamFilmovTester {
        public static void main(String[] args) {
                ZoznamFilmov zoznamFilmov = new SuborovyZoznamFilmov();
                Film matrix = zoznamFilmov.nájdiFilm("The Matrix");
                System.out.println(matrix);
```

```
                Film casablanca = zoznamFilmov.nájdiFilm("Casablanca");
                System.out.println(matrix);
        }
}
```

Po spustení tejto triedy získame výpis `null` a to preto, že metóda `nájdiFilm()`, tak ako sme ju uviedli vyššie, vždy a stále vracia `null`.

Zamerajme pozornosť na vytváranie inštancie:

```
ZoznamFilmov zoznamFilmov = new SuborovyZoznamFilmov();
```

Na ľavej strane máme premennú typu `ZoznamFilmov`, čo je interfejs, a na pravej vytvárame konkrétnu inštanciu. Voľne povedané, interfejsu priraďujeme implementáciu. To je úplne korektné, pretože medzi interfejsom `ZoznamFilmov` a implementáciou `SuborovyZoznamFilmov` existuje hierarchia dedičnosti a teda všeobecnejšiemu typu (`ZoznamFilmov`) vieme priradiť konkrétnejší `SuborovyZoznamFilmov`).

Tento výraz sa dá intepretovať aj inak: vľavo povieme **aký kontrakt** chceme používať (***čo*** chceme), a vpravo uvedieme triedu, ktorá tento kontrakt realizuje (***ako*** to spravíme).

Pri pohľade na triedu, ktorá implementuje interfejs (realizuje kontrakt) máme jeden vážny problém. Máme len také metódy, ktoré odovzdávajú alebo vypisujú informácie, ale žiadny spôsob, ako naplniť inštanciu zoznamu filmov skutočnými v minulosti vytvorenými dátami. Ale to vyriešime jednoducho tak, že dáta, ktoré sú jedinečné pre daný typ zoznamu filmov pošleme cez konštruktor. Konštruktor totiž nikdy v interfejse neuvádzame, lebo je jedinečný pre kažú triedu, ktorá ho implementuje. Ďalšie spôsoby už vyžadujú hlbšie znalosti javy.

```
File subor = new File("filmy.txt");
ZoznamFilmov zoznamFilmov = new SuborovyZoznamFilmov(subor);
```

Takýto konštruktor sme už mali aj minule.

Teraz nám len stačí upraviť metódy v triede `SuborovyZoznamFilmov` tak, aby naozaj napĺňali kontrakt. Ukážme si to na jednom prípade metódy `vlozFilm()` a zvyšné metódy ponecháme na pozorného čitateľa.

```
public class SuborovyZoznamFilmov implements ZoznamFilmov {
...
        public void vlozFilm(Film film) {
                Film[] novePole = new Film[filmy.length + 1];
                for (int i = 0; i < filmy.length; i++) {
                        novePole[i] = filmy[i];
                }
                novePole[filmy.length] = film;
                filmy = novePole;
        }
...
}
```

Interfejsy ako roly

Ďalším typickým príkladom použitia interfejsov je prípad, keď interface predstavuje "rolu", ktorú môže napĺňať viacero rozličných tried. V Jave je takéto použitie interfejsov bežné:

- interface `Serializable` znamená, že trieda, ktorá ho implementuje, môže byť uložená do postupnosti bajtov a uložená napr. do súboru. Serializácia je ako zaváranie ovocia -- vezmeme objekt

a uložíme ho na horšie časy, odkiaľ ho v prípade potreby vieme vytiahnuť a použiť. Trieda, ktorá ho implementuje, napĺňa rolu "Zavárateľný" (voľný, ale absolútne nepresný preklad).
- interface `Comparable` spomenieme nižšie: predstavuje rolu "porovnateľného" objektu. Inštancia triedy, ktorá ho implementuje, sa dokáže porovnať s inou inštanciou.

Ak by sme definovali interface `Nakresliteľný` s jedinou metódou `nakresliSa(WinPane)`, akýkoľvek objekt triedy, ktorá tento interfejs implementuje, môže byť nakreslený na WinPane bez ohľadu na to či dedí od `Turtle` alebo hociakej inej triedy. Takto vieme napr. dosiahnuť triedu `EuroMinca`, ktorý dedí od `Minca` a implementuje interfejs `Nakresliteľný`, aby sa nám nakreslila na plochu.

Alternatívnym príkladom je definovať interface `UložiteľnýNaDisk` s metódou `ulož(File súbor)`, kde objekt triedy, ktorá tento interfejs implementuje možno uložiť na disk do textového súboru. Ak zmeníme `SúborovýZoznamFilmov` tak, aby implementoval tento interfejs a dodáme kód, ktorý obsah zoznamu uloží na disk, vieme získať funkcionalitu, ktorou daný súborový zoznam uložíme.

Takéto použitie je veľmi výhodné, pretože sa ním dosahuje **viacnásobná dedičnosť**, teda trieda môže implementovať viacero interfejsov. Trieda v Jave môže dediť od jedinej triedy (je to zámer autorov), ale môže implementovať viacero interfejsov.

Príkladov na interfejsy ako roly bude ešte viac, dostaneme sa k nim po vysvetlení kolekcií.

Balíčky

V doterajšom kurze sme používali len triedy s jednoduchými, zvyčajne slovenskými názvami. Mali sme korytnačky `Turtle`, ale mali sme aj `Film`y, či `FilmVPocitaci`. Java je však stavaná na budovanie obrovských systémov s tisíckami tried rozložených cez desiatky projektov a knižníc. V takýchto megaprojektoch nie je nič výnimočné, keď nastane kolízia v názvoch tried.

Vezmime si reálny príklad: do projektu si zavedieme triedu reprezentujúcu atribút nejakého tovaru (Predávame televízory, ktoré majú atribút *farba*) a nazveme si ju správne po anglicky `Attribute`. Lenže beda! Taká trieda už existuje, pretože niekde v hlbinách Javy sa nachádza trieda s rovnakým názvom, ktorou sa nastavujú atribúty tlačovej zostavy. A keď sa rozhodneme zaviesť do projektu knižnicu na spracovanie webových stránok, kde `Attribute` reprezentuje syntaktický prvok jazyka HTML, máme problém.

Jedným z riešení by bola dohoda a premenovanie tried: náš atribút by sme premenovali na `ObjectAttribute`, a druhý konflikt vyriešili dohodou s autorom knižnice pre spracovanie HTML. Problém však tkvie v nemožnosti dohodnúť sa s autormi Javy.

Našťastie, takýmto problémom možno ľahko predchádzať, pretože v Jave existujú balíčky.

Balíčky ako analógia súborového systému

V každom rozumnom operačnom systéme existuje adresárová štruktúra, ktorá udržiava poriadok v súboroch. Príklad z Windowsu:

```
c:\Users\Public\Music\Sample Music\Kalimba.mp3
```

hovorí, že máme súbor `Kalimba.mp3` na disku `C` v adresári `Users\Public\Music\Sample Music`. Inak povedané, máme adresár `Users` s podadresárom `Public`, ktorý má podadresár `Music`, atď. (Kto neverí, nech tam beží, príklad je prevzatý z Windows 7.)

Keby adresáre neexistovali (čo bol stav v praoperačných systémoch na 8-bitových počítačoch), veľmi rýchlo by nastal na disku chaos.

Balíčky plnia presne úlohu organizovanosti jednotlivých tried. Každá javovská trieda patrí do niektorého balíčka (tak ako každý súbor je v nejakom adresári) a samotné balíčky sú organizované v hierarchickej štruktúre (práve tak, ako adresár môže obsahovať ďalšie podadresáre). Každá trieda v Jave má *úplný názov* (fully qualified name), ktorý predstavuje ,,úplnú cestu" v hierarchii balíčkov.

Keď sme napríklad spomínali triedu `File`, ktorá reprezentuje súbor, v skutočnosti sme sa rozprávali o triede s úplným názvom `java.io.File`. Balíček `java` má podbalíček `io`, ktorý obsahuje triedu `File`.

Asi je zjavné, že takáto štruktúra predchádza problémom z úvodu. Už nemusíme premenovávať triedy s konfliktnými názvami, stačí, že majú triedy rozličné úplné názvy. A naozaj:

- trieda z útrob Javy má názov `javax.print.attribute.Attribute`
- trieda pre spracovávanie HTML má plný názov `org.htmlparser.Attribute`
- a našu triedu môžeme zaradiť napr. do balíčka `sk.upjs.ics.paz`, čiže bude mať názov `sk.upjs.ics.paz.Attribute`

Panuje úzus, že názvy balíčkov sa odvodzujú od webových adries vývojára, čím sa zaručí ich jednoznačnosť. (Keďže spadáme pod Ústav informatiky, ktorý má web ics.upjs.sk, je to vhodný kandidát na názov balíčka). Na

rozdiel od webových adries sa cesta udáva v obrátenom poradí.

Hierarchia balíčkov

Každá trieda v Jave patrí do nejakého balíčka. Dokonca i úplne triviálna trieda

```
public class HlúpaVec {
  // tu nie je nič, žiadne metódy!
}
```

patrí do tzv. implicitného balíčka, ktorý sa nachádza na vrchole hierarchie (je analógiou koreňového adresára).

Ako zaradiť triedu do balíčka

Ak chceme zaradiť triedu do balíčka, potrebujeme spraviť dve veci:

1. do úplne prvého riadku zadať klauzulu s menom balíčka, v ktorom je trieda
2. zaradiť ju do správneho adresára

Klauzula `package`

Vezmime si triedu `Film` z predošlých príkladov a zaraďme ju do balíčka `sk.upjs.ics.paz.filmy`. To dosiahneme klauzulou

```
package sk.upjs.ics.paz.filmy;
```

Výsledná trieda bude vyzerať:

```
package sk.upjs.ics.paz.filmy;

public class Film {
...
}
```

Zaradenie do správneho adresára

O vytvorenie adresárov a zaradenie do správneho balíčka a aj adresára (teda v našom prípade do adresára src/sk/upjs/ics/paz/filmy) sa nám postará Eclipse.

Vytvorme si najprv nový package cez pravý klik na názov projektu > new > package, kde zadáme názov package-u `sk.upjs.ics.paz.filmy` . Následne si označíme všetky java súbory, ktoré chceme presunúť a metódou drag&drop ich presunieme na nové miesto. Môžeme sa pozrieť, že vo všetkých presunutých súboroch pribudol nový riadok

```
package sk.upjs.ics.paz.filmy;
```

Použitie tried z iných balíčkov

V bežnom kóde platí zásada, že vždy keď použijeme názov triedy (napr. v deklarácii dátového typu premennej), tak kompilátor predpokladá, že ide o meno triedy, ktorá sa nachádza v rovnakom balíčku.

V predošlých príkladoch s dedičnosťou filmov sme mali šesť tried: `Film`, `FilmNaDvd`, `FilmNaPaske`, `FilmVPocitaci`, `ZoznamFilmov` a `Spustac`. Skúsme zaradiť všetky triedy do balíčka `sk.upjs.ics.paz.filmy`.

Ak potom v metóde `main()` triedy `Spustac` (teda triedy s plným menom `sk.upjs.ics.paz.filmy.Spustac`) použijeme triedu `FilmNaDvd`, kompilátor bude predpokladať, že

chceme používať triedu `sk.upjs.ics.paz.filmy.FilmNaDvd`. Inak povedané, pred názov triedy `FilmNaDvd` si domyslí plnú cestu v balíčkoch. Kód

```
package sk.upjs.ics.paz.filmy;

public class Spustac {
        public static void main(String[] args) {
                FilmNaDvd matrix = new FilmNaDvd();
        }
}
```

bude teda ekvivalentný kódu

```
package sk.upjs.ics.paz.filmy;

public class Spustac {
        public static void main(String[] args) {
                sk.upjs.ics.paz.filmy.FilmNaDvd matrix = new
sk.upjs.ics.paz.filmy.FilmNaDvd();
        }
}
```

Je však asi zjavné, že takýto kód by bol veľmi rýchlo neprehľadný a teda domýšľanie mien má svoju logiku a zmysel.

Používanie tried z iných balíčkov a importy

Ak chceme používať triedu z iného balíčka, nie je nič jednoduchšie, stačí ju uviesť pomocou jej plného názvu. Tento prístup je ľahký na pochopenie, ale nik to tak nerobí:

```
package sk.upjs.ics.paz.filmy;

public class Spustac {
        public static void main(String[] args) {
                java.io.File súbor = new java.io.File("data.txt");
        }
}
```

V triede `sk.upjs.ics.paz.filmy.Spustac` používame triedu `java.io.File` a teda sa na ňu musíme odkázať plným menom.

Tento prístup je prostý, pretože sa drží jasnej zásady: Ak chceš používať triedu z iného balíčka, musíš sa na ňu odkázať plným menom. Existujú však projekty, ktoré používajú mimoriadne dlhé názvy, a tento spôsob je pre nich fatálny:

```
org.springframework.aop.framework.autoproxy.metadata.AttributesThreadLocalTargetSourceCreator
c
  = new
org.springframework.aop.framework.autoproxy.metadata.AttributesThreadLocalTargetSourceCreator();
```

Našťastie existuje idea importov, ktoré tento problém odstraňujú.

Importy

Pomocou importov je možné skrátiť zápis kilometrových názvov tried. Dajme si príklad:

```
package sk.upjs.ics.paz.filmy;

import java.io.File;
```

```
public class Spustac {
        public static void main(String[] args) {
                File súbor = new File("data.txt");
        }
}
```

Klauzula `import java.io.File` hovorí, že akonáhle sa v kóde spomenie trieda `File`, znamená to v skutočnosti triedu `java.io.File`. Import dramaticky skracuje a sprehľadňuje zápisy využívajúce triedy v iných balíčkov.

Importy možno zoskupovať: ak trieda používa viacero importov z jedného balíčka:

```
import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;
```

možno zápis skrátiť na

```
import java.io.*;
```

Takýto zápis hovorí "importni všetky triedy z balíčka `java.io`".

Pozor ale: hromadný import sa týka len tried z konkrétneho balíčka. Neimportuje ani triedy z nadradených balíčkov, ani z podbalíčkov!

Rovnako platí, že v importe sa dá použiť len jediná hviezdička: snaha o import všetkých balíčkov Javy zlyhá: ~~`java.*.*.*`~~

Automatické importy

Existujú dve situácie, keď import tried prebieha automaticky. Prvú sme si už spomínali: trieda, ktorá sa nachádza v rovnakom balíčku, nemusí byť importovaná.

Druhým prípadom sú triedy z balíčka `java.lang`. Nemusíme teda importovať bežné Java triedy, teda `String` (`java.lang.String`), `System` (`java.lang.System`) a pod.

Importy a implicitné balíčky

Trieda, ktorá nemá žiadnu deklaráciu `package` patrí do implicitného balíčka na vrchole hierarchie. V rozumnej knižnici a projekte by sa však takéto triedy nemali nachádzať, a ak sa tak dialo, tak to bolo kvôli jednoduchosti a z didaktických dôvodov.

Implicitný balíček je síce zdanlivo jednoduchý, ale môže viesť k neporiadku a najmä ku konfliktom v názvoch, ktoré sme spomínali na začiatku. Trieda v tomto balíčku má ešte jednu nevýhodu: nemôže byť importovaná z tried zaradených v balíčkoch.

Modifikátory viditeľnosti

Vidieľnosť tried, metód a inštančných premenných sa dá nastavovať pomocou modifikátorov vidieľnosti. Platí, že ak niečo nevidíme, nevieme s tým pracovať priamo. Nevidieľná trieda sa prejavuje tak, že nevieme vytvárať premenné typu tejto triedy ani objekty tejto triedy z miesta, kde triedu nevidíme. Neviditeľné metódy nevieme z danúho miesta zavolať a neviditeľné inštančné premenné priamo meniť.

Triedy, ktoré majú vlastný súbor s rovnakým menom, môžu mať dva typy modifikátorov:

- **public** - viditeľné všade,

```
public class NejakaVerejnaTrieda {
...
}
```

- *(nič)* - viditeľné iba v svojom balíčku, t.j. neviditeľné vo všetkých iných balíčkoch aj podbalíčkoch

```
class NejakaTrieda {
...
}
```

To, čo sa nachádza v týchto triedach, teda inštančné premenné a metódy (časom možno narazíte aj na vnorené triedy) majú možnosť mať jeden zo 4 rôznych modifikátorov viditeľnosti

- **public** - viditeľné všade,

```
public int verejnáPremenná;
...
public void verejnáMetóda(){
}
```

- **protected** - viditeľné v svojom balíčku a vo všetkých potomkoch aj keby boli z iných balíčkov

```
protected int chránenáPremenná;
...
protected void chránenáMetóda(){
}
```

- *(nič)* - viditeľné iba v svojom balíčku

```
int balíčkováPremenná;
...
void balíčkováMetóda(){
}
```

- **private** - viditeľné iba vo svojej triede

```
private int súkromnáPremenná;
...
private void súkromnáMetóda(){
}
```

Nasledujúca tabuľka ukazuje prehľad dostupnosti (viditeľnosti) k prvkom triedy pre daný modifikátor viditeľnosti

Modifikátor	Trieda	Package	Podtrieda	Všade
public	áno	áno	áno	áno
protected	áno	áno	áno	nie
(nič)	áno	áno	nie	nie

private	áno	nie	nie	nie

To, aký modifikátor použiť, záleží od konkrétneho návrhu. V reálnom projekte platí, že modifikátory by mali byť čo najprísnejšie. Začíname s **private** a iba keď máme dobrý dôvod, nastavujeme voľnejšie modifikátory.

Modifikátor **public** by mali mať iba tie metódy, ktoré poskytujeme iným programátorom a používateľom nášho programu. Typicky sú to metódy zo zadania, t.j. inferfejsu, ktorý má všetky metódy **public** aj keď to nenapíšeme.

Triedenie

Triedenie (*sorting*) je typickým príkladom problému, s ktorým sa i normálny človek bežne stretáva, a ktorý je predmetom dlhoročného stále trvajúceho výskumu. Triedenie je potrebné na usporiadanie mien v telefónnom zozname, manažér potrebuje zotriedené údaje na to, aby zistil finačný zisk za jednotlivé divízie, a používateľ Windowsu chce mať utriedené súbory, aby v nich vedel rýchlo vyhľadávať.

V súčasnosti existuje množstvo triediacich algoritmov rozličnej zložitosti a implementačnej náročnosti. Niektorým z nich sa budete venovať v budúcom semestri. V tomto kurze sa naučíme len základné techniky triedenia a spôsoby, ktorými je ho možné dosiahnuť.

Keďže bežný vývojár nepotrebuje vlastnú implementáciu triediacich algoritmov, v Jave existuje viacero zabudovaných spôsobov, ktorými možno triediť nielen bežné ,,utriediteľné" veci (čísla, reťazce), ale aj akékoľvek vlastné objekty.

Triedenie čísiel

Najtypickejším príkladom je triedenie čísiel, ktoré sú uložené v poli, čo je možné dosiahnuť prostými dvoma riadkami.

```
int[] platy = new int[] { 2400, 1000, 3700, 1800};
Arrays.sort(platy);
// pole <platy> je teraz utriedené
```

Zavolaním statickej metódy Arrays.sort(pole) vieme zotriediť ľubovoľné pole čísiel podľa veľkosti.

Metóda `sort()` je preťažená pre mnoho základných typov. Vieme triediť pole `short`ov, `long`ov, či `double`ov alebo `float`ov.

Triedenie reťazcov

Triedenie reťazcov je úplne analogické k triedeniu čísiel. Reťazce budú usporiadané v lexikografickom usporiadaní (t.j. normálne utriedenie podľa abecedy ako v telefónnom zozname):

Reťazec a_1 a_2 ... a_n je v usporiadaní pred reťazcom b_1 b_2 ... b_n, ak

- buď a_1 < b_1
- alebo existuje *k* z intervalu [1,n] také, že pre každé *i* < *k* je *a_i* = *b_i* a *a_k* < *b_k*

V prípade, že sú reťazce nerovnakej dĺžky, kratší reťazec sa vyplní medzerami na dĺžku dlhšieho, a predpokladá sa, že znak medzery je menší ako akýkoľvek iný znak.

Podľa definície je

- `PES` < `VELRYBA`, lebo `P` < `V`. V skutočnosti však porovnávame `PES...` < `VELRYBA`
- `PERO` < `PES`, lebo porovnávame `PERO` < `PES.` a existuje *k* = *3*, kde prefixy reťazcov sa rovnajú (*PE* == *PE*) a `R` < `S`.

Kód pre utriedenie:

```
String[] mená = new String[] { "Zuzana", "Adam", "Cyril", "Boris"};
Arrays.sort(mená);
// pole <mená> je teraz utriedené
```

Triedenie objektov

V Jave však existuje možnosť triediť ľubovoľné objekty, a nie iba čísla či reťazce. V tomto prípade však musíme poznať kritérium, na základe ktorého vieme povedať, že jeden objekt je v usporiadaní pred nejakým iným.

- Kedy je číslo v usporiadaní pred iným? Ak je menšie ako to druhé.
- Kedy je reťazec v usporiadaní pred iným? Ak je v lexikografickom usporiadaní pred druhým reťazcom.

V týchto prípadoch sú kritériá jasné. Čísla a reťazce majú svoje *prirodzené usporiadanie* (natural ordering). Čo však s objektami? Kedy je film v usporiadaní pred iným? Musíme sa rozhodnúť a dohodnúť. Môžeme napríklad povedať, že film je v usporiadaní pred iným filmom, ak je jeho názov v lexikografickom usporiadaní pred názvom druhého filmu. (Potom bude `Casablanca` pred `The Matrix`).

Rozhodnutie vieme zaviesť do ľubovoľnej triedy pomocou interfejsu `java.lang.Comparable`. Ak trieda implementuje tento interfejs (a teda prekrýva jeho jedinú metódu `compareTo()`, vie plniť rolu objektu, ktorý sa vie porovnať s iným. Táto metóda nasledovnú hlavičku:

```
int compareTo(TypObjektu druhýObjekt)
```

a od implementátora sa očakáva, že táto metóda vráti

- záporné číslo ak je objekt, ktorého metódu sme volali, v usporiadaní **pred** objektom `druhýObjekt`, teda je "menší" ako `druhýObjekt`
- 0, ak sú objekty v usporiadaní rovnaké
- kladné číslo, ak je objekt, ktorého metódu sme volali, v usporiadaní **za** objektom `druhýObjekt`, teda je "väčší" ako `druhýObjekt`

Príklad triedy pre triedu `Film` je nasledovný:

```
import java.lang.Comparable;

public class Film implements Comparable<Film> {
  private String nazovFilmu;

  public int compareTo(Film inyFilm) {
    // zistíme, či nazovFilmu je v usporadaní pred inyFilm.getNazovFilmu()
  }
}
```

Pri uvádzaní interfejsu `Comparable` nás čaká drobný syntaktický zádrheľ: do lomených zátvoriek musíme uviesť dátový typ, ktorý bude použitý v porovnávaní. Keďže v tomto prípade porovnávame Film, v deklarácii `implements` musíme uviesť `Comparable<Film>`.

Ako zistíme, či je názov aktuálneho filmu v usporiadaní pred iným? Jednoducho: reťazec `String` má metódu `compareTo()`, ktorá presne vráti záporné číslo, nulu alebo kladné číslo, v závislosti od toho, či je reťazec pred, rovnako, alebo za druhým reťazcom. Táto metóda sa používa pri normálnom triedení `String`ov.

```
public int compareTo(Film inyFilm) {
  return this.nazovFilmu.compareTo(inyFilm.getNazovFilmu());
}
```

Ak vieme porovnať dve inštancie `Film`ov, nič nám nebráni triediť pole filmov:

```
FilmNaDvd matrix = new FilmNaDvd();
// ... nastavime vsetky atributy

FilmNaPaske shawshank = new FilmNaPaske();
// ... nastavíme všetky atribúty

FilmVPocitaci fontana = new FilmVPocitaci();
// ... nastavíme všetky atribúty

Film[] filmy = new Film[] { matrix, shawshank, fontana };
Arrays.sort(filmy);

// pole <filmy> je teraz zotriedené podľa názvu
```

Triedenie rovnakých objektov rôzne

Triedenie filmov s využitím interfejsu `Comparable` má jednu nevýhodu: kým čísla a reťazce majú jasne definované prirodzené usporiadanie, v prípade filmov jestvuje viacero kritérií, podľa ktorých ich môžeme triediť. Čo ak chceme usporiadavať filmy podľa hodnotenia? Alebo podľa počtu hercov? Metódou `copareTo()` však vieme nastaviť iba jednu možnosť usporiadania.

Na porovnávanie objektov sa môžeme pozerať dvojako, presnejšie, z dvoch perspektív. Z prvej perspektívy môžeme povedať *ja sa viem porovnať s iným objektom." Druhá perspektíva hovorí* ja viem porovnať dva objekty." Prvej perspektíve zodpovedá metóda `compareTo()` uvedené priamo v triede, ktorej objekty porovnávame (u nás vo `Film`e).

Druhej perspektíve zodpovedá interfejs `java.util.Comparator` a metóda `int compare(TypObjektu o1, TypObjektu o2)`. Trieda, ktorá implementuje `Comparator`, dokáže porovnať dva objekty a povedať, či je prvý v usporiadaní pred druhým.

```
public class FilmPodľaMenaComparator implements Comparator<Film> {
  public int compare(Film film1, Film film2) {
    return film1.getNazovFilmu().compareTo(film2.getNazovFilmu());
  }
}
```

V príklade sme urobili triedu implementujúcu `Comparator` filmov (opäť použijeme trik z lomenými zátvorkami), ktorý porovná dva filmy a určí kritérium ich porovnania -- v tomto prípade porovná názvy filmov.

Implementovať komparátor podľa hodnotenia je potom ľahké:

```
import java.util.Comparator;

public class FilmPodľaHodnoteniaComparator implements Comparator<Film> {
        public int compare(Film film1, Film film2) {
                return film1.getHodnotenie() - film2.getHodnotenie();
        }
}
```

A ako zotriedime samotné objekty? Metóda `sort()` má preťaženú metódu s dvoma parametrami: prvý je pole, ktoré chceme triediť, a druhý inštancia typu `Comparator`. Stačí teda vytvoriť inštanciu niektorého z vyššieuvedených komparátorov a budeme vedieť triediť ľubovoľné objekty podľa ľubovoľného kritéria.

```
FilmNaDvd matrix = new FilmNaDvd();
// ... nastavime vsetky atributy
FilmNaPaske shawshank = new FilmNaPaske();
// ... nastavíme všetky atribúty
FilmVPocitaci fontana = new FilmVPocitaci();
// ... nastavíme všetky atribúty

Film[] filmy = new Film[] { matrix, shawshank, fontana };

Comparator<Film> kritérium = new FilmPodľaHodnoteniaComparator();
Arrays.sort(filmy, kritérium);

// pole <filmy> je teraz zotriedené
```

Pole `filmy` sa teraz zotriedilo podľa hodnotenia. Ak chceme zmeniť kritérium na triedenie, stačí zmeniť jediný riadok:

```
Comparator<Film> kritérium = new FilmPodľaMenaComparator();
```

Výmenou jediného riadku sme zmenili správanie algoritmu, a to vďaka polymorfizmu, kvôli ktorému sa premenná typu `Comparator` správa raz tak, raz onak -- a to v závislosti od skutočného typu premennej.

Ak by sme niekedy potrebovali triediť v opačnom poradí nie je potrebné písať nový komparátor s obrátenou logikou. Stačí nám na to zavolať "otáčač komparátorov" a použiť ho namiesto pôvodného koparátora

```
Comparator<Film> porovnavac = FilmPodlaHodnoteniaComparator();

Arrays.sort(zoznamFilmov, Collections.reverseOrder(porovnavac));
// pole je utriedené podľa hodnotenia zostupne
```

V praxi sa ešte často stretneme s tým, že potrebujeme triediť reťazce nie lexikograficky ale tak ako to určuje slovenská abeceda. V štandardnom lexikografickom usporiadaní napríklad platí, že Z < Á alebo CH < D. V slovenčine však platia opačné znamienka. Na vyriešenie tohto problému stačí použiť komparátor, ktorý už za nás naprogramovali autori Javy a ktorý rieši triedenie podľa národných zvyklostí.

```
import java.text.Collator;
import java.util.Arrays;
import java.util.Locale;

public class Skuska {

        public static void main(String[] args) {
                String[] mená = new String[]{"Adam", "Cecília",
"Cháron","Ábel","Daniel"};

                Collator skPorovnavac = Collator.getInstance(new Locale("sk"));

                Arrays.sort(mená, skPorovnavac);
                System.out.println(Arrays.toString(mená));  //Vypíše sa [Ábel, Adam,
Cecília, Daniel, Cháron]
        }
}
```

Výnimky (vyhadzovanie, vlastné výnimky)

Už vieme, že pri volaní nejakej metódy môže nastať za istých okolností nejaká výnimka. Keď ju neodchytíme, program nám skončí a vypíše sa *stack trace*.

Tiež vieme, že výnimky, ktorým vieme zabrániť overovaním vhodných podmienok pred spustením kritickej operácie neriešime odchytávaním, ale práve overovaním podmienok vopred.

Výnimky, ktorým nevieme predchádzať overovaním vstupných podmienok môžeme odchytávať v **try-catch-finally** blokoch.

```
try {

  // blok príkazov z ktorého odchytávame výnimky

} catch (TypVýnimky1 e) {
  // vysporiadanie sa s daným typom výnimky
} catch (TypVýnimky2 e) {
  // vysporiadanie sa s daným typom výnimky
} finally {
  // príkazy, ktoré sa vykonajú bez ohľadu na to, či bola vyhodená výnimka alebo nie
  // a bez ohladu na to či sa výnimka vzniknutá v bloku try odchytila alebo nie
}
```

Vyhadzujeme výnimky

Výnimky sa používajú pri výnimočných stavoch, s ktorými si daná metóda nevie poradiť. Ak si metóda vie poradiť sama, nemá zmysel výnimku vyhadzovať. Metóda vyhadzuje výnimku s tým, že je šanca, že sa s tým kus kódu, ktorý danú metódu volal, vie vysporiadať. Dôvodom na vyhodenie výnimky je väčšinou neschopnosť vykonať požadovanú akciu a vrátiť požadovanú hodnotu. Príčinou môže byť:

- Volajúci kód porušil kontrakt - v dokumentácii sme uviedli, za akých podmienok vie metóda vykonať očakávanú činnosť, ale tieto podmienky neboli dodržané
    - metóde na výpočet logaritmu podhodíme záporné číslo
    - metóde na nastavovanie rodného čísla pošleme reťazec "Michael Jackson is not dead!"
- Nie sú potrebné podmienky na úspešné vykonanie kódu
    - nemáme internet
    - padol server, s ktorým sa potrebujem rozprávať
    - súbor sa nenašiel
    - zaplnil sa disk
    - nemám dáta o ktoré žiadaš

Doteraz, sme pracovali iba s výnimkami, ktoré hádzali cudzie metódy. Výnimky môžeme vyhadzovať aj zo svojich metód. Môžeme vyhadzovať už známe výnimky (`java.lang.NumberFormatException`, `java.lang.IllegalArgumentException`, ...) alebo si môžeme vyrobiť aj vlastné výnimky. Výnimky sú obyčajné triedy, ktoré dedia od triedy `java.lang.Exception`. Môžu teda mať vlastné inštančné premenné, metódy a konštruktory. Dokážeme tak odoslať veľmi komplexnú informáciu o vzniknutej sitácii, a nezriedka aj návrh na riešenie. Vyrobme si vlastnú výnimku `ZápornýVstupException`.

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

Vyrobme si príklad, ktorý počíta faktoriál čísla `n`. Ak príde `n` záporné, vyhodíme výnimku `ZápornýVstupException`.

```
public class Pocitadlo {
        public int faktorial(int n) throws ZápornýVstupException {
                if (n < 0)
                   throw new ZápornýVstupException(); //vyhadzujeme výnimku
                int faktorial = 1;
                for (int i = 1; i < n; i++) {
                        faktorial = faktorial * i;
                }
                return faktorial;
        }
}
```

Všimnite si, že výnimka je obyčajný objekt, vytvorený cez **new**. Pomocou kľúčového slova **throw** okamžite ukončíme beh metódy a vyhodíme výnimku. Výnimky, ktoré môžeme vyhadzovať, uvádzame v hlavičke za kľúčovým slovom **throws**. Ak môžeme vyhadzovať viac druhov výnimiek, oddeľujeme ich čiarkami.

Keď teraz budeme volať túto metódu, Java od nás bude požadovať ošetrenie výnimky. Zabalíme teda toto volanie do **try-catch** blokov.

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

Aby mohli bez ujmy na zdraví používať vaše programy aj iní ľudia (resp. po dlhšom čase aj vy sami), nikdy nevyhadzujte všeobecné výnimky, napr:

```
public int faktorial(int n) throws Exception {
       ...
}
```

Ak nastane všeobecná výnimka, používateľ vášho kódu nemusí pochopiť, a obvykle ani nepochopí, čo sa stalo a čo bolo príčinou. Na druhej strane, keď dostane výnimku s rozumným menom, napr. `ZápornýVstupException`, už z jej názvu vie veľmi rýchlo pochopiť, že problém zrejme spočíva v záporných vstupoch.

(Ne)ošetrujeme výnimky

Ak výnimku spôsobilo volanie inej metódy, máme dve možnosti:

- výnimku odchytíme v catch bloku a ošetríme ju
- pošleme výnimku na spracovanie metóde, ktorá našu metódu volala - výnimka vybuble vyššie

V druhom prípade neošetrenú výnimku uvedieme v `throws` a necháme vyššie metódy nech to riešia. Skôr či neskôr sa však toto prehadzovanie zodpovednosti za ošetrenie vzniknutého stavu musí niekde zastaviť -- v opačnom prípade výnimka vybuble von z metódy `main()`, čo vyústi v násilné ukončenie programu. Napríklad, keby sme pri čítaní súboru neodchytili `FileNotFoundException` mohli by sme ju proste nechať vybublať vyššie:

```
public class Pocitadlo {
        public List<Integer> nacitajCisla(File f) throws FileNotFoundException {
                List<Integer> cisla = new ArrayList<Integer>();
                Scanner citac = new Scanner(f); //neodchytená výnimka
FileNotFoundException
                while(citac.hasNextInt()) {
                        cisla.add(citac.nextInt());
                }
                citac.close();
                return cisla;
        }
}
```

Obveľa zaujímavejšou možnosťou je prebaliť výnimku do novej, popisnejšej. Vytvoríme si novú výnimku `PocitadloException`, ktorej cez konštruktor vložíme textový popis a aj pôvodnú výnimku `FileNotFoundException`, ktorú prebaľujeme. Ten, čo odchytí výnimku má tak lepšiu informáciu o tom, čo sa stalo, pretože má vlastne popisnú výnimku aj pôvodnú výnimku v jednom. Konštruktory triedy `PocitadloException` si vygenerujeme cez **Source > Generate Constructors From Superclass**.

```
public class Pocitadlo {
          public List<Integer> nacitajCisla(File f) throws PocitadloException {
                List<Integer> cisla = new ArrayList<Integer>();
                Scanner citac = null;
                try      {
                        citac = new Scanner(f);
                        while(citac.hasNextInt()) {
                                cisla.add(citac.nextInt());
                        }
                } catch (FileNotFoundException e) {
                        throw new PocitadloException("Čísla nenačítateľné", e);
                } finally {
                        if (citac!=null) citac.close();
                }
                return cisla;
          }
        }
```

Prebaľovanie výnimiek má zmysel aj preto, že používateľ cudzieho kódu môže mať s nízkoúrovňovými výnimkami, vybublanými z vnútra programu, problém. Má pochopiť, čo sa stalo, a hlavne čo bolo príčinou a ako to treba riešiť. Predstavte si, že vám cudzí modul vyhodí napr. výnimku `NumberFormatException` a vy ste pritom volali metódu na poslanie matice na server a nevidíte súvis.

Príklad správneho prebaľovania výnimiek bublacúcich z vnútra nejakého projektu:

- `VSúboreChýbaPremennáException: početBalíčkov`
- `KonfiguráciaNemáPremennúException: početBalíčkov`
- `ModulKonfiguráciaOutOfDateException`
- `StarrýJarSúborException: konfiguracia.jar`

Keby sa k vám dostala prvá výnimka, neviete čo s tým. Ale na základe poslednej už viete, že potrebujete zohnať novší modul konfigurácie.

Čo všetko môžeme vyhadzovať

Už vieme, že v Jave sú dva druhy výnimiek - kontrolované a nekontrolované. Okrem nich môžu metódy vyhadzovať aj chyby.

- ***Kontrolované výnimky*** sa musia buď odchytiť v `try-catch` blokoch alebo nechať vybublať do volajúcej metódy. Ak sa kontrolované výnimky vyhadzujú, musia sa uviesť v hlavičke za `throws`.

- ***Nekontrolované výnimky*** sú potomkovia triedy `RuntimeException`. Nemusia odchytiť a nemusia sa uvádzať v `throws`.
- ***Chyby*** sú potomkovia triedy `Error`. Podobne ako nekontrolované výnimky sa nemusia odchytávať a uvádzať v `throws`.

Chyby (`Errors`) predstavujú taký stav systému, z ktorého sa aplikácia nemá šancu zotaviť. Väčšinou ju vyhadzujú nízkoúrovňové metódy z vnútra Javy (bežný programátor ich nevyhadzuje), a predstavujú fatálne chyby, z ktorých je nemožné sa zotaviť. Príkladmi sú `OutOfMemoryError` -- indikácia vyčerpanej pamäte alebo `VirtualMachineError`, keď virtuálny stroj nevie púšťať Java programy.

Otázka, kedy použiť kontrolovanú a kedy nekontrolovanú výnimku, nie je jednoduchá. Sú vývojári, ktorí nepoužívajú kontrolované výnimky vôbec (iné OOP jazyky kontrolované výnimky nemajú). Nekontrolované výnimky majú tú nevýhodu, že zvádzajú k lenivosti. Dobrý programátor, keď hádže nekontrolované výnimky, uvádza ich, napriek tomu, že to nie je nutné, aj do dokumentácie aj do `throws`.

Výhodou nekontrolovaných výnimiek je, že ak sme si istí, že sme dodržali kontrakt a nenastanú výnimky identifikujúce nedodržanie kontraktu, nemusíme ich odchytávať v `try-catch` blokoch. Ak sa naviac vyhadzované nekontrolované výnimky uvedú do dokumentácie a do `throws`, nenastane situácia, že nejaká neočakávaná (nezdokumentovaná) výnimka vám vybuble z vnútra cudzieho kódu. Takéto situácie sú veľmi nepríjemné a spôsobujú často mnohé hodiny tápania a hľadania príčiny. Uvedenie v `throws` je to najmenej, čo môžeme urobiť, aj keby sme dokumentáciu ešte nemali dotiahnutú. Je to aspoň informácia o možnom probléme. Naviac vďaka `throws` vám Eclipse vie sám generovať `try-catch` bloky.

Výnimky pri prekrývaní metód

Pri kombinácii výnimiek a dedičnosti si treba dávať pozor. Platí totiž, že prekrývajúce metódy môžu vyhadzovať iba také výnimky ako ich náprotivky v rodičovských triedach. Inými slovami, ak chceme v oddedenej triede v nejakej metóde vyhadzovať viac výnimiek ako vyhadzuje metóda ktorú prekrývame, musíme rozšíriť zoznam výnimiek v `throws` pôvodnej rodičovskej metódy.

Napríklad ak metóda `dajUmiestnenie()` triedy `Film` vyhadzuje výnimku `FilmException`:

```
public class Film {
   public void dajUmiestnenie() throws FilmException {
      ...
   }
}
```

tak v metóde `dajUmiestnenie()` triedy `FilmNaDvd`, ktorá dedí od triedy `Film` môžeme vyhadzovať iba `FilmException`, jej potomkov alebo nič. Napríklad nasledujúci kód by nefungoval:

```
public class FilmNaDvd extends Film {
   public void dajUmiestnenie() throws FilmException, DvdException { // Exception
DvdException is not compatible with throws clause in Film.dajUmiestnenie()
      ...
   }
}
```

Možné opravy:

```
public class Film {
   public void dajUmiestnenie() throws FilmException,DvdException {
      ...
   }
}
```

Táto oprava je však pomerne nešťastná. Všeobecný `Film` by totiž nemal mať do činenia s výnimkami, ktoré zväzujú triedu `Film`u s jej potomkom DVD.

```
public class Film {
   public void dajUmiestnenie() throws Exception { //alebo ina spolocna nadtrieda tried
FilmException a DvdException
      ...
   }
}
```

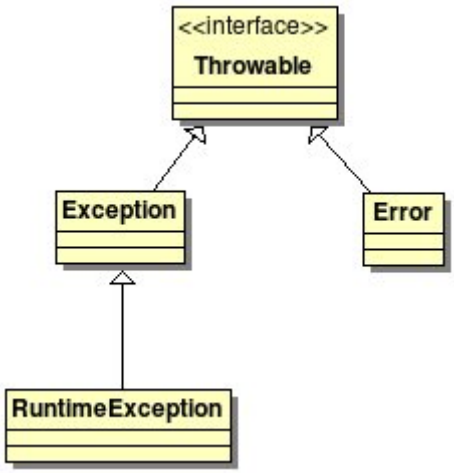
I v tomto prípade je oprava netradičná. Hádzanie všeobecnej výnimky `Exception` sme totiž vyššie v texte vyhlásili za nevhodné.

V tomto konkrétnom príklade je ideálne, keď `DVDException` oddedí od `FilmException`, a metóda `dajUmiestnenie()` bude môcť vyhodiť `DVDException` bez toho, aby narušila podmienku v rodičovi.

Odchytávanie viacerých výnimiek

Už vieme že odchytávanie výnimiek sa deje v `catch` blokoch. Tiež vieme, že `catch` blokov môže byť viac. Fungovanie viacerych blokov je založené na tom, že výnimky odchytí prvý `catch` blok, ktorý vie prijať vyhodenú výnimku. V každom `catch` bloku sa uvádza ako parameter nejaká premenná, ktorej typ určuje, či sa výnimka odchytí alebo nie.

Na to, aby sme vedeli, ktoré výnimky budú odchytené ktorým `catch` blokom, potrebujeme poznať hierarchiu dedenia vyhadzovateľných (`Throwable`) objektov.



Už vieme, že z premennej nadradenej triedy vieme referencovať ľubovoľný objekt potomka tejto triedy. Tento vzťah platí aj vo svete výnimiek. To znamená, že ak v prvom `catch` bloku odchytávame výnimku do premennej typu `Exception` tak odchytíme ľubovoľnú výnimku. Jediné čo by sa neodchytilo by boli chyby. Z toho vyplýva, že keby sme v ďalších `catch` blokoch odchytávali napríklad výnimku `FileNotFoundException`, ktorá je potomkom triedy `Exception` tak, by sme ju týmto catch blokom nikdy neodchytili, pretože by bolá odchytená prvým `catch` blokom s typom `Exception`. `catch` bloky teda uvádzame vždy od najkonkrétnejších po najvšeobecnejšie.

Príklad správneho poradia odchytávania:

```
try {
        ...
} catch (FileNotFoundException e) {
        System.out.println("Nenašiel som súbor");
} catch (IOException e) {
```

```
        System.out.println("Vstupno-výstupná chyba");
} catch (Exception e) {
        System.out.println("Nastala nejaká výnimka, napríklad aj ľubovoľná
RuntimeException");
}
```

Časté chyby pri používaní výnimiek

- Chyba č.1 - Odchytíme výnimku a nespravíme nič.

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

- Dôsledok chyby č.1 - Program obvykle spadne na úplne inom mieste a nik netuší prečo sa to stalo

- Chyba č.2 - Banality riešime výnimkami. V prípade, že sa dá výnimkam predchádzať overovaním jednoduchých podmienok, je vhodnejšie výnimkam zabrániť, ako ich potom odchytávať. V nasledujúcom príklade je príklad na zlé použitie výnimiek, kde neprechádzame pole cez for cyklus ale počkáme si pokiaľ nezačneme čítať neexistujúci prvok poľa.

```
try {
   int i = 0;
   while (true) {
      pole[i+1] = 2 * pole[i];
      i++;
   }
} catch (ArrayIndexOutOfBoundsException e) {}
```

- Dôsledok chyby č.2 - kód je veľmi ťažko čitateľný

- Chyba č.3 - nerobíme zmysluplné mená výnimiek, alebo proste vyhadzujeme rovno výnimku typu `Exception`.

```
void metóda() throws Exception {
   ...
}
```

- Dôsledok chyby č.3 - Používateľ metódy netuší, čo sa môže stať a nevie sa na to pripraviť. Aby bol schopný sa pripraviť, musí čítať cudzie zdrojáky.

- Chyba č.4 - Necháme vybublať výnimky príliš vysoko v zmysle hesla "nechce sa mi písať `try-catch` bloky, `throws` je rýchlejšie napísané"

```
void upečKoláč() throws IOException, SQLException {
   ...
}
```

- Dôsledok chyby č.4 - Výnimky sa majú odchytiť na mieste, kde sa s tým vieme vysporiadať. Vo vysokoúrovňových metódach netušíme, čo sa mohlo na nižších úrovniach pokaziť a ani nie sme v stave to riešiť. Ak sa s tým nevedia vysporiadať nižšie vrstvy, treba takéto nízkoúrovňové výnimky prebaliť do zrozumiteľnejších, aby bolo jasné, ako to z vysokej úrovne vyriešiť.

Statické metódy a premenné

Dôvodom, prečo to vieme písať takto je to, že metóda `sin()` je **statická**. Jej hlavička vyzerá nasledovne:

```
public static double sin(double a)
```

Všimnime si, že metóda je označená za statickú pomocou modifikátora `static`.

Na rozdiel od bežných metód, statické metódy nepotrebujú inštanciu triedy (teda objekt) na to, aby sme ich mohli zavolať. Vieme ich volať priamo na triede. Volajú sa preto aj **metódy triedy**.

Statické metódy síce narušujú základnú filozofiu OOP, v ktorej máme objekt, na ktorom voláme metódy, ale umožňujú prehľadnejším spôsobom volať pomocné metódy, pri ktorých nie je inštancia objektu potrebná.

Statické premenné

Popri statických metódach môže mať trieda aj statické inštančné premenné. Na rozdiel od klasických inštančných premenných, ktoré si udržiava každý objekt nezávisle, hodnota statickej inštančnej premennej je zdieľaná **všetkými** objektami danej triedy. Statickú inštančnú premennú možno chápať ako ,,kolektívnu pamäť" všetkých objektov daného typu. Ak hodnotu statickej premennej zmeníme v jednom objekte, zmena sa prejaví vo všetkých ostatných objektoch tejto triedy v rámci bežiaceho programu.

Majme napríklad statickú premennú `polomerDvd` v triede `FilmNaDvd`.

```
public class FilmNaDvd {
   static double polomerDvd;
   ...
}
```

Demonštrujme zmeny v nejakej testovanej triede:

```
public class TesterStatic {
   public static void main(String[] args) {
        FilmNaDvd matrix = new FilmNaDvd();
        matrix.polomerDvd = 5.5;
        FilmNaDvd shawshank = new FilmNaDvd();
        System.out.println(matrix.polomerDvd);
        System.out.println(shawshank.polomerDvd);
        shawshank.polomerDvd = 3.9;
        System.out.println(matrix.polomerDvd);
        System.out.println(shawshank.polomerDvd);
   }
}
```

Po spustení dostávame výpis:

```
5.5
5.5
3.9
3.9
```

Pokiaľ niekto zabudne, že ide o statickú premennú, toto chovanie môže byť mätúce. Aby sme predišli zmätku ničnetušiaceho čitateľa, je vhodnejšie namiesto zápisu `matrix.polomerDvd = 5.5;` použiť `FilmNaDvd.polomerDvd = 5.5;`. Takto je jasnejšie, že ide o inštančnú premennú spravovanú v triede a nie v jej objektoch. Samozrejme každý objekt má prístup ku všetkým statickým inštančným premenným.

Použitie statických inštančných premenných

Použitie statických inštančných premenných je nutné dobre zvážiť, pretože obvykle vedie ich použitie k zlému návrhu a mnohým ďalším problémom. V praxi existuje len niekoľko konkrétnych situácií, keď majú statické premenné svoj zmysel. V začiatkoch programovania v Jave možno povedať, že v jednoduchých programoch sa možno úplne vyvarovať statických inštančných premenných.

Konštanty

Jedno zo zmysluplných využití statických premenných spočíva v deklarovaní konštánt, teda logicky označených hodnôt, ktoré sa počas behu programu nemenia. Typickými príkladmi sú `Math.PI` a `Math.E` alebo farby v triede `Color`. My si v našom príklade môžeme zmeniť `polomerDvd` na konštantu, keďže zrejme všetky DVD budú mať rovnaký polomer. Ak by to tak nebolo, treba opustiť statický charakter tejto premennej a urobiť z nej obyčajnú inštančnú premennú.

Konštantu v Jave označujeme modifikátorom **final**, ktorý hovorí, že do danej premennej možno priradiť obsah len raz. Pri ďalšom pokuse nám to už Eclipse nedovolí.

```
public class FilmNaDvd {
   public static final double POLOMER_DVD = 6.0;
   ...
}
```

Je dobrým zvykom uvádzať konštanty veľkými písmenami. Ak ide o viacslovné pomenovanie, slová oddeľujeme podtržníkom.

Statické metódy a statické premenné

Statické metódy nevedia pristupovať k inštančným premennými svojej triedy. Je to logické -- keďže statické metódy vieme volať priamo na triedach (teda bez existencie objektov), nemôžeme pracovať s hodnotami, ktoré patria konkrétnym objektom. Pri pokuse o prístup k nestatickej inštančnej premennej zo statickej metódy nám to Eclipse nedovolí:

```
public class FilmNaDvd {
        private double String nazovFilmu;
        static void vypisNazov()
                System.out.println(nazovFilmu); //Cannot make a static reference to a
nonstatic field nazovFilmu
        }
}
```

Mnoho začínajúcich programátorov by sa pokúsilo vyriešiť tento problém zmenou inštančnej premennej `nazovFilmu` na statickú. (Ide o prvoplánové riešenie "daj pokoj, nepodčiarkuj, toto musí bežať. Keď chceš to mať statické tak si to maj.") Toto však vedie nielen k hroznému návrhu, ale zavádza do programu chyby. `FilmNaDvd` so statickou inštančnou premennou `nazovFilmu` by bol chybný: hodnota tejto premennej by sa zdieľala medzi **všetkými** objektami typu FilmNaDvd, čo by znamenalo, že všetky filmy na DVD by mali rovnaký názov.

Zmysluplné použitie statických metód je v pseudotriedach, ktoré sú iba akási zbierka užitočných metód a konštánt.

```
package sk.upjs.ics.znalosti.registracia;

import java.util.Arrays;
import java.util.Collection;
import java.util.List;

public abstract class Utils {
        public static final String ODDELOVAC = ";";

        /**
         * Rozseka retazec na polozky podla oddelovaca.
         * <p>
         * "AHOJ;SVET" => ["AHOJ", "SVET"]       *
         * @param retazec vstupny retazec
         * @return zoznam jednotlivych poloziek po rozdeleni
         */
        public static List<String> explode(String retazec) {
                if (retazec == null) {
                        throw new IllegalArgumentException("Retazec nemoze byt null!");
                }
                String[] zlozky = retazec.split(ODDELOVAC);
                return Arrays.asList(zlozky);
        }

        /**
         * Zluci kolekciu poloziek a medzi kazde dve polozky vlozi oddelovac.
         * <p>
         * ["AHOJ", "SVET"]     => "AHOJ;SVET"
         * @param retazec vstupny retazec
         * @return vysledny zluceny retazec.
         */

        public static String implode(Collection<String> polozky) {
                StringBuilder stringBuilder = new StringBuilder();
                for (String polozka : polozky) {
                        stringBuilder.append(polozka);
                        stringBuilder.append(ODDELOVAC);
                }
                if (stringBuilder.length() > 0) {
                        stringBuilder.deleteCharAt(stringBuilder.length() - 1);
                }
                return stringBuilder.toString();
        }
}
```

Doteraz sme používali najmä statické metódy z tried `java.util.Collections`, `java.util.Arrays`, `java.lang.Math` alebo `java.lang.System`.