Príručka Java

Java SE API a OOP

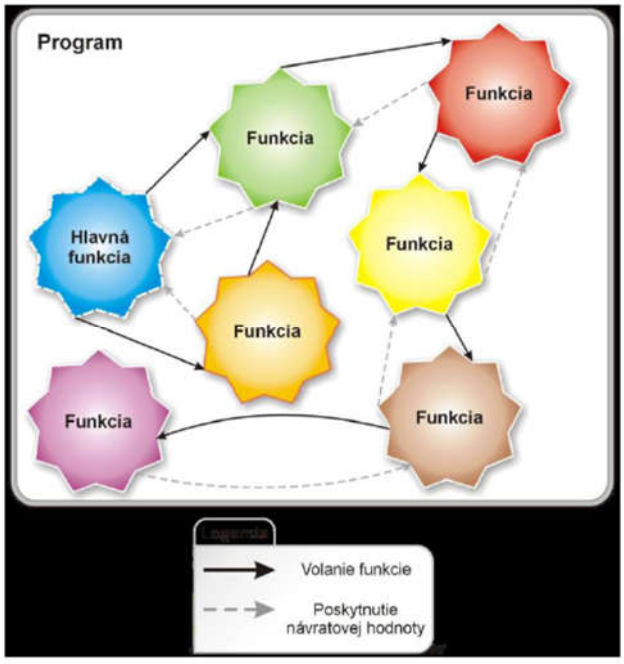
IT Academy s.r.o., Tomášikova 50/A, 831 04 Bratislava, tel.: 0917/095 406, 0907/375 543
IČO: 46 759 786, DIČ: 2023556766, IČ DPH: SK2023556766

Obsah:

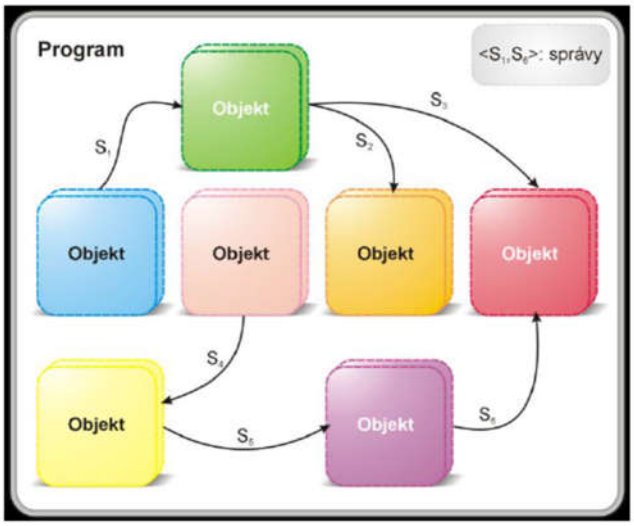
Túto príručku môžete využiť ako pomôcku pri práci s programom Java. **Príručka podlieha autorským právam a jej vlastníkom je spoločnosť IT Academy s.r.o.**

I. Zoznámenie sa s triedami, objektmi a metódami

Porovnanie štruktúrovaného a objektovo orientovaného programovania



Obr. 1 Štruktúrovaný program



Obr. 2 Objektovo orientovaný program

Koncepty objektovo orientovaného programovania

- identita objektu
- zachovanie stavu
- zapuzdrenie
- skrývanie informácii
- objekty navzájom interagujú
- niektoré objekty obsahujú iné objekty
- dedenie
- polymorfizmus (mnohotvárnosť)

Objekt (object)

Objekt je entita ktorá má:

- **stav** (uložený v atribútoch/premenných)
- **správanie** (definované metódami/funkciami/operáciami /správami)
- **identitu**

Objektovo orientovaný program sa vykonáva **vzájomnou interakciou objektov**. Objekty komunikujú volaním funkcii = **volaním metód** = posielaním správ.

Príklad:

lampa
- stav: zapnutá/vypnutá
- metódy: zapni, vypni

rádio
- stav: zapnuté/vypnuté, hlasitosť, frekvencia
- metódy: zapni, vypni, zmeň hlasitosť, zmeň frekvenciu, automaticky vyhľadaj stanicu

Trieda (class)

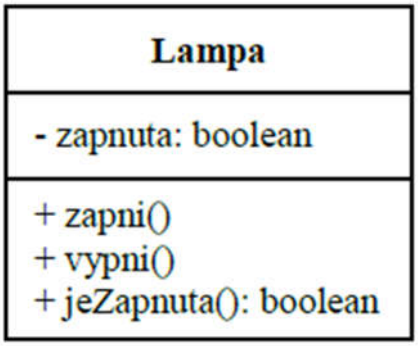
Správanie objektu definuje jeho typ. Typ objektu sa označuje ako trieda.

Trieda definuje:

- **atribúty**
- **metódy (funkcie)** – správanie objektu

Po zadefinovaní triedy môžeme vytvoriť ľubovoľný počet objektov. Tieto objekty sú **inštanciami triedy**, podľa ktorej vznikli.

Príklad: Definícia triedy v jazyku java a uml:



Obr. 3

```
public class Lampa {
    private boolean zapnuta;

    public void zapni() {
        // kod funkcie
    }

    public void vypni() {
        // kod funkcie
    }

    public boolean jeZapnuta() {
        return zapnuta;
    }
}
```

Vytvorenie objektu v jazyku Java:

```
Lampa mojaLampa; // objekt mojaLampa je inštanciou triedy Lampa
Lampa susedovaLampa; // objekt susedovaLampa je tiež inštanciou triedy Lampa
Radio mojeRadio; // objekt mojeRadio je inštanciou triedy Radio
```

Obr. 4 Vytvorenie objektu v jazyku Java

Identita objektu (identity object)

Identita objektu je vlastnosť, podľa ktorej možno každý objekt identifikovať a pracovať s ním ako zo samostatnou **softvérovou entitou**. V Jave je objekt identifikovaný referenciou. V iných jazykoch môže byť identifikovaný adresou.

Zachovanie stavu

Napríklad vo funkcionálnom alebo procedurálnom programovaní, si funkcia (procedúra) medzi volaniami neuchováva stav (každé volanie funkcie je nezávislé od predchádzajúceho volania). Objekty si počas svojej existencie uchovávajú stav. Môžu sa riadiť podľa svojej minulosti. Stav je uchovávaný pomocou **atribútov**.

Zapuzdrenie (encapsulation)

Zapuzdrenie je **zoskupenie súvisiacich ideí** do jednej jednotky, na ktorý sa možno odkazovať jediným názvom. V OOP je to zabalenie operácii a atribútov do jedného typu objektu, ktorý poskytuje rozhranie cez ktoré môže komunikovať s inými objektmi.

Zapuzdrenie zahrňuje aj **skrývanie informácii a implementácii**. Skrývanie informácii znamená, že informácie v objekte nemožno zistiť z vonku (z miesta mimo objektu). Skrývanie implementácie znamená, že podrobnosti implementácie nie sú z vonku prístupné.

Príkladom skrytia implementácie môže byť trieda **Obdĺžnik**, ktorá poskytuje metódu obsah(), vracajúcu obsah obdĺžnika. Používateľ tejto triedy ale napríklad nevie kedy dochádza k výpočtu obsahu (obsah sa môže vypočítať pri zmene dĺžok strán a uložiť od atribútu, alebo sa jeho hodnota neukladá do atribútu, ale sa vypočíta vždy pri volaní funkcie obsah()).

Súčasťou dobrého zapuzdrenia je povolenie prístupu k atribútom iba cez **funkcie** (väčšia bezpečnosť a ľahšia modifikovateľnosť implementácie).

Interakcia objektov

Objekty spolupracujú pomocou zasielania správ = volanie metód

Obr .5 Interakcia objektov

II. Asociácia celku a časti

Dve najpoužívanejšie asociácie celku a časti sú kompozícia a agregácia.

Kompozícia (zloženie)

- Zložený objekt (kompozícia) **neexistuje** bez svojich častí (zložiek, komponentov)
- V každom okamihu môže byť akýkoľvek komponent súčasťou iba **jednej** kompozície
- Kompozícia je väčšinou **heterometrická** (komponenty sú väčšinou rôznych typov)

Napríklad lietadlo sa sklad z trupu a krídiel.

Agregácia (zoskupenie)

Zoskupený objekt (agregácia) môže potenciálne existovať bez svojich častí (tvoriacich/ konštitučných objektov). Napr. oddelenie môže existovať aj bez zamestnancov (táto vlastnosť nemusí byť vždy užitočná).

- **Jeden** objekt môže byť súčasťou **viacerých** zoskupení.
- Agregácia má tendenciu byť **homeometrická** (tvoriace objekty sú väčšinou toho istého typu).

Napríklad les je agregáciou stromov, stádo zoskupením oviec.

Polymorfizmus (polymorphism, mnohotvárnosť)

Polymorfizmus umožňuje pracovať z rôznymi typmi objektov rovnakým spôsobom. Prejavuje sa viacerými spôsobmi.

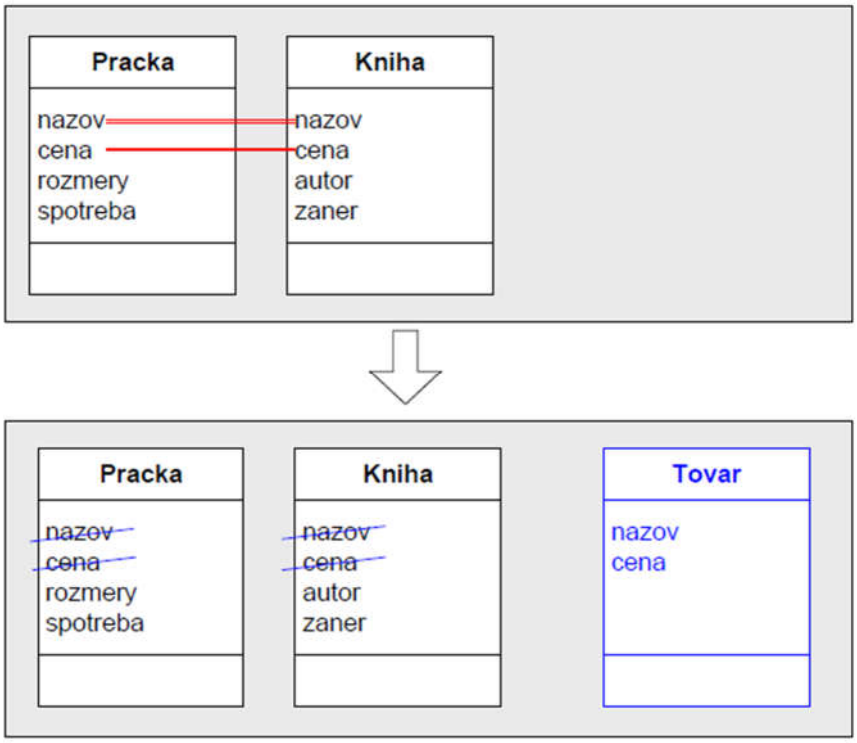
Napríklad:

1. **dedičnosť** (inheritance) (obr.č. 1)
    - **prekrývanie funkcii** (overriding)
    - **preťaženie funkcii** (overloading)
    - objekt typu podtriedy môžeme použiť všade tam, kde možno použiť aj objekt typu nadtriedy (pretypovanie typu podtriedy na typ nadtriedy)
    - **dynamická väzba** (**virtuálne funkcie**)
2. **preťaženie funkcii** (overloading)
3. **generické typy** (šablóny)

Ďalšie pojmy

Abstrakcia – možnosť abstrahovať od niektorých detailov. Na použitie objektu nepotrebujeme poznať jeho implementáciu. Stačí poznať jeho rozhranie (z vonku prístupné metódy a atribúty).

Delegovanie – ak požiadame objekt o službu a on požiada o túto službu iný objekt.



Obr. 6 Dedičnosť

III. Konštruktory

Konštruktor je **funkcia**, ktorá sa automatický volá pri vytváraní objektu.

Konštruktor slúži pre **inicializáciu objektu.**

Deklarácia konštruktora je podobná ako deklarácia bežnej funkcie, ale neobsahuje návratový typ (konštruktor nevracia hodnotu) a názov je rovnaký ako názov triedy.

Príklad:

```
public class Obdlznik {
    private String nazov;
    private int sirka;
    private int vyska;

    public Obdlznik(String pocNazov, int pocSirka,
int pocVyska) {
        this.nazov = pocNazov;
        this.sirka = pocSirka;
        this.vyska = pocVyska;
    }

    public void tlacInfo() {
        System.out.println(this.nazov + ": "
                + this.sirka + ", " + this.vyska);
    }
```

```
    public static void main(String[] args) {
        Obdlznik obdl = new Obdlznik("moj
    obdlznik",11,22);
        obdl.tlacInfo(); // vytlaci: moj obdlznik:
    11, 22
    }
}
```

Kľúčové slovo **new** je operátor jazyka **Java**. Tento operátor vyhradí v pamäti miesto pre objekt, zavolá konštruktor objektu, ktorý je uvedený za slovom **new** a vráti referenciu na novovytvorený objekt.

Príkaz **Obdlznik obdl** = new Obdlznik("moj obdlznik",11,22); vytvorí v pamäti miesto pre inštanciu triedy **Obdlznik**, zavolá uvedený konštruktor a referenciu na novovytvorenú inštanciu nakopíruje do deklarovanej premennej obdl.

Kľúčové slovo v hore uvedenom Príklade označuje práve ten objekt, nad ktorým sa vykonáva funkcia.

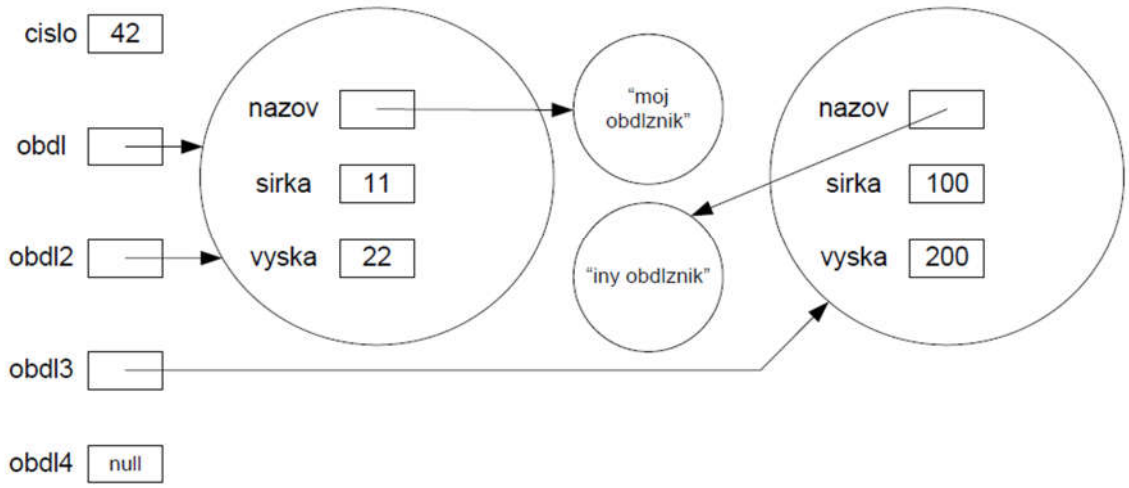
Referenčný typ premennej

Ak je typom premennej základný typ (**int, double**, ...) , potom premenná obsahuje priamo
hodnotu daného typu.

Ak je typom premennej trieda, potom premenná môže obsahovať iba referenciu na objekt (neobsahuje samotný objekt). Ak premenná neobsahuje referenciu na objekt, tak je hodnota referencie **null**. Referenciu na jeden objekt môže obsahovať viacero premenných.

Príklad:

```
int cislo = 42;
Obdlznik obdl = new Obdlznik("moj obdlznik", 11, 22);
Obdlznik obdl2 = obdl;
Obdlznik obdl3 = new Obdlznik("iny obdlznik", 100, 200)
Obdlznik obdl4 = null;
```



Obr. 7 Referenčný typ premennej

Prístup k atribútom a metódam objektu

Používa sa operátor . (bodka) v tvare:

- **z vnútra objektu:**

```
this.nazovPremennej // this je možné vynechať
alebo
this.nazovFunkcie(parametre) // this je možné vynechať
```

- **mimo objektu:**

referenciaNaObjekt.nazovPremennej
alebo
referenciaNaObjekt.nazovFunkcie(parametre)

Implicitný (počiatočný) konštruktor

Ak v triede nie je explicitne definovaný žiadny konštruktor, prekladač automaticky vytvorí implicitný konštruktor bez parametrov.

Poznámka: Implicitný konštruktor vyvolá konštruktor nadtriedy bez parametrov. Ak nadtrieda neobsahuje takýto konštruktor, tak vznikne chyba pri preklade (toto súvisí s dedičnosťou, ktorou sa budeme zaoberať neskôr).

Príklad: (implicitný konštruktor):

```
public class Obdlznik {
    private String nazov;
    private int sirka;
    private int vyska;

    public void tlacInfo(){
        System.out.println(this.nazov + ": "
                    +   this.sirka   +   ",   "   +
                this.vyska);
    }

    public static void main(String[] args) {
        Obdlznik o1 = new Obdlznik();
        o1.tlacInfo(); // vytlaci: null, 0, 0
    }
```

Kompilátor priradil neinicializovaným členským premenným nulové hodnoty.

Preťažovanie konštruktorov

Java umožňuje preťažovať aj konštruktory. **Mechanizmus** je podobný ako pri obyčajných funkciách t.j. metódach.

Príklad:

```
public class Obdlznik {
    private String nazov;
    private int sirka;
    private int vyska;
    public Obdlznik() {
        this(0,0);
    }

    public Obdlznik(int pocSirka, int pocVyska) {
        this("",pocSirka,pocVyska);
    }

    public  Obdlznik(String  pocNazov,int  pocSirka,
int pocVyska) {
        this.nazov = pocNazov;
        this.sirka = pocSirka;
        this.vyska = pocVyska;
    }
```

```
public void tlacInfo() {
    System.out.println(this.nazov + ": "
                + this.sirka + ", " + this.vyska);
    }

    public static void main(String[] args) {
        Obdlznik o1 = new Obdlznik();
        o1.tlacInfo(); // vytlaci: : 0, 0
        Obdlznik o2 = new Obdlznik(10,20);
        o2.tlacInfo(); // vytlaci: : 10, 20
        Obdlznik o3 = new Obdlznik("treti",11,22);
        o3.tlacInfo(); // vytlaci: treti: 11, 22
    }
}
```

Kľúčové slovo **this** je v tomto príklade použité aj na vyvolanie konštruktora.

Predávanie vstupných argumentov

Argumenty základného typu (napr.: int, double) aj referenčného typu (napr. objekty) sa predávajú hodnotou.

Parameter základného typu:

Do funkcie sa predáva kópia hodnoty parametra. Zmeny hodnoty parametra vykonané vo funkcii sa preto mimo funkcie neprejavia.

Parameter referenčného typu:

Do funkcie sa predáva kópia referencie na objekt. To znamená že zmeny hodnôt v objekte zostávajú aj po ukončení funkcie. Zmeny samotnej referencie vykonané vo funkcii sa mimo funkcie neprejavia.

Príklad:

```
public class PredavanieParametrov {
    static class Udaje {
        public int atribut;
    }

    public static void vytlacInkrement(int x) {
        x++;
        System.out.println(x);
    }
    private static void zmenUdaj(Udaje udaj) {
        udaj.atribut = 20;
        udaj = new Udaje();
        udaj.atribut = 30;
    }

    public static void main(String[] args) {
        int x = 0;
        vytlacInkrement(x);             // vytlaci: 1
        System.out.println(x);          // vytlaci: 0
        Udaje udaj = new Udaje();
        udaj.atribut = 10;
        System.out.println(udaj.atribut);          //
    vytlaci: 10
        zmenUdaj(udaj);
        System.out.println(udaj.atribut);          //
    vytlaci: 20
    }
}
```

Automatická správa pamäti (garbage collector)

Nové objekty sa vytvárajú pomocou operátora new. Odstraňovanie nepotrebných objektov je v Jave automatické (menšia pracnosť a náchylnosť voči chybám).

Automatická správa pamäti pravidelne uvoľňuje nepoužívané objekty z pamäte, v intervaloch nezávislých na programátorovi. Objekt možno odstrániť ak naň neobsahuje referenciu žiadna premenná.

Neskôr si uvedieme:
metóda finalize() – metóda volaná pri odstraňovaní objektu
System.runFinalization() // vynútenie spustenie finalizéru

Príklad:

```
String ret1 = new String(”oblak”);
String ret2 = ”ahoj”;
String ret3 = ”stolicka”;
String ret4 = ret3;
ret1 = null;              //    ”oblak”    môže    byť
odstránený
re2  =  new  String(”cau”);      //  ”ahoj”  môže  byť
odstránený
ret3 = null;         // na ”stolicka” ešte obsahuje
referenciu ret4
ret4 = null;              //   ”stolicka”   môže   byť
odstránený

void nazovFunkcie() {
    ret = ”mesto”;
}   // ak na ”mesto” už neodkazuje žiadna premenná,
    // tak ho po skončení funkcie možno odstrániť
```

Použitie kľúčového slova this

Prístup k členskej premennej, alebo členskej funkcii.

V rámci členskej metódy, alebo konštruktora predstavuje kľúčové slovo odkaz na objekt, nad ktorým sa metóda, alebo konštruktor práve vykonáva. Pomocou **this** možno vo vnútri členskej metódy, alebo konštruktora **odkazovať** na ľubovoľnú členskú premennú, alebo členskú funkciu. V tomto prípade možno slovo **this** väčšinou vynechať.

Príklad:

```
public class Bod {
    private int x;
    private int y;

    // Tri verzie nastavenia

    public void nastav1(int noveX, int noveY) {
        this.x = noveX;
        this.y = noveY;
    }

    // to iste ako nastav1, ale môžeme vynechať this
    public void nastav2(int noveX, int noveY) {
        x = noveX;
        y = noveY;
    }
/*
```

```
Parametre funkcie majú rovnaký názov ako členské
premenné.
Vo funkcii bude x označovať vstupný parameter x,
nie členskú
premennú x (podobne y). Preto pre označenie
členskej premennej
musíme použiť kľúčové slovo this.
*/
    public void nastav3(int x, int y) {
        this.x = x
        this.y = y;
    }
}
```

IV. Bližší pohľad na metódy a triedy

Volanie konštruktora

V konštruktore je možné pomocou kľúčového slova vyvolať iný konštruktor tej istej triedy. Ak konštruktor obsahuje volanie iného konštruktora, musí byť toto volanie uvedené na jeho prvom riadku.

Príklad:

```
public class Bod {
    private String nazov;
    private int x;
    private int y;

    public Bod() {
        this("",0,0); // volanie konštruktora musí
    byť prvé
                  // tu môže byť ďalší kód
    }

    public Bod(String nazov) {
        this(nazov,0,0);  //  volanie  konštruktora
    musí byť prvé
                    // tu môže byť ďalší kód
    }

    public Bod(int x, int y) {
        this("",x,y); // volanie konštruktora musí
    byť prvé
                  // tu môže byť ďalší kód
    }
```

```
    public Bod(String nazov, int x, int y) {
        this.nazov = nazov;
        this.x = x;
        this.y = y;
    }
}
```

Riadenie prístupových práv (viditeľnosti) pomocou modifikátorov

Trieda, členská premenná, alebo funkcia je v danej oblasti:

- **viditeľná** = prístupná = môžeme s ňou v danej oblasti priamo pracovať alebo
- **neviditeľná** = neprístupná = nemôžeme s ňou v danej oblasti pracovať

Táto časť súvisí s dedičnosťou a balíkmi, ktorými sa budeme zaoberať neskôr. Balík je **zoskupenie tried.**

Riadenie prístupových práv k triede

Ak je pred deklaráciu triedy uvedený modifikátor **public**, tak je daná trieda viditeľná z ľubovoľného miesta (môžeme ju používať na ľubovoľnom mieste). Ak pred deklaráciou triedy nie je uvedený žiadny modifikátor, tak je trieda viditeľná iba v rámci vlastného balíka. Toto východzie nastavenie bez modifikátora sa označuje ako **package-private**.

Príklad:

```
// trieda viditeľná všade
public class Trieda1 {
    // .....
}

// trieda viditeľná iba v rámci balíku (package-private)
class Trieda2 {
    // .....
}
```

Riadenie prístupových práv k členom triedy

- **modifikátor public** – člen triedy je prístupný na ľubovoľnom mieste
- **modifikátor protected** – člen triedy je dostupný v celom svojom balíku a tiež v podtriede, ktorá môže byť aj v inom balíku
- **bez modifikátora (package-private)** – člen triedy je dostupný v celom svojom balíku
- **modifikátor private** – člen je prístupný iba vo vlastnej triede

Modifikátor	V rámci triedy	V inej triede toho istého balíka	V podtriede, ktorá je v inom balíku	V inej triede iného balíka
public	ÁNO	ÁNO	ÁNO	ÁNO
protected	ÁNO	ÁNO	ANO	NIE
bez modifikátoru	ÁNO	ÁNO	NIE	NIE
private	ÁNO	NIE	NIE	NIE

Tab. 1 Tabuľka prístupových práv

Odporučenia:

Treba nastavovať čo najnižšie prístupové práva. Nepoužívajte členské premenné s modifikátorom **public** (možno okrem konštánt). Hlavnou výhodou nastavenia čo najnižších prístupových práv je jednoduchšia možnosť meniť implementáciu v prípade potreby.

Členské premenné a funkcie triedy

Členské premenné a funkcie objektu

- každý objekt má svoje vlastné premenné
- funkcie sa vykonávajú nad objektom

Členské premenná a funkcie triedy

- každá premenná existuje práve **raz** (patrí triede, resp. všetkým objektom). Takéto premenné existujú aj v prípade že nie je vytvorený žiadny objekt.
- funkcie sa vykonávajú **nad triedou**. (Pre zavolanie funkcie nie je potrebné, aby existoval objekt).

Členské premenné a funkcie triedy sa označujú slovom **static**. Práve preto sa niekedy nazývajú statické.

Na statické premenné a **funkcie triedy** sa môžeme odkazovať pomocou triedy, alebo pomocou objektu. Odkazovanie pomocou objektu sa neodporúča, pretože zo zápisu nie je jasné, že je to premenná resp. funkcia triedy.

Príklad:

```
Trieda.premenná // odporúčaný spôsob
objekt.premenná // neodporúča sa
Trieda.funkcia() // odporúčaný spôsob
objekt.funkcia() // neodporúča sa
```

Obmedzenia vo funkciách triedy:

- **Funkcie objektu** môžu priamo pristupovať k premenným a funkciám triedy.
- **Funkcie triedy** môžu priamo pristupovať k premenným a funkciám triedy.
- Funkcie triedy **nemôžu** priamo pristupovať k premenným a funkciám objektu, pretože sa funkcia vykonáva nad triedou.

  Pre prístup k premennej alebo funkcii objektu, musia použiť referenciu na objekt. Nemožno použiť slovo na prístup k premenným a funkciám objektu.

Príklad:

```
public class Vyrobok {
    private static int pocetVyrobkov = 0;
    private int id;

    public static int pocetVsetkych() {
        return pocetVyrobkov;
    }
```

```
    public Vyrobok() {
        id = ++pocetVyrobkov;
    }

    public int idVyrobku() {
        return id;
    }

    public static void dalsiaFunkcia(Vyrobok v) {
        // int a = id; <- CHYBA
        // int b = this.id; <- CHYBA
        int c = v.id; // OK
    }

    public static void main(String[] args) {
        Vyrobok v1 = new Vyrobok();
        Vyrobok v2 = new Vyrobok();
        System.out.println(Vyrobok.pocetVsetkych())
    ; // 2
        System.out.println(v1.idVyrobku());
    // 1
        System.out.println(v2.idVyrobku());
    // 2
        Vyrobok v3 = new Vyrobok();
        System.out.println(v3.idVyrobku());
    // 3
        System.out.println(Vyrobok.pocetVsetkych())
    ; // 3
    }
}
```

Napríklad trieda **java.lang.Math** obsahuje funkcie pre matematické výpočty (napr.: sínus, mocnina, maximum, logaritmus). Jej funkcie sú statické, vďaka čomu nemusíme vytvárať jej inštancie pri výpočtoch.

Príklad:

```
int a = Math.max(2,4);
double b = Math.sin(0.1);
```

V. Konštanty

Pre definíciu konštanty slúži kľúčové slovo final.

Príklad:

```
public void funkcia() {
    final int MAX = 10; // priradenie pri deklarácii
    final int MIN;
    MIN = -10;      // priradenie mimo deklarácie
        // MIN = -20; <- CHYBA konštanta už má definovanú hodnotu
}
```

Konštanta sa často definuje kombináciou modifikátorov static a final.

Príklad:

```
public class Kalkulacka {
    private static final double PI = 3.14159265358979323846;
    // .....
}
```

Ak je konštanta základného typu, alebo reťazec a jej hodnota je v dobe prekladu známa,
nahradí prekladač na všetkých miestach v kóde názov konštanty jej hodnotou.

Inicializácia členských premenných

Členské premenné (statické aj nestatické) možno inicializovať pri deklarácii. Ak inicializáciania je jednoduchá (napr. naplnenie pola môže vyžadovať cyklus), potom môžeme využiť ďalšie konštrukcie.

Inicializácia statických premenných

- **inicializácia pri deklarácii**
- **statické inicializačné bloky** - Statických inicializačných blokov môže byť viacero. Statické inicializačné bloky sa vykonajú v poradí, v ktorom sú napísané.
- **súkromné statické metódy**

Inicializácia inštančných premenných

- **konštruktor**
- **inicializácia pri deklarácii**
- **inicializačné bloky**

Kompilátor skopíruje inicializačné bloky do všetkých konštruktorov **finálne metódy** (metódy, ktoré nemožno v podtriedach meniť).

Príklad:

```
public class Inicializacia {
    private static int statickaPremenna1 = 10;
    private static int statickaPremenna2;
    private static int statickaPremenna3;
    private static int statickaPremenna4;
    private   static   int   statickaPremenna5   =
statickaInicialFukcia();
    private int instancnaPremenna1 = 100;
    private int instancnaPremenna2;
    private int instancnaPremenna3;
    private int instancnaPremenna4 =
    finalnaInicializacnaFunkcia();
    private int instancnaPremenna5;
    private int instancnaPremenna6;
    static { // statický inicializačný blok
        statickaPremenna2 = 20;
        statickaPremenna3 = 30;
    }
    static { // statický inicializačný blok
        statickaPremenna4 = 40;
    }

    private static int statickaInicialFukcia() {
        return 50;
    }

    protected final int
```

```
finalnaInicializacnaFunkcia() {
        return 400;
    }

    { // inicializačný blok pre nestatické premenné
        instancnaPremenna2 = 200;
    }

    public Inicializacia(){
        instancnaPremenna3 = 300;
    }

    { // inicializačný blok pre nestatické premenné
        instancnaPremenna5 = 500;
        instancnaPremenna6 = 600;
    }
}
```

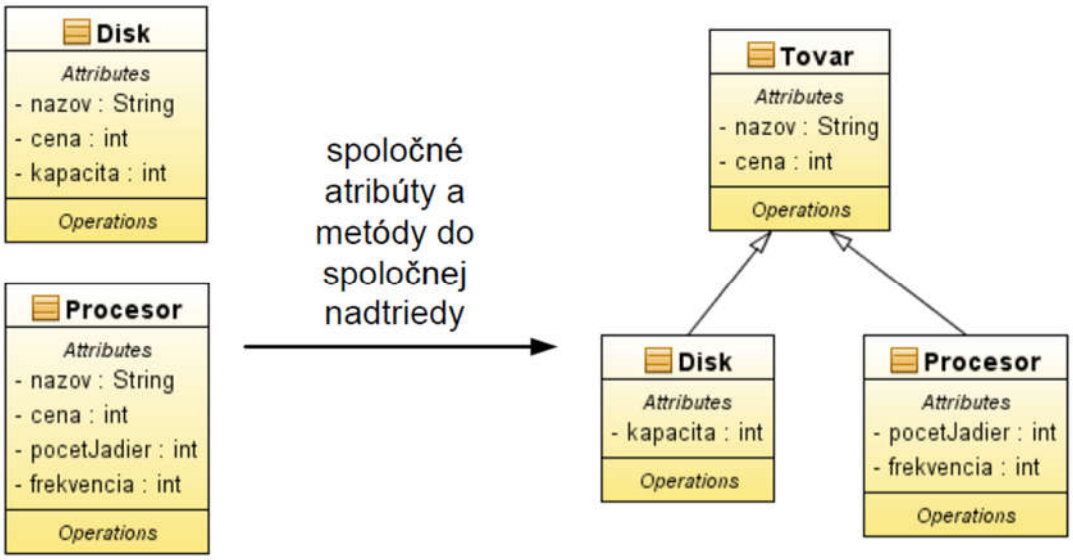
VI. Dedičnosť (inheritance)

Použitie: Ak chceme vytvoriť novú triedu a pritom existuje iná trieda, ktorá obsahuje časť potrebného kódu, môžeme svoju novú triedu odvodiť od existujúcej triedy.

Trieda, ktorá je odvodená od inej triedy sa nazýva podtrieda, odvodená trieda, rozšírená trieda, potomok, alebo podriadená trieda. Trieda od ktorej sa podtrieda odvodzuje, sa nazýva **nadtrieda**, základná trieda, rodič, alebo nadriadená trieda.

Pri tvorbe novej triedy je uvedené meno nadtriedy za kľúčovým slovom **extends**.

Príklad dedičnosti



Obr. 8 Dedičnosť

Nadtrieda Tovar:

```
public class Tovar {
    private String nazov;
    private int cena;

    public Tovar(String nazov, int cena) {
        this.nazov = nazov;
        this.cena = cena;
    }

    public int getCena() {
        return cena;
    }

    public void setCena(int cena) {
        this.cena = cena;
    }

    public String getNazov() {
        return nazov;
    }

    public void setNazov(String nazov) {
        this.nazov = nazov;
    }
}
```

podtrieda Disk:

```
public class Disk extends Tovar {
    private int kapacita; // v MB

    public Disk(String nazov, int cena, int
kapacita) {
        super(nazov, cena);
        this.kapacita = kapacita;
    }

    public int getKapacita() {
        return kapacita;
    }

    public void setKapacita(int kapacita) {
        this.kapacita = kapacita;
    }

}
```

podtrieda Procesor:

```
public class Procesor extends Tovar {
    private int frekvencia;
    private int pocetJadier;

    public Procesor(String nazov, int cena,
                int frekvencia, int pocetJadier) {
        super(nazov, cena);
        this.frekvencia = frekvencia;
        this.pocetJadier = pocetJadier;
    }

    public int getFrekvencia() {
        return frekvencia;
    }

    public void setFrekvencia(int frekvencia) {
        this.frekvencia = frekvencia;
    }

    public int getPocetJadier() {
        return pocetJadier;
    }

    public void setPocetJadier(int pocetJadier) {
        this.pocetJadier = pocetJadier;
    }
}
```

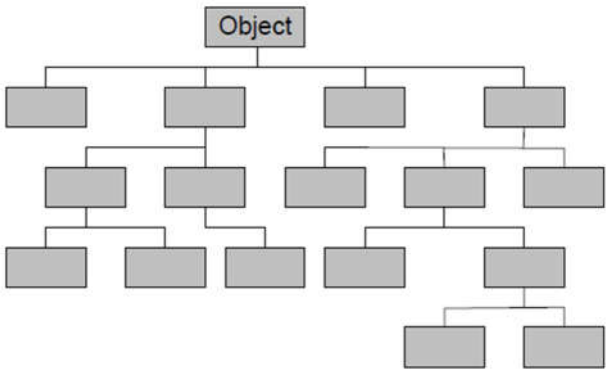
Kľúčové slovo **super** sa používa na volanie konštruktora nadtriedy (podobne ako **this** na volanie konštruktora tej istej triedy).

Podtrieda dedí (v podtriede možno používať) všetky členy (premenné, metódy, vnorené typy) typu **public a protected** bez ohľadu na balík v ktorom sa nachádzajú. Ak je podtrieda
súčasťou rovnakého balíku ako nadtrieda, dedí aj členy označené ako **private-package**
(označenie bez modifikátora).

Konštruktory nepatria medzi členov, takže ich podtrieda nededí. Je však možné z podtriedy
vyvolať konštruktor nadtriedy. Každá trieda v Jave (okrem triedy Object) dedí od niektorej inej triedy. Každá trieda (okrem Object) dedí práve od jednej triedy. Java **neumožňuje** viacnásobnú dedičnosť.

Ak pri vytváraní novej triedy neuvedieme jej nadtriedu, tak je automaticky nová trieda odvodená z triedy **Object**.

Táto trieda je umiestnená **najvyššie** v hierarchii dedičnosti tried v jave.



Obr. 9 Object

Trieda Object definuje a implementuje spoločné chovanie všetkým triedam. Je definovaná
v balíku **java.lang**.

Súkromné členy v nadtriede

Podtrieda **nededí** členy nadtriedy, ktoré sú označené ako **private**.

Tiež nededí členy nadtriedy **bez modifikátora** (označované ako private-package), ak sú nadtrieda a podtrieda v rôznych balíkoch.

Ak má napríklad nadtrieda metódy označené ako **public**, alebo **protected**, ktoré pracujú s jej členmi označenými ako private, potom možno pomocou týchto metód nepriamo pristupovať k nezdedeným členom. Podobne môžu pristupovať k nezdedeným členom aj cez zdedené vnorené triedy.

VII. Pretypovanie objektov (casting, type casting)

Pri prístupe k členom objektu zapisujeme

- **názov premennej,** ktorá obsahuje referenciu na objekt,
- **bodku**
- **názov člena objektu**.

Kompilátor podľa typu premennej dokáže skontrolovať, či daný objekt obsahuje zadaný člen,
alebo nie.

Mechanizmus dedenia zabezpečuje, že podtrieda obsahuje okrem členov v jej definícii, aj
členy nadtriedy.

Implicitné pretypovanie

Majme triedu **T**. Premenná, ktorá je referenčnou premennou typu **T**, môže obsahovať nie len referencie na inštancie triedy **T**, ale aj na inštancie všetkých (priamych aj nepriamych) podtried triedy **T**.

Vráťme sa k príkladu **Tovar, Disk, Proc**esor.

Nech kód programu obsahuje:

```
Disk disk = new Disk(“nazov”,45,500);
Tovar tovarDisk = disk; // implicitné pretypovanie
```

Uvedený zápis neobsahuje chybu, pretože premenná **tovarDisk**, môže obsahovať referenciu nie len na inštanciu triedy Tovar, ale aj na inštancie jej podtried.

Zápis:`Tovar tovarDisk = disk;`

obsahuje pretypovanie, ktoré sa označuje ako **implicitné pretypovanie.**

Trieda **Tovar** aj všetky jej podtriedy obsahujú metódu **getNazov()**, preto je nasledujúce volanie v poriadku.

```
tovarDisk.getNazov();
```

Trieda Tovar ale neobsahuje metódu **getKapacita()**, preto kompilátor vyhodnotí nasledujúce volanie ako chybné, aj napriek tomu že premenná bude za behu referenciou na objekt ktorý metódu getKapacita() obsahuje:

```
tovarDisk.getKapacita(); // CHYBA
```

Kompilátor vo všeobecnosti nemôže vždy overiť, aký typ objektu bude za behu referencovaný danou premennou.

Explicitné pretypovanie

Majme **triedu T**. Referenčná premenná typu T nemôže vo všeobecnosti obsahovať referencie na inštancie nadtried typu T.

```
Zápis Tovar tovar = new Tovar(„tovar“, 10);
Procesor procesor = tovar; // CHYBA
```

je preto chybný. Kompilátor by počas kompilácie nedokázal kontrolovať volania metód nad objektmi referencovanými premennou procesor.

Zápis:

```
Tovar tovarProcesor = new Procesor("meno",100,2800,2);
Procesor proc2 = (Procesor)tovarProcesor; // explicitne pretypovanie
```

je v poriadku.

Jeho druhý riadok obsahuje **explicitné pretypovanie**. Pri explicitnom pretypovaní kompilátor vloží do programu kontrolu, ktorá za behu programu skontroluje **korektnosť pretypovania** (či premenná tovarProcesor obsahuje inštanciu triedy Procesor). V prípade chyby kontrola vyhodí **výnimku** (výnimkami sa budeme zaoberať neskôr).

Zápis:

```
proc2.getNazov();      // zdedená metóda
proc2.getFrekvencia(); // metóda definovaná v triede Procesor
```

je OK.

Ak bude v kóde programu zápis:

```
Procesor proc = (Procesor)tovar // CHYBA ZA BEHU
```

nevznikne chyba pri kompilácii, ale vznikne chyba za behu programu.

VIII. Operátor instanceof

Testovanie typu objektu umožňuje **operátor instanceof**.

V nadväznosti na predchádzajúci príklad:

```
public static void main(String[] args) {
    Disk disk = new Disk("nazov",45,500);
    Tovar tovarDisk = disk; // implicitné
    pretypovanie Tovar tovar = new Tovar("tovar",
    10);
    Tovar tovarProcesor = new
Procesor("meno",100,2800,2);
    Procesor proc2 = (Procesor)tovarProcesor; //
    explicitne pret.
    System.out.println(disk instanceof Disk); //
    true System.out.println(disk instanceof Tovar);
    // true
    System.out.println(tovarDisk instanceof Tovar);
    // true System.out.println(tovarDisk instanceof
    Disk); // true
    System.out.println(tovar instanceof Tovar); //
true
    System.out.println(tovar instanceof Disk); //
    false System.out.println(tovarProcesor
    instanceof Tovar); // true
    System.out.println(tovarProcesor instanceof
    Procesor); // true
}
```

Napr. objekt typu **Disk** je typu Disk aj typu Tovar, bez ohľadu na typ premennej.

Operátor instanceof možno použiť aj na testovanie, či objekt implementuje určité rozhranie.

```
public interface RozhranieA { ..... }
public interface RozhranieB extends RozhranieA {
.....}
public class ImplementaciaA implements RozhranieA {
..... }
public class ImplementaciaB implements RozhranieB {
..... }
public class main {
    public static void main(String[] args) {
     ImplementaciaA a = new ImplementaciaA();
     ImplementaciaB b = new ImplementaciaB();
     Object o = new Object();

    System.out.println(a instanceof RozhranieA); //
    true System.out.println(a instanceof
    RozhranieB); // false System.out.println(b
    instanceof RozhranieA); // true
    System.out.println(b instanceof RozhranieB); //
    true System.out.println(o instanceof
    RozhranieB); // false
    }

}
```

IX. Prekrývanie (override) metód inštancie

Ak má inštančná metóda v podtriede rovnakú signatúru a návratový typ ako inštančná metóda v nadtriede, tak inštančná metóda v podtriede prekrýva **(override)** inštančnú metódu v nadtriede.

Prekrývajúca metóda v podtriede môže tiež vracať podtyp typu, ktorý vracia metóda v nadtriede. Ide o **kovariantný** návratový typ.

Príklad:

```
public class Nadtrieda {
    public int neprekryta(int a, int b) {
        System.out.println("Nadtrieda.neprekryta(in
    t,int)");
        return 2*(a+b);
    }
    public int prekryta1(int a, int b) {
        System.out.println("Nadtrieda.prekryta1(int
    ,int)");
        return a+b;
    }
    public Nadtrieda prekryta2(double a) {
        System.out.println("Nadtrieda.prekryta2(dou
        ble)");
        return new Nadtrieda();
        }
    public Object prekryta3() {
        System.out.println("Nadtrieda.prekryta3()")
    ;
        return new Object();
    }
```

```
public class Podtrieda extends Nadtrieda{
    public int prekryta1(int a, int b) {
        System.out.println("Podtrieda.prekryta1(int
        ,int)");
        return a*b;
    }
    public Nadtrieda prekryta2(double a) {
        System.out.println("Podtrieda.prekryta2(dou
        ble)");
        return new Podtrieda();
    }
    public Podtrieda prekryta3() {
        System.out.println("Podtrieda.prekryta3()")
        ;
        return new Podtrieda();
    }
}
```

```
public static void main(String[] args) {
    Nadtrieda nad = new Nadtrieda();
    Podtrieda pod = new Podtrieda();
    nad.neprekryta(1, 2);   //
    Nadtrieda.neprekryta(int,int) pod.neprekryta(1,
    2); // Nadtrieda.neprekryta(int,int)
    nad.prekryta1(1, 2);        //
    Nadtrieda.prekryta1(int,int)
    pod.prekryta1(1, 2);    //
    Podtrieda.prekryta1(int,int) nad.prekryta2(1);
            // Nadtrieda.prekryta2()
    pod.prekryta2(1);       //
Podtrieda.prekryta2()
    nad.prekryta3();        // Nadtrieda.prekryta3(
    pod.prekryta3();        //
    Podtrieda.prekryta3()
}
```

X. Ukrývanie (hide) metód triedy

Ak je v podtriede definovaná statická metóda s rovnakou signatúrou ako v nadtriede, tak metóda v podtriede ukrýva príslušnú metódu z nadtriedy.

Modifikátory prístupu

Zmena modifikátora prístupu

Pri prekrývaní metód môžeme zmeniť modifikátor prístupu k metóde. Prístupové práva **môžeme rozšíriť, ale nemôžeme ich zmenšiť.**

modifikátor v nadtriede	možný modifikátor v podtriede
private	tieto členy sa nededia
package-private	package-private, protected, public
protected	protected, public
public	Public

Tab. 2 Zmena modifikátora prístupu

Prístupové práva

modifikátor	v rámci triedy	v inej triede toho istého balíka	v podtriede, ktorá je v inom balíku	v inej triede, iného balíka
public	ÁNO	ÁNO	ÁNO	ÁNO
protected	ÁNO	ÁNO	NIE	NIE
Bez modifikátoru	ÁNO	ÁNO	NIE	NIE
private	ÁNO	ÁNO	NIE	NIE

Tab. 3 Tabuľka prístupových práv

Ukrývanie členských premenných

Členská premenná, ktorá má rovnaký názov ako členská premenná v nadtriede, ukryje príslušnú premennú z nadtriedy (aj vtedy ak sa líšia typom). Potom možno k členskej premennej z nadtriedy pristupovať pomocou kľúčového slova **super** (bude vysvetlené ďalej).

Ukrývanie členských premenných vo všeobecnosti **komplikuje** čitateľnosť kódu.

Použitie kľúčového slova super

Kľúčové slovo **super** sa používa na prístup k členom nadtriedy a pre volanie koštruktora nadtriedy. Jeho použitie je analogické s použitím kľúčového slova **this**.

Použitie super na prístup k členom nadtriedy

V členských funkciách možno pristupovať k členom nadtriedy použitím kľúčového slova super. Využíva sa pri **ukrytých** členských premenných a **prekrytých** metódach.

Kľúčové slovo **super** je akoby **referenciou** na časť objektu (definovanú v nadtriede) nad ktorým sa práve vykonáva členská funkcia.

Príklad:

Nadtrieda.java:

```
public class Nadtrieda {
    protected int atribut;

    public void prekrytaFunkcia() {
        System.out.println("funkcia v nadtriede");
    }
}
```

Podtrieda.java:

```
public class Podtrieda extends Nadtrieda {
    protected int atribut;

    public void prekrytaFunkcia() {
        System.out.println("startujem v
        podtriede"); super.prekrytaFunkcia();
        this.atribut = 10; // this možno vynechať
        super.atribut =20;
        System.out.println("this.atribut="+this.atr
        ibut
                    +",
                       super.atribut="+super.atribut
                       );
    }
}
```

Použitie:

```
public static void main(String[] args) {
    Podtrieda pod = new Podtrieda();
    pod.prekrytaFunkcia();
}
```

Výstup:

```
startujem v podtriede
funkcia v nadtriede
this.atribut = 10, super.atribut = 20
```

Volanie konštruktora nadtriedy

Pri vytváraní objektu, každý konštruktor zavolá najprv konštruktor nadtriedy a potom sa vykoná zvyšok daného konštruktora.

Príklad:

Nech sú definované triedy A, B, C také, že:

- C dedí od B,
- B dedí od A,
- A nemá uvedenú nadtriedu, preto je jej nadtriedou trieda **Object**

Ak vytvoríme inštanciu triedy C, tak:

- **konštruktor triedy C** na svojom začiatku vyvolá **konštruktor triedy B**,
- **konštruktor triedy B** na svojom začiatku vyvolá **konštruktor triedy A,**
- **konštruktor triedy A** na svojom začiatku vyvolá **konštruktor triedy Object**,
- po dokončení **konštruktora triedy Object** sa vykoná zvyšok **konštruktora triedy A**
- po dokončení **konštruktora triedy** A sa vykoná zvyšok **konštruktora triedy B**
- po dokončení **konštruktora triedy** B sa vykoná zvyšok **konštruktora triedy C**

Toto sa nazýva **reťazenie** konštruktorov.

Konštruktor nadtriedy možno vyvolať **explicitne** pomocou kľúčového slova **super**. Ak konštruktor explicitne nevyvoláme, potom volanie konštruktora doplní kompilátor (implicitné volanie konštruktora).

Príklad:

Nadtrieda.java:

```
public class Nadrieda {
    private int atribut1;

    public Nadrieda(int atribut1) {
        this.atribut1 = atribut1;
    }
        public Nadrieda() {
            this(0);
    }
}
```

Podtrieda.java:

```
public class Podtrieda extends Nadrieda {
    private int atribut2;

    public Podtrieda(int atribut1, int atribut2) {
        super(atribut1);
        this.atribut2 = atribut2; }

    public Podtrieda(int atribut2) {
        super();  // mozeme vynechat (kompilator
    dokaze doplnit)
        this.atribut2 = atribut2;
    }
        public Podtrieda() {
        this(0);
    }
}
```

Konštruktor nadtriedy môže byť vyvolaný iba z konštruktora. Volanie konštruktora nadtriedy musí byť uvedené ako **prvý príkaz** konštruktora.

Nadtrieda môže obsahovať **niekoľko** konštruktorov. Požadovaný konštruktor nadtriedy sa vyberie podľa parametrov uvedených pri jeho volaní.

Ak volanie konštruktora nadtriedy **nie je** explicitne uvedené, kompilátor automaticky doplní volanie konštruktora nadtriedy bez parametrov. Ak nadtrieda takýto konštruktor neobsahuje, potom vznikne **chyba pri kompilácii**.

Podobne, ak v triede explicitne **nedefinujeme** konštruktor, potom jej implicitný konštruktor zavolá konštruktor nadtriedy bez parametrov.

Poznámka 1: Trieda **Object** obsahuje konštruktor bez parametrov.

Poznámka 2: Ak je v triede explicitne definovaný konštruktor s parametrami, ale nie je explicitne definovaný konštruktor bez parametrov, tak trieda neobsahuje implicitný konštruktor bez parametrov (kompilátor nedoplní implicitný konštruktor).

Finálne metódy

Finálna metóda je metóda, ktorej implementácia sa v podtriedach **nemôže** meniť (v podtriede nemôže byť prekrytá). Označuje sa kľúčovým slovom **final**.

Finálne metódy sa napr. používajú pri inicializácii inštančných premenných.

Vo všeobecnosti je vhodné deklarovať metódy volané z konštruktorov ako **final** (predefinovanie takejto metódy v podtriede môže mať nežiaduce následky)

Príklad:

```
public class Trieda {
    public final void finalnaMetoda(int p1, int p2)
{
    // implemntacia metody,
    // ktoru v podtriede nie je mozne prekryt
    }
}
```

Finálne triedy

Finálna trieda je trieda, ktorá **nemôže** mať **podtriedy**. Označuje sa kľúčovým slovom **final**.

Finálne triedy je vhodné použiť napr. pri vytváraní tried, ktorých inštancie sú **nemenné** (immutable) t.j. ich **vnútorný stav** (atribúty objektu) sa po vytvorení nemení. Príkladom je trieda **String**.

Príklad:

```
public final class FinalnaTrieda {
    private int atribut1;

    public void funkcia(){
        // .....
    }
}
```

Abstraktné metódy a triedy

Abstraktná metóda je metóda deklarovaná bez implementácie. Deklaruje sa kľúčovým slovom **abstract**.

Abstraktná trieda je trieda, z ktorej **nemožno** vytvoriť **inštancie**. Je deklarovaná s kľúčovým slovom **abstract**. Môže, ale nemusí obsahovať abstraktné metódy. Ak trieda obsahuje aspoň jednu abstraktnú metódu, potom musí byť deklarovaná s kľúčovým slovom **abstract**.

Príklad:

GrafickyPrvok.java:

```
public abstract class GrafickyPrvok {
    private boolean zobrazovat;

    public abstract void kresli();
    public abstract void posun(double posunX,
double posunY);

    public boolean jeZobrazeny() {
        return zobrazovat;
    }
}
```

Kruznica.java:

```
public class Kruznica extends GrafickyPrvok {
    private double stredX, stredY;
    private double polomer;

    public void kresli() {
        // kod pre vykreslenie }

    public void posun(double posunX, double posunY)
{
        stredX + = posunX;
        stredY + = posunY;
    }

    public double obvod() {
        return 2 * Math.PI * polomer;
    }
}
```

Použitie vo funkcii main:

```
public static void main(String[] args) {
    // GrafickyPrvok g = new GrafickyPrvok(); <-
CHYBA
    Kruznica k = new Kruznica();
}
```

Na rozdiel od rozhraní, môžu abstraktné triedy obsahovať členské premenné, ktoré nie sú typu **static** a **final**, a môžu obsahovať aj implementované metódy.

Implementácia rozhrania abstraktnou triedou

Abstraktná trieda nemusí implementovať všetky metódy rozhrania.

Príklad:

Rozhrane.java:

```
public interface Rozhranie {
    void metoda1(int parameter1, int parameter2);
    void metoda2(int parameter);
}
```

AbstraktnaTrieda.java:

```
public abstract class AbstraktnaTrieda implements
Rozhranie{
    public void metoda1 (int parameter1, int
parameter2){
        // implementacia
    }
    // metoda2 zostáva abstraktnou metódou
}
```

KonkretnaTrieda.java:

```
public class KonkretnaTrieda extends
AbstraktnaTrieda{
    public void metoda2(int parameter) {
        // implementacia
    }
}
```

Všetky metódy rozhrania sú **implicitne abstraktné**, takže sa modifikátor **abstract** pri definícii metód rozhrania nemusí používať.

V príklade metóda 2 zostáva v triede **AbstraktnaTrieda** abstraktnou metódou. Je implementovaná až v triede KonkretnaTrieda.

Trieda, ktorá dedí od nadtriedy aj implemntuje rozhrania

Podľa konvencie sa najprv uvádza klauzula **extends** a potom klauzula **implements**.

Príklad:

```
public class Trieda extends Nadtrieda implements
Rozhranie1, Rozhranie2 {
    // .....
}
```

XI. Metódy triedy Object

Všetky triedy sú priamymi, alebo nepriamymi **potomkami** triedy **Object** v balíku **java.lang**. Tento balík sa importuje automaticky.

Trieda **Object** definuje niekoľko metód. Niektoré z nich možno prekryť **kódom**, ktorý bude špecifický pre danú triedu.

Zoznámime sa s metódami:

- protected Object clone() throws CloneNotSupportedException
- public boolean equals(Object obj)
- public int hashCode()
- public String toString()
- public final Class<?> getClass()
- protected void finalize() throws Throwable

Neskôr sa zoznámime s metódami:

- public final void notify()
- public final void notifyAll()
- public final void wait() throws InterruptedException
- public final void wait(long timeout) throws InterruptedException
- public final void wait(long timeout,int nanos) throws InterruptedException

XII. Metóda clone()

```
protected Object clone() throws CloneNotSupportedException
```

Ak trieda, alebo niektorá z jej nadtried implementuje rozhranie **Cloneable**, potom môžeme pomocou metódy clone() vytvoriť kópiu existujúceho objektu.

Implementácia tejto metódy v triede Object najprv kontroluje, či bola vyvolaná nad objektom ktorý implementuje rozhranie Cloneable.

Ak objekt toto rozhranie neimplementuje, tak vyhodí výnimku **CloneNotSupportedException** (výnimkami sa budeme zaoberať neskôr.)

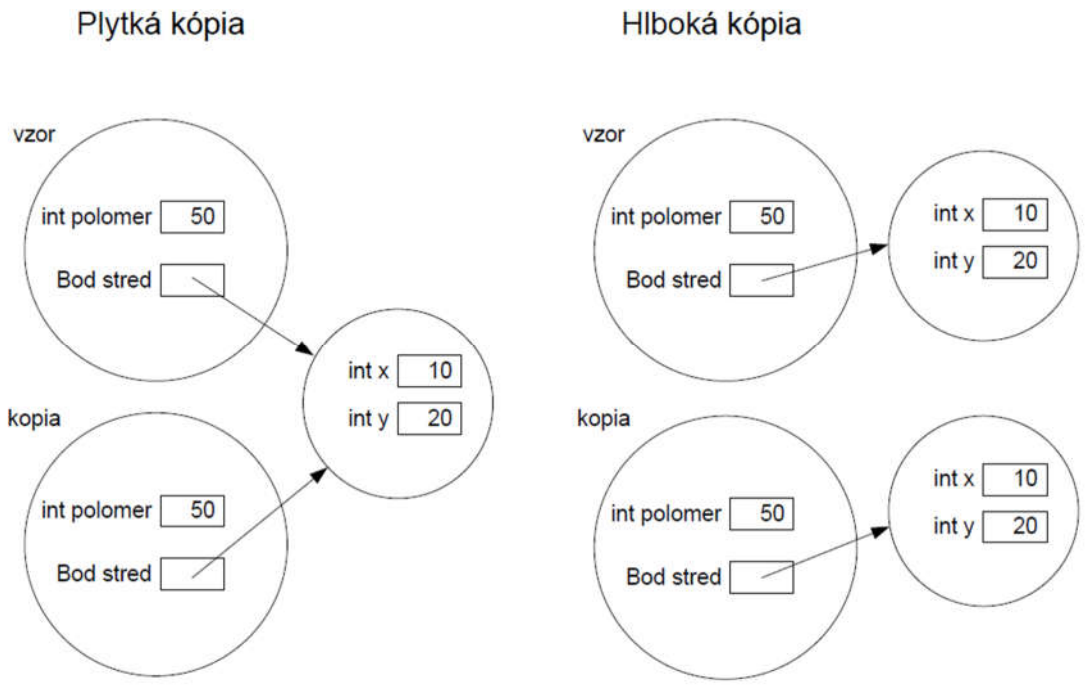
Ak objekt toto rozhranie implementuje, tak metóda vytvorí a vráti jeho kópiu (vráti nový objekt rovnakého typu s rovnakými hodnotami atribútov).

Rozoznávame 2 druhy vytvárania kópii objektov:

- **Plytká kópia objektu** (shallow copy) – atribúty referenčného typu v originály aj v kópii odkazujú na tie isté objekty.
- **Hlboká kópia objektu** (deep copy) – atribúty referenčného typu v originály aj v kópii odkazujú na iné objekty, ktorých vnútorný stav je ale rovnaký.

V prípadoch keď potrebujeme vytvoriť plytkú kópiu objektu, je implementácia metódy v triede Object vyhovujúca. Ak chceme vytvoriť hlbokú kópiu, treba vytvoriť vlastnú implementáciu metódy **clone().**

Príklad (plytká a hlboká kópia objektov triedy kružnica):



Obr. 10 Plytká a hlboká kópia

Príklad 1:

– plytká kópia:

```
public class Bod {
    private int x;
    private int y;
    public Bod(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
public class Kruznica implements Cloneable {
    private Bod stred;
    private int polomer;

    public Kruznica(Bod stred, int polomer) {
        this.stred = stred;
        this.polomer = polomer;
    }

    public Object clone() throws
CloneNotSupportedException {
        return super.clone();
    }
}
```

Príklad 2:

– hlboká kópia

```
public class Bod {
    private int x;
    private int y;

    public Bod(int x, int y){
        this.x = x;
        this.y = y; }

    public Bod(Bod vzor) {
        this.x = vzor.x;
        this.y = vzor.y;
    }
}
```

```
public class Kruznica implements Cloneable {
    private Bod stred;
    private int polomer;

    public Kruznica(Bod stred, int polomer) {
        this.stred = new Bod(stred);
        this.polomer = polomer;
    }

    public Object clone() throws
CloneNotSupportedException {
        return new Kruznica(new Bod(stred),
    polomer);
    }
}
```

Príklad 3:

– hlboká kópia:

```
public class Bod implements Cloneable {
    private int x;
    private int y;

    public Bod(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Bod(Bod vzor) {
        this.x = vzor.x;
        this.y = vzor.y;
    }
```

```
    public Object clone() throws
CloneNotSupportedException {
        return super.clone();
    }
}

public class Kruznica implements Cloneable {
    private Bod stred; private int polomer;

    public Kruznica(Bod stred, int polomer) {
        this.stred = new Bod(stred);
        this.polomer = polomer;
    }

    public Object clone() throws
CloneNotSupportedException {
        Kruznica kopia = (Kruznica) super.clone();
        kopia.stred = (Bod)stred.clone();
        // alebo kopia.stred = new Bod(stred)
        return kopia;
    }
}
```

XIII. Metóda equals()

```
public boolean equals(Object obj)
```

Metóda porovnáva **dva** objekty. Ak sú rovnaké, vráti **true**, inak vráti **false**. Implementácia metódy v triede **Object** porovnáva referencie na objekty (pomocou operátora identity ==). Ak chceme zistiť či sú dva objekty v rovnakom stave (či majú rovnaké hodnoty atribútov), potom treba túto metódu prekryť.

Ak je prekrytá metóda **equals(),** potom je potrebné prekryť aj metódu **hashCode().**

Príklad:

```
public class Bod {
    private int suradnicaX;
    private int suradnicaY;
    public Bod(int suradnicaX, int suradnicaY) {
        this.suradnicaX = suradnicaX;
        this.suradnicaY = suradnicaY;
    }
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Bod other = (Bod)obj;
        if (this.suradnicaX != other.suradnicaX) {
            return false;
        }
```

```
        if (this.suradnicaY != other.suradnicaY) {
            return false;
        }
        return true;
    }

    public int hashCode(){
        int hash = 7;
        hash = 53 * hash + this.suradnicaX;
        hash = 53 * hash + this.suradnicaY;
        return hash;
    }
}

public class Kruznica {
    private Bod stred;
    private int polomer;
    public Kruznica(Bod stred, int polomer) {
        this.stred = new Bod(stred);
        this.polomer = polomer;
    }
    public boolean equals(Object obj) {
        if(obj == null){
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
```

```
        final Kruznica other = (Kruznica) obj;
        if(this.stred != other.stred && (this.stred
        == null ||
        !this.stred.equals(other.stred))) {
            return false;
        }
        if (this.polomer != other.polomer) {
            return false;
        }
            return true;
        }
    public int hashCode() {
        int hash = 7;
        hash = 17 * hash +
            (this.stred != null ?
            this.stred.hashCode() : 0);
        hash = 17 * hash + this.polomer;
        return hash;
    }
}

public static void main(String[] args) {
    Bod b1 = new Bod(10, 20);
    Bod b2 = new Bod(10, 20);
    Bod b3 = new Bod(7, 8);

    Kruznica k1 = new Kruznica(b1, 5);
    Kruznica k2 = new Kruznica(b2, 5);
    Kruznica k3 = new Kruznica(b3, 5);
    System.out.println(b1.equals(b2)); // true
    System.out.println(b1.equals(b3)); // false
```

```
        System.out.println(b1.hashCode()); // 20213
        System.out.println(b2.hashCode()); // 20213
        System.out.println(b3.hashCode()); // 20042

        System.out.println(k1.equals(k2)); // true
        System.out.println(k1.equals(k3)); // false
}
```

XIV. Metóda hashCode()

```
public int hashCode()
```

Metóda hashCode() vracia hešovací kód objektu. Ak sú dva objekty **rovnaké**, potom musia mať aj **rovnaké hešovacie kódy**. Prekrytie metódy **equals(),** zmení spôsob porovnávania objektov. Preto ak prekryjeme metódu equals(), treba prekryť aj metódu hashCode().

Príklad prekrytia metódy hashCode() je uvedený vyššie.

Metóda toString()

```
public String toString()
```

Metóda toString() vracia **textovú reprezentáciu objektu**. Táto metóda je napr. užitočná pri ladení programu. Využívajú ju niektoré funkcie. Napríklad volanie **System.out.println**(Object obj) zavolá funkciu **String.valueOf(obj),** ktorá v prípade že obj nie je **null** zavolá funkciu **obj.toString()** a reťazec vrátený metódou **toString()** sa vytlačí.

Príklad:

```
public class Bod {
    private int suradnicaX;
    private int suradnicaY;

    public Bod(int suradnicaX, int suradnicaY) {
        this.suradnicaX = suradnicaX;
        this.suradnicaY = suradnicaY;
    }
    public String toString() {
        return "(" + suradnicaX + ", " + suradnicaY
    + ")";
    }
public static void main(String[] args) {

    Bod b1 = new Bod(10, 20);
    System.out.println(b1.toString()); // "(10,
20)"
    System.out.println(b1);          // "(10, 20)"

}
```

XV. Metóda finalize()

```
protected void finalize() throws Throwable
```

Túto metódu môže volať **garbage collector** ak objekt už nie je používaný (žiadne premenná neobsahuje referenciu na objekt). Táto metóda sa môže použiť aby si objekt „po sebe poupratoval“ napr. uvoľnením používaných prostriedkov.

Nie je ale zaručené, že túto metódu systém zavolá. **Implementácia** metódy **finalize()** v triede **Object** nemá **žiadny efekt**. Prekrytá metóda finalize() by mala na konci volať metódu finalize() nadtriedy.

Príklad:

```
public class Bod {
    private int suradnicaX;
    private int suradnicaY;

    public Bod(int suradnicaX, int suradnicaY) {
        this.suradnicaX = suradnicaX;
        this.suradnicaY = suradnicaY;
    }

    protected void finalize() throws Throwable {
        System.out.println("metoda finalize");
        super.finalize();
    }
}
```

XVI. Metóda getClass()

```
public final Class<?> getClass()
```

Metódu **getClass()** nemožno prekryť. Táto metóda vráti objekt typu **Class** s metódami, ktoré umožňujú získať informácie o triede.

Príklad:

```
public class Bod {
    private int suradnicaX;
    private int suradnicaY;
    public Bod() {
        this(0, 0);
    }
        public Bod(int suradnicaX, int suradnicaY)
    {
        this.suradnicaX = suradnicaX;
        this.suradnicaY = suradnicaY; }
    public String toString() {
        return "(" + suradnicaX + ", " + suradnicaY
    + "(";
    }
}

import java.lang.reflect.Field;
```

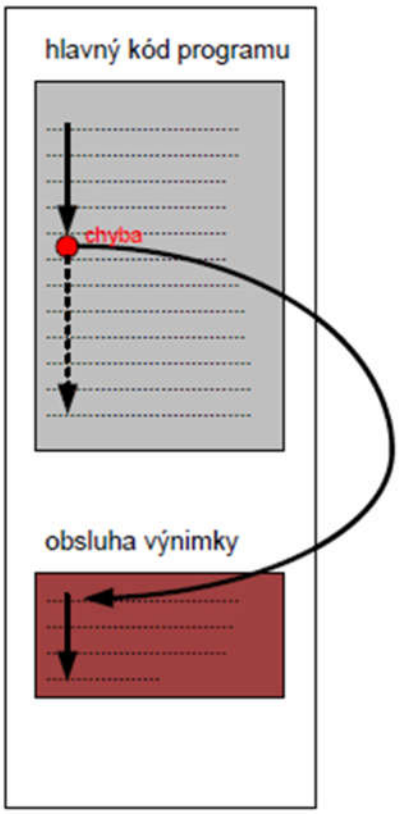
```
public static void main(String[] args) {
    Bod b1 = new Bod(10, 20);
    System.out.println(b1.getClass().getSimpleName(
    ));
    System.out.println(b1.getClass().getSuperclass(
    ).getSimpleName());

    System.out.println(b1.getClass().getSuperclass(
    ).getName()); System.out.println();

    for (Field atribut :
       b1.getClass().getDeclaredFields()) {
       System.out.println(atribut.getName());
    }
    System.out.println();

    Bod b2 =
    b1.getClass().getConstructor(int.class,
    int.class).newInstance(11,22);
    System.out.println(b2);

    Bod b3 = b1.getClass().newInstance();
    System.out.println(b3);
}
```

Výstup:

```
Bod
Object
java.lang.Object
suradnicaX
suradnicaY
(11, 22)
(0, 0)
```

XVII. Spracovanie výnimiek (exceptions)

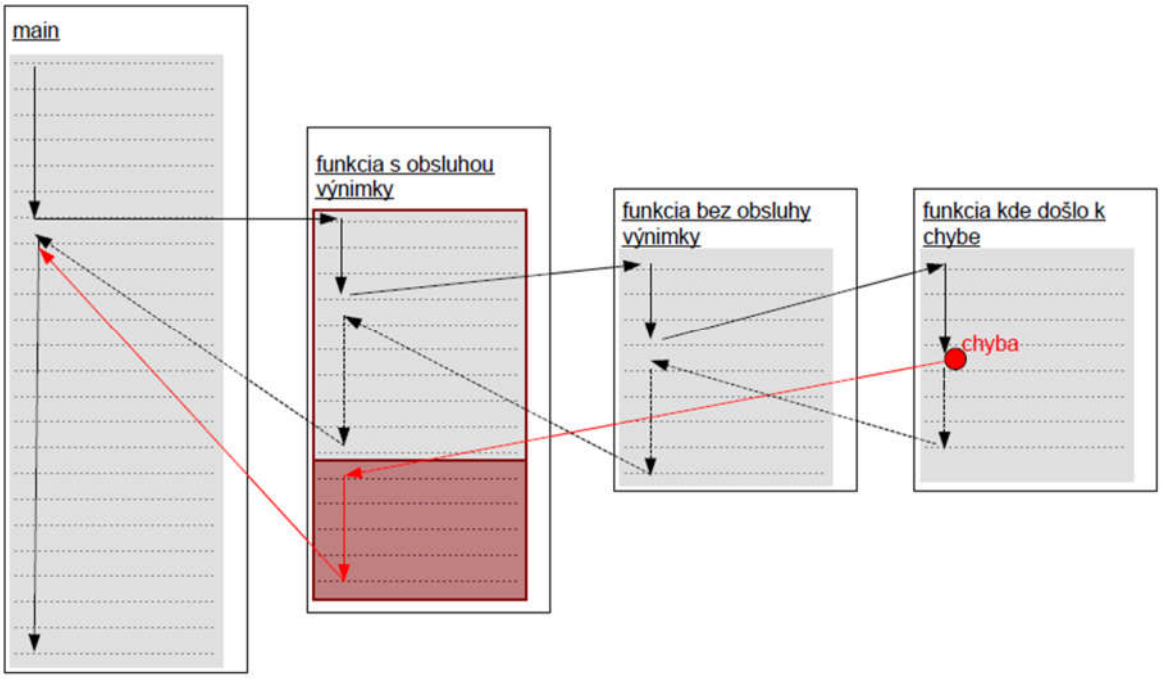
Výnimka (exception) je udalosť, ktorá narušuje normálny priebeh poradia vykonávaných
inštrukcii programu. Používa sa pre **obsluhu chýb** a iných mimoriadnych udalostí.



Obr. 12 Spracovanie výnimiek

Pri vzniku chyby je vytvorený **objekt výnimky**. Ten je predaný do **runtime** systému. Objekt
výnimky obsahuje **informácie o chybe**.

Vytvorenie objektu výnimky a jeho predanie runtime systému sa označuje ako spôsobenie
výnimky alebo vyhodenie výnimky **(throwing an exception).**



Obr. 13 Objekt výnimky

Po vyhodení výnimky runtime systém hľadá najbližší blok kódu určený pre spracovanie výnimky. Tento blok kódu sa nazýva obsluha výnimky **(exception handler).**

Každá obsluha je určená na **spracovanie** určitého typu výnimky – **typ výnimky** je daný typom **objektu výnimky.**

Hovoríme, že obsluha výnimky zachytáva výnimku **(catch exception).**

Ak chyba vznikla vo funkcii ktorá neobsahuje príslušnú obsluhu výnimky, tak sa obsluha hľadá v niektorej aktuálne nadradenej funkcii (z hľadiska volania funkcii). Na to sa využíva **zásobník volaní** (call stack).

Zachytenie a spracovanie výnimiek

```
try {
    // kód kde môže vzniknúť chyba
}
catch (TypVynimky1 objektVynimky) {
    // obsluha výnimky typu (alebo podtypu) TypVynimky1
}
catch (TypVynimky2 objektVynimky) {
    // obsluha výnimky typu (alebo podtypu) TypVynimky2
}
finally {
  // kód ktorý sa vykonaná vždy po bloku try

}
```

- **Blok try** – uzaviera **časť kódu,** kde môžu vzniknúť výnimky ktoré chceme zachytávať. Za blokom try musí nasledovať blok catch (prípadne viac blokov catch), alebo blok finally, alebo obidva druhy blokov s ktorých musia byť prvé bloky catch.

- **Bloky catch** – ku bloku try môže byť pridružený jeden, alebo viac blokov **catch**, ktoré slúžia **na obsluhu výnimiek** vyhodených z bloku **try**. Medzi koncom bloku try a začiatkom bloku catch nesmie byť žiadny kód.

  Každý blok catch slúži na obsluhu určitého typu výnimky. Typ výnimky je daný triedou, ktorá je priamym, alebo nepriamym potomkom triedy **Throwable**.

Ak sa v niektorej časti bloku try vyhodí **výnimka**, tak sa nepokračuje ďalej vo vykonávaní príkazov bloku try, ale začne sa vykonávať obsluha výnimky v bloku catch. Zvyšok bloku try sa už nevykoná. Po obsluhe výnimky sa pokračuje za blokmi catch.

- **Blok finally** – Slúži na **„upratanie/čistenie“**. Príkazy v bloku **finally** sa vykonajú vždy po ukončení bloku try, bez ohľadu na to či bola vyhodená výnimka, alebo nie. A to aj v prípade ak sa v bloku try vykonal príkaz **return, continue**, alebo **break**.

  Umiestnenie „upratovacieho“ kódu do bloku finally patrí medzi dobré programátorské postupy. Výhodou bloku finally je to, že sa vykoná aj v prípade, ak je vyhodená výnimka obslúžená v bloku catch umiestenom v inej funkcii (nadradenej v zásobníku volaní).

Postup vykonávania blokov:

1. **blok try** (dokonca, alebo po vyhodenie výnimky)

2. **blok cath** – vykoná sa ak v bloku try dôjde k vyhodeniu výnimky a typ objektu výnimky je rovnaký ako typ uvedený v bloku catch, alebo je podtypom typu uvedeného v bloku catch.

3. **blok finally** (vždy, ak je uvedený)

4. pokračuje sa **vykonávaním kódu** uvedenom za týmito blokmi

Príklad:

```
System.out.println("zaciatok programu");
try {
    System.out.println("blok try (zaciatok)");
    // String retazec = "male pismena";
    String retazec = null;
    System.out.println(retazec.toUpperCase());    //
    vyhodí  výnimku  System.out.println("blok  try
    (dokonceny)");
}
catch (NullPointerException objektVynimky) {
    System.out.println("blok catch");
}
finally {
    System.out.println("blok finally");
}
System.out.println("dalsi kod programu");
```

Výstup:

```
zaciatok programu
blok try(zaciatok)
blok catch
blok finally
dalsi kod programu
```

Príklad (viac catch blokov, použitie objektu výnimky v obsluhe výnimky):

```
System.out.println("zaciatok programu");
try {
    System.out.println("blok try (zaciatok)");
    String retazec = "male pismena";
    // retazec = null;
    System.out.println(retazec.toUpperCase()); //
    teraz OK
    System.out.println(retazec.charAt(50)); //
    vyhodi vynimku System.out.println("blok try
    (dokonceny)");
}
catch (NullPointerException objektVynimky) {
    System.out.println("blok catch (nulovy
    pointer)"); objektVynimky.printStackTrace(); //
    pouzitie objektu vynimky
}
catch (IndexOutOfBoundsException objektVynimky) {
    System.out.println("blok catch (index mimo
    rozsahu)"); objektVynimky.printStackTrace(); //
    pouzitie objektu vynimky
```

```
}
finally {
    System.out.println("blok finally");
}
System.out.println("dalsi kod programu");
```

Výstup:

```
zaciatok programu
blok try(zaciatok)
MALE PISMENA
blok catch(index mimo rozsahu)
java.lang.StringIndexOutOfBoundsException:    String
index out of range: 50
    at java.lang.String.charAt(String.java:686)
    at
vynimky.VynimkyTest2.main(VynimkyTest2.java:11)
blok finally
dalsi kod programu
```

Poznámka: Funkcia printStackTrace vytlačí na štandardný chybový výstup informácie o trasovanie zásobníka volaní.

Vo výstupe sa poradie riadkov môže líšiť, pretože funkcia printStackTrace vypisuje na štandardný chybový výstup, ale ostatné výpisy sú na štandardný výstup.

Ak je uvedených viacero blokov catch za sebou, obsluha výnimky sa hľadá v takom poradí v akom sú bloky catch uvedené

Príklad (vnorenie try-catch-finally v inom try, obsluha výnimky vyššie):

```
public static void pracujSRetazcom(String retazec)
{
    try {
        System.out.println("blok try - vnutorny
        (zaciatok)");
        System.out.println(retazec.charAt(50)); //
        NullPointerException
        System.out.println("blok try - vnutorny
        (dokonceny)");
    }
    catch (IndexOutOfBoundsException objektVynimky)
        { System.out.println("blok catch - vnutorny
        (index mimo rozsahu)");
    }
    finally {
        System.out.println("blok finally -
        vnutorny");
    }
```

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

public static void main(String[] args) {
    try {
        System.out.println("blok try - vonkajsi
    (zaciatok)");
        String retazec = "male pismena";
        retazec = null;
        pracujSRetazcom(retazec);
        System.out.println("blok try - vonkajsi
    (dokonceny)");
    }
    catch (NullPointerException objektVynimky) {
    System.out.println("blok catch - vonkajsi
    (nulovy pointer)");
    }
    finally { System.out.println("blok finally -
vonkajsi");
    }
}
```

Výstup:

- blok try - vonkajsi (zaciatok)
- blok try - vnutorny (zaciatok)
- blok finally - vnutorny
- blok catch - vonkajsi (nulovy pointer)
- blok finally - vonkajsi

Príklad (blok finally a príkaz return):

```
public static void main(String[] args) {
    System.out.println("zaciatok programu");
    try {
        System.out.println("blok try (zaciatok)");
        String retazec = "male pismena";
        // retazec = null;
        System.out.println(retazec.toUpperCase());
        // nevyhodi vynimku
        System.out.println("blok try (dokonceny)");
        return;
    }
    catch (NullPointerException objektVynimky) {
        System.out.println("blok
    catch(NullPointerException)");
    }
    finally {
        System.out.println("blok final - tento kod
    sa vykona");
    }
    System.out.println("tento kod sa uz nevykona");
}
```

Výstup:

zaciatok programu
blok try (zaciatok)
MALE PISMENA
blok try (dokonceny)
blok final - tento kod sa vykona

Príklad (bez bloku catch):

```
try {
    System.out.println("blok try (zaciatok)");
    String retazec = null;
    System.out.println(retazec.toUpperCase());    //
    vyhodi   vynimku   System.out.println("blok   try
    (koniec)");
}
finally {
    System.out.println("blok finally");
}
```

Zachytiť alebo určiť požiadavku (catch or specify requirement)

Kód ktorý by mohol spôsobiť výnimku treba uzavrieť jedným z dvoch spôsobov:

- uzavrieť do **bloku try**, za ktorým nasleduje obsluha výnimky
- uzavrieť do **funkcie**, ktorá uvádza (pomocou throws), že môže spôsobiť výnimku

Toto pravidlo sa nazýva **„zachytiť, alebo určiť“.** Niektoré výnimky musia splňovať toto pravidlo, iné ho splňovať nemusia.

Tri druhy výnimiek:

1. kontrolovaná výnimka (checked exception) – výnimočné stavy, s ktorými by sa mala aplikácia vedieť vyrovnať (napr. práca s I/O)

Kontrolované výnimky podliehajú požiadavke „zachytiť, alebo určiť“. Medzi kontrolované výnimky patria všetky výnimky okrem tých, ktoré sú inštanciami **tried Error, RuntimeException** a ich **podtriedami**.

2. chyba (error) – tieto výnimočné stavy majú obvykle príčinou **mimo** aplikácie. Aplikácia ich obvykle nemôže predpokladať a vyriešiť.

Napr. aplikácia úspešne otvorí súbor na čítanie, ale nemôže ho čítať kvôli poruche hárdveru, alebo poruche systému.

Programátor môže, ale nemusí takúto výnimku zachytávať. Na tento druh výnimiek sa požiadavka „zachytiť, alebo určiť“ nevzťahuje.

Tieto výnimky sú označené triedou **Error** a jej podtriedami.

3. výnimka za behu (runtime exception) – tieto výnimočné stavy majú príčinu v aplikácii a aplikácia ich obvykle nemôže predpokladať a vyriešiť. Obvykle znamenajú programátorskú chybu. Napríklad logické chyby, alebo nesprávne použitie rozhrania API.

Napríklad ak je na niektorom mieste programu očakávaná „nenulová“ referencia na objekt, ale pri vykonávaní má táto referencia hodnotu **null**.

Aplikácia môže zachytiť túto výnimku, ale lepšie je odstrániť chybu v programe, kvôli ktorej k výnimke došlo. Tieto výnimky nepodliehajú požiadavke „zachytiť, alebo určiť“. Výnimky sú označené triedou **RuntimeException**, alebo jej podtriedami.

Nekontrolované výnimky **(unchecked exception)** – spoločné označenie pre tieto druhy výnimiek: „chýba“ a „výnimka za behu“.

Určenie výnimky spôsobenej metódou

Určenie výnimky spôsobenej metódou sa používa v prípade, že vo funkcii môže vzniknúť chyba, ale chceme používateľa funkcie iba informovať o chybe a nechať ho aby rozhodol ako spracovať výnimku. Vtedy vymenujeme zoznam výnimiek, ktoré môže funkcie vyhodiť pomocou kľúčového slova thorws. Povinné je určovať iba kontrolované výnimky.

Príklad:

```
public static void pracujSRetazcom(String retazec)
throws                              NullPointerException,
IndexOutOfBoundsException {
    System.out.println("funkcia - zaciatok");
    System.out.println(retazec.charAt(50));       //
NullPointerException
    System.out.println("funkcia - dokoncena");
}
public static void main(String[] args) {
    try {
        System.out.println("blok try (zaciatok)");
        String retazec = "male pismena";
        retazec = null;
        pracujSRetazcom(retazec);
        System.out.println("blok try (dokonceny)");
    }

    catch (NullPointerException objektVynimky){
        System.out.println("blok catch
    (NullPointerException)");
    }
```

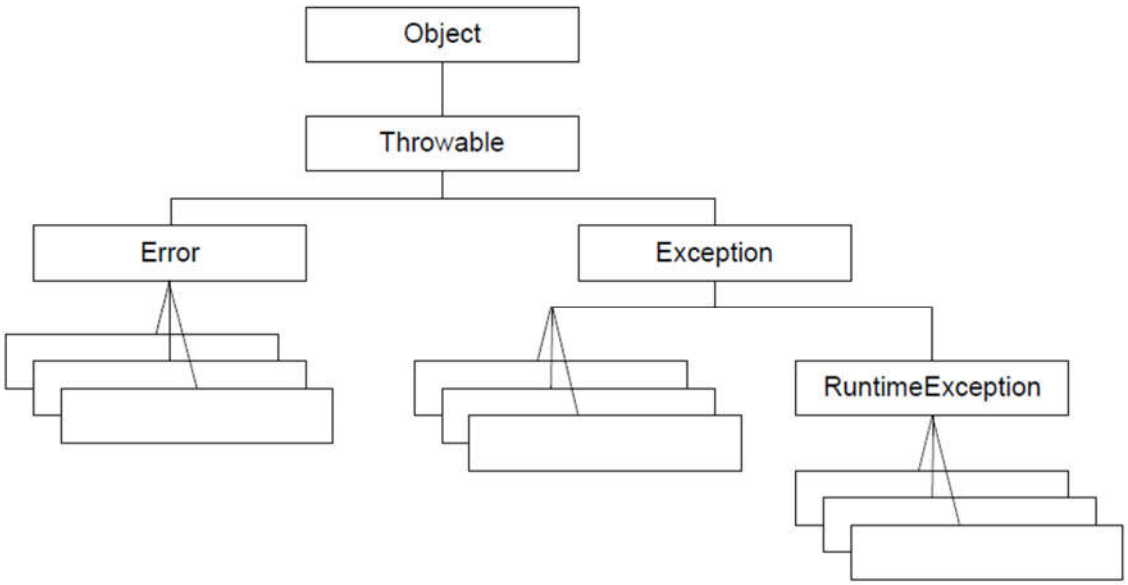
```
    catch (IndexOutOfBoundsException
objektVynimky){
        System.out.println("blok catch
    (IndexOutOfBoundsException)");
    }
    // blok final nie je povinny
}
```

Tento príklad nie je celkom dobrý pretože:

Výnimky **NullPointerException** a **IndexOutOfBoundsException** sú nekontrolované výnimky, preto ich v tomto príklade nie je nutné uvádzať.

XVIII. Trieda Throwable a jej podtriedy

Typ výnimky je daný triedou objektu výnimky. Všetky triedy výnimiek sú priamymi, alebo
nepriamymi potomkami triedy Throwable (java.lang.Throvable).



Obr. 14 Trieda Throwable

Trieda Error

Ak dôjde k chybe dynamického prepojenia, alebo inej závažnej chybe modulu JVM, spôsobí modul výnimku typu **Error**.

Jednoduché programy bežne nezachytávajú, ani nespôsobujú výnimky typu Error.

Trieda Exception

Väčšina programov vyvoláva a zachytáva objekty odvodené od triedy Exception. Objekt typu Exception oznamuje, že došlo k problému, ale nejde o závažný systémový problém.

Príklady:

- **IllegalAccessException** – nepodarilo sa nájsť konkrétnu metódu
- **NegativeArraySizeException** – pokus o vytvorenie poľa zo zápornou veľkosťou

Trieda RuntimeException

Je vyhradená pre výnimky oznamujúce nesprávne použitie rozhrania **API**.

Príklady:

- **NullPointerException** – nastáva pri pokuse o prístup k členu objektu pomocou referencie s hodnotou null
- **IndexOutOfBoundsException** – index mimo rozsahu

Ako spôsobiť (vyhodiť) výnimku

Výnimky sa vyhadzujú príkazom throw. Príkaz throw vyžaduje jediný argument: objekt ktorý možno vyvolať. Tieto objekty sú inštanciami ľubovoľnej podtriedy triedy **Throwable**. throw objektVynimky;

Príklad (vytvorenie vlastnej výnimky, spôsobenie výnimky):

```
public class ZasobnikVynimka extends Exception{
    public ZasobnikVynimka() {
        this("");
    }
    public ZasobnikVynimka(String sprava) {
        super(sprava);
    }
}
```

```
public class ZasobnikPrazdnyVynimka extends
    ZasobnikVynimka { private static final String
    sprava = "zasobnik prazdny";
    public ZasobnikPrazdnyVynimka() {
        super(sprava);
    }
}

public class ZasobnikPlnyVynimka extends
    ZasobnikVynimka {
    private static final String sprava = "zasobnik
    plny";

    public ZasobnikPlnyVynimka() {
        super(sprava);
    }

    public ZasobnikPlnyVynimka(double hodnota, int
kapacita){
        super(sprava + "(hodnota = " + hodnota
            + ", kapacita = " + kapacita + ")");
    }
}

public class Zasobnik {
    private double[] udaje;
    private int pocet;

    public Zasobnik(int kapacita){
        udaje = new double[kapacita];
        pocet = 0;
    }

    public double pop()throws
```

```
    ZasobnikPrazdnyVynimka {
        if (pocet == 0){
            throw new ZasobnikPrazdnyVynimka();
        }
        pocet--;
        return udaje[pocet];
    }

    public void push(double hodnota) throws
ZasobnikPlnyVynimka {
        if (pocet == udaje.length){
            throw new
        ZasobnikPlnyVynimka(hodnota,udaje.length);
        }
        udaje[pocet] = hodnota;
        pocet++;
    }
}
```

```
public static void main(String[] args){
    try{
        Zasobnik z = new Zasobnik(4);
        z.push(1);
        z.push(2);
        z.push(3);
        z.push(4);
        z.push(5);
        z.pop();
        z.pop();
        z.pop();
        z.pop();
        z.pop();
    }
    catch(ZasobnikPlnyVynimka vynimka){
        System.err.println(vynimka.getMessage());
    }
    catch (ZasobnikPrazdnyVynimka vynimka){
        System.err.println(vynimka.getMessage());
    }
}
```

Zreťazené výnimky

Aplikácie často reagujú na výnimku tak, že spôsobia ďalšiu výnimku

Príklad:

```
try {
    // .....
}

catch (IOException exception) {
    throw new NovaVynimka("detailna sprava",
exception);
}
```

Trieda Throwable obsahuje konštruktory a metódy podporujúce zreťazené výminky:

- **Throwable(String, Throwable)**
- **Throwable(Throwable)**
- **Throwable getCause()**
- **Throwable initCause(Throwable)**

Argument typu **Throwable** metódy initCause() a konštruktorov predstavuje výnimku, ktorá je príčinou aktuálnej výnimky.

Metóda **getCause()** vráti výnimku, ktorá spôsobila aktuálnu výnimku.

Kontrolované a nekontrolované výnimky

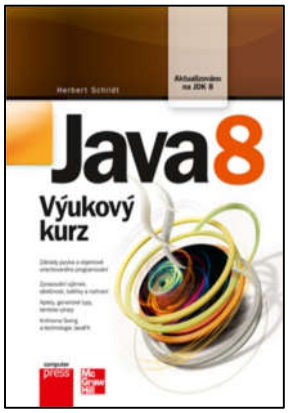
- Pri vytváraní vlastných typov výnimiek je vhodné , aby výnimky boli **kontrolované**.
- **Uvádzanie výnimiek**, ktoré môže funkcia spôsobiť je v popise metódy rovnako dôležité ako vstupné parametre a návratová hodnota. To sa týka hlavne kontrolovaných výnimiek.
- **Výnimky za behu** (napr. aritmetické výnimky, prístup k členu cez null) môžu nastať na ľubovoľnom mieste programu a v typickom programe ich môže byť veľa. Sú následkom programátorských chýb. Preto sa kvôli prehľadnosti výnimky za behu nepridávajú do deklarácii funkcii.

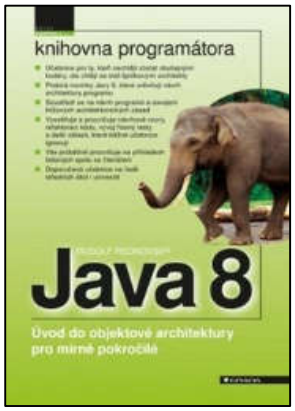
Výhoda použitia výnimiek

- **prehľadnosť**
- **oddelenie logiky** hlavného programu od spracovanie chýb
- ak je spracovanie chyby v niektorej nadradenej funkcii v zásobníku volaní, **netreba** prenášať informáciu o chybe cez návratové hodnoty, alebo cez argumenty. **Informácie** sa prenášajú **v objekte výnimky**

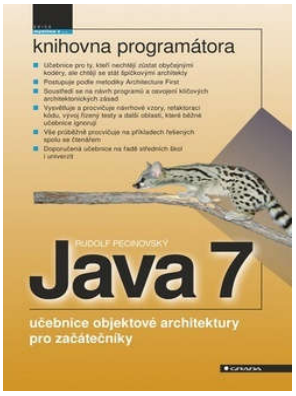
XIX. Odporúčaná literatúra a zdroje

1. **Java 8 -** Herbert Schildt
2. **Java bez předchozích znalostí -** James Keogh
3. **Java 8 -** Rudolf Pecinovský
4. **Myslíme objektově v jazyku Java -** Rudolf Pecinovský



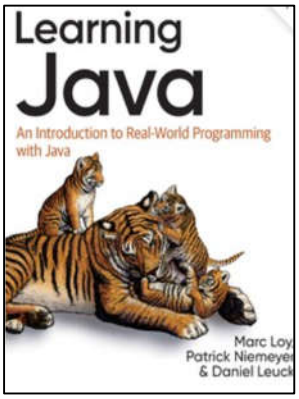


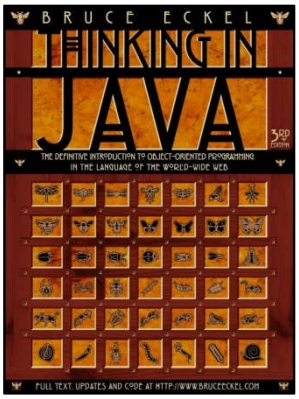
5. **Mistrovství Java -** Herbert Schildt
6. **Java 7 -** Rudolf Pecinovský
7. **1001 tipů a triků pro jazyk Java -** Bogdan Kiszka
8. **Učebnice jazyka Java 5 -** Pavel Herout

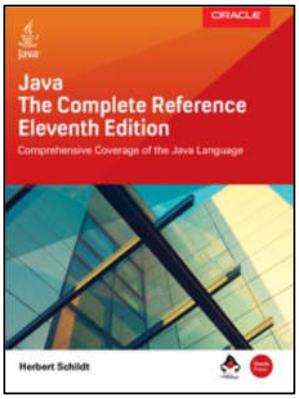


Zahraničná literatúra

1. **Learning Java -** Marc Loy
2. **Java for beginners -** Manuj Aggarwal
3. **Thinking in Java -** Bruce Eckel
4. **Java: The Complete Reference -** Herbert Schildt







5. **Optimizing Java -** Benjamin J Evans
6. **Effective Java -** Joshua Bloch
7. **Java Performance -** Scott Oaks
8. **Head First Java -** Kathy Sierra, Bert Bates