

# 3

## Algoritmus

# Pojmy zavedené v 2. prednáške<sub>(1)</sub>

- trieda ako
  - inštancia metatriedy
  - priamo definovaný objekt – Java
- delenie jazykov
  - založené na objektoch
  - založené na triedach – Java
- UML – grafické znázornenie triedy
  - názov, atribúty, metódy
  - rôzne pohľady podľa účelu
- UML – grafické znázornenie inštancie

# Pojmy zavedené v 2. prednáške<sub>(2)</sub>

- definícia triedy (trieda ako šablóna) – Java
  - definícia atribútov
  - definícia konštruktorov
  - definícia metód
- parametre konštruktorov a metód
  - formálne – v definícii metódy
  - skutočné – v správe
- návratová hodnota metódy
  - typ void

# Pojmy zavedené v 2. prednáške<sub>(3)</sub>

- primitívne dátové typy
  - pre celé čísla
  - pre reálne čísla
  - znaky
  - logické hodnoty
- objektový typ String
- literály
  - primitívnych dátových typov
  - String

# Pojmy zavedené v 2. prednáške<sub>(4)</sub>

- prirad'ovací príkaz
  - výraz, operátor priradenia
- príkaz return
- príkazy pre tlač do okna terminálu
- komentáre
  - jednoriadkové - // komentar
  - viacriadkové /\* komentar \*/
    - dokumentačné /\*\* komentar \*/

# Cieľ prednášky

- aritmetický výraz
- identifikátor
- algoritmus
  - definícia a vlastnosti
  - základné konštrukčné prvky
    - vetvenie
  - Algoritmus – znázornenie v UML
- lokálna premenná
- príklad: Lepšia verzia automatu MHD

# Aritmetický výraz<sub>(1)</sub>

- příklady výrazov

`this.trzba + this.cenaListka`

`this.vlozenaCiastka - this.cenaListka`

`0`

`cenaListka`

`this.pocetPredanychListkov + 1`

# Aritmetický výraz<sub>(2)</sub>

- predpis na výpočet číselnej hodnoty
- obvykle má tvar matematického výrazu
- aritmetický výraz môže mať formu:
  - bez operátorov
    - číselný literál
    - číselný parameter
    - číselný atribút
  - s aritmetickým operátorom
    - unárnyOperátor operand
    - operand binárnyOperátor operand



# Aritmetický výraz<sub>(3)</sub>

- příklady výrazov

`this.trzba + this.cenaListka`

`this.vlozenaCiastka - this.cenaListka`

`0`

`cenaListka`

`this.pocetPredanychListkov + 1`

# Unárne aritmetické operátory

- tvar:

operator operand

- operand môže byť ľubovoľný aritmetický výraz
- unárne aritmetické operátory:
  - - (mínus) – unárne mínus – zmena znamienka
  - + (plus) – unárne plus

# Binárne aritmetické operátory

- tvar

prvýOperand operátor druhýOperand

- operand môže byť ľubovoľný aritmetický výraz
- binárne aritmetické operátory:
  - + (plus) – súčet
  - - (mínus) – rozdiel
  - \* (hviezdička) – súčin
  - / (lomka) – podiel
  - % (percento) – zvyšok po delení

# Priorita operátorov<sub>(1)</sub>

- operátory sa vyhodnocujú v nasledujúcom poradí:
  - unárne +, -
  - binárne \*, /,%
  - binárne +, -
- teda rovnako ako v matematike

# Priorita operátorov<sub>(2)</sub>

- poradie vyhodnocovania sa dá ovplyvňovať zátvorkami – spôsobom obvyklým v matematike
- zátvorky treba používať vždy, keď to zlepší čitateľnosť výrazu

$$\frac{a \cdot b}{c \cdot d}$$

~~$a * b / c * d$~~

$(a * b) / (c * d)$

# Typ hodnoty aritmetického výrazu

- operand v aritmetickom výraze môže reprezentovať len číselnú hodnotu

operandy	byte, short, int	long	float	double
byte, short, int	int	long	float	double
long	long	long	float	double
float	float	float	float	double
double	double	double	double	double

# Typová kompatibilita – príkaz návratu<sub>(1)</sub>

```
public int getCenaListka() {  
    return this.cenaListka;  
}
```

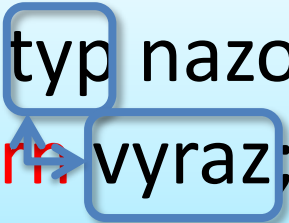




# Typová kompatibilita – príkaz návratu<sub>(2)</sub>

- typ výrazu musí byť konvertovateľný na typ návratovej hodnoty

```
public typ nazovMetody(parametre) {  
    return vyraz;  
}
```



# Typová kompatibilita – priradenie<sub>(1)</sub>

```
this.trzba = this.trzba + this.cenaListka;
```



The diagram illustrates the concept of type compatibility in variable assignment. It shows the code snippet `this.trzba = this.trzba + this.cenaListka;` where the variable `this.trzba` is used on both sides of the assignment. A blue box highlights the `this.trzba` on the left, and another blue box highlights the `this.trzba` on the right. A blue arrow points from the right-hand `this.trzba` to the left-hand `this.trzba`, indicating that the variable is being reassigned its own value plus another value, which is a common pattern in iterative calculations or accumulators.

# Typová kompatibilita – priradenie<sub>(2)</sub>

- typ výrazu na pravej strane priradovacieho príkazu musí byť konvertovateľný na typ ľavej strany priradovacieho príkazu

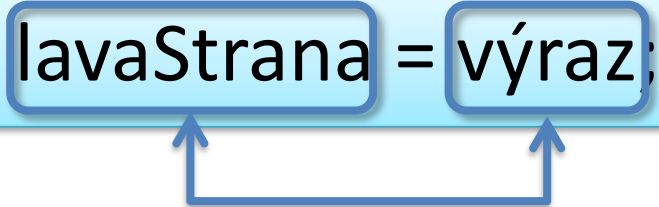


Diagram illustrating the assignment statement: `lavaStrana = výraz;`. The variable `lavaStrana` and the expression `výraz` are highlighted in blue boxes. A blue arrow points from the expression `výraz` to the variable `lavaStrana`, indicating the requirement for type compatibility between the right-hand side expression and the left-hand side variable.

# Typová kompatibilita – konverzie

- prvý stĺpec – cieľový typ konverzie
- prvý riadok – zdrojový typ konverzie

operandy	byte	short	int	long	float	double
byte	A	N	N	N	N	N
short	A	A	N	N	N	N
int	A	A	A	N	N	N
long	A	A	A	A	N	N
float	A	A	A	A	A	N
double	A	A	A	A	A	A

# Typová kompatibilita príklad

```
private int hodnotaInt;  
private byte hodnotaByte;  
private double hodnotaDouble;
```

```
this.hodnotaInt = 5;
```

```
this.hodnotaInt = this.hodnotaByte;
```

```
this.hodnotaInt = this.hodnotaDouble;
```

# Identifikátor

- názov, pomenovanie inštancie, triedy, správy, metódy a atribútu sa nazýva identifikátor
- identifikátor je jedno alebo viacslovné pomenovanie

# Návrh identifikátorov<sub>(1)</sub>

- identifikátory navrhujeme, volíme tak, aby vyjadrovali význam, zmysel pojmu
- nepoužívame neznáme skratky
- programy sú síce určené na to, aby ich vykonávali počítače, ale čítajú ich aj ľudia
- úspešný program sa stále nejako mení

# Návrh identifikátorov<sub>(2)</sub>

- pre tvorbu identifikátorov platia
  - pravidlá – určuje programovací jazyk
  - konvencie – určuje programátorská komunita



# Pravidlá tvorby identifikátorov

- programovací jazyk definuje pravidlá, ktoré musí spĺňať každý identifikátor
- Java definuje tieto pravidlá
  - môže sa skladať z písmen, číslic a znakov „\_“ (podčiarkovník) a „\$“ (dolár)
  - nesmie začínať číslicou
  - rozlišujú sa malé a VEĽKÉ písmená
  - nesmie sa zhodovať so žiadnym kľúčovým slovom
    - kľúčové slovo – slovo alebo identifikátor so špecifickým významom v programovacom jazyku

# Konvencie pre tvorbu identifikátorov

- jednotlivé slová sa píšú bez medzier
- prvé slovo sa píše malým písmenom
- druhé a ďalšie slová začínajú veľkým písmenom
- výnimka: identifikátor triedy vždy začína veľkým písmenom
- príklady
  - trieda: AutomatMHD
  - metóda: tlacListok

# Algoritmy a AutomatMHD

- telo konštruktora obsahuje tie príkazy, ktoré predstavujú inicializáciu práve vytvoreného automatu
- telá metód obsahujú tie príkazy, ktoré urobí automat ako reakciu na prijatie rovnomennej správy
- telá predstavujú [algoritmy](#)

**A**

ALGORITHM (NOUN)  
WORD USED BY  
PROGRAMMERS WHEN  
THEY DO NOT WANT TO  
EXPLAIN WHAT THEY DID.

zdroj: <https://www.facebook.com/ProgrammersCreateLife/photos/a.241809332534619.55240.241806149201604/1134445459937664/?type=3&theater>

# Algoritmus

- algoritmus
  - popis pracovného postupu, ktorým sa rieši určitá skupina úloh.
  - presne definovaná konečná postupnosť príkazov (krokov), vykonávaním ktorých pre každé prípustné vstupné hodnoty získame po konečnom počte krokov odpovedajúce výstupné hodnoty.
- algoritmizácia
  - tvorivý proces hľadania a vytvárania algoritmu.

# Vlastnosti algoritmu

- determinovanosť – po vykonaní každého kroku musí byť jednoznačne určený krok nasledujúci
- rezultatívnosť – pre rovnaké vstupné údaje musí algoritmus dať rovnaké výstupné údaje
- konečnosť – vykonávanie algoritmu má vždy konečný počet krokov
- hromadnosť – nie je riešením jedinej úlohy, ale všetky úlohy danej kategórie, ktoré sa líšia len hodnotami vstupných údajov

# Procesor

- algoritmus vykonáva objekt, ktorý sa nazýva procesor.
- algoritmus musí byť vyjadrený v jazyku, ktorému procesor rozumie a vie vykonávať príkazy zapísané pomocou toho jazyka.
- predpokladom je, že procesor „neuvažuje“ – príkazy algoritmu vykonáva mechanicky.

# Vyjadrenie algoritmu

- postup vyjadrený v prirodzenom jazyku – procesorom je človek.
- znázornený graficky v podobe diagramu – používa najčastejšie človek, keď sa chce vyjadriť nezávisle od prirodzeného jazyka autora.
- zapísaný v programovacom jazyku – ak je procesorom počítač.



# Štruktúrované programovanie

- doteraz – všeobecne o algoritmoch
- ich zápis môže mať rôzne formy v rôznych programovacích jazykoch
- odteraz len štruktúrované programovanie
- podporuje ho aj Java

# Základné konštrukčné prvky

- prvkami sú príkazy
- jednoduché
  - priradovací príkaz
  - príkaz návratu
- štruktúrované
  - postupnosť (sekvencia)
  - vetvenie (alternatíva )
  - cyklus

# Štruktúrované príkazy

- postupnosť (sekvencia)
  - určuje poradie vykonávania príkazov
  - príkazy môžu byť jednoduché aj štruktúrované
- vetvenie – výber jednej alternatívy
  - alternatívne vetvy algoritmu
  - vetva sa uplatní, ak je splnená podmienka
- cyklus – opakovanie časti algoritmu
  - opakovaná časť – telo cyklu
  - opakovanie na základe podmienky cyklu

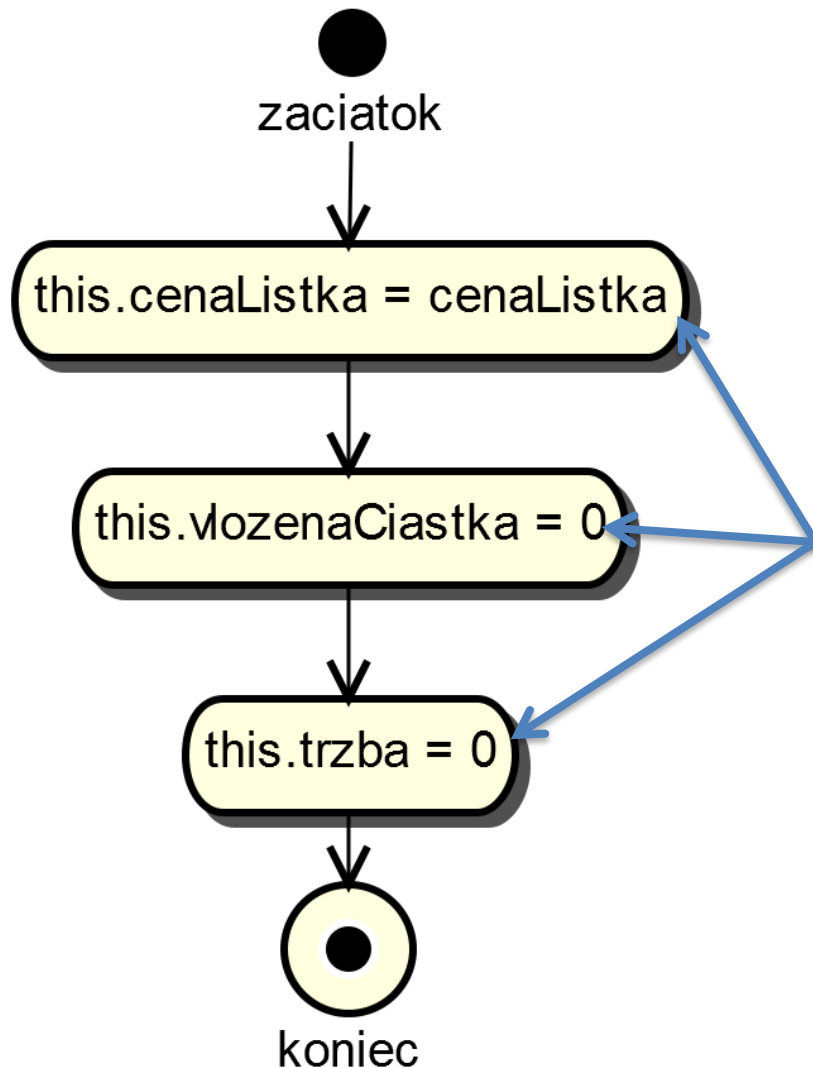
# Primitívny automat MHD

- telá konštruktora a všetkých metód sú postupnosti (sekvencie) jednoduchých príkazov.
- príklad – konštruktor
- telo konštruktora tvoria 3 príkazy – nastavenie automatu do začiatočného stavu.

# Java – postupnosť príkazov

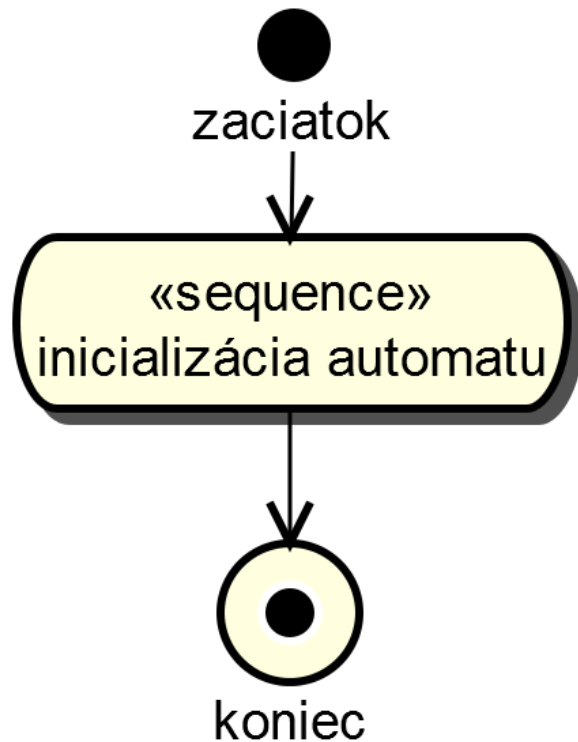
```
this.cenaListka = cenaListka;  
this.vlozenaCiastka = 0;  
this.trzba = 0;
```

# Diagram aktivít UML – postupnosť<sub>(1)</sub>



aktivita = jednoduchý príkaz

# Diagram aktivít UML – postupnosť<sub>(2)</sub>



← aktivita = príkaz – sekvencia

# Nedostatky prvej verzie automatu

1. chýba kontrola, či vložená čiastka pokrýva cenu lístka
2. chýba kontrola, či je zadaná cena lístka kladná
3. automat nevráti preplatok, ak sa vloží väčšia čiastka, ako je cena lístka
4. chýba kontrola, či je hodnota vloženej mince kladná
5. automat vydáva len lístky jednej ceny

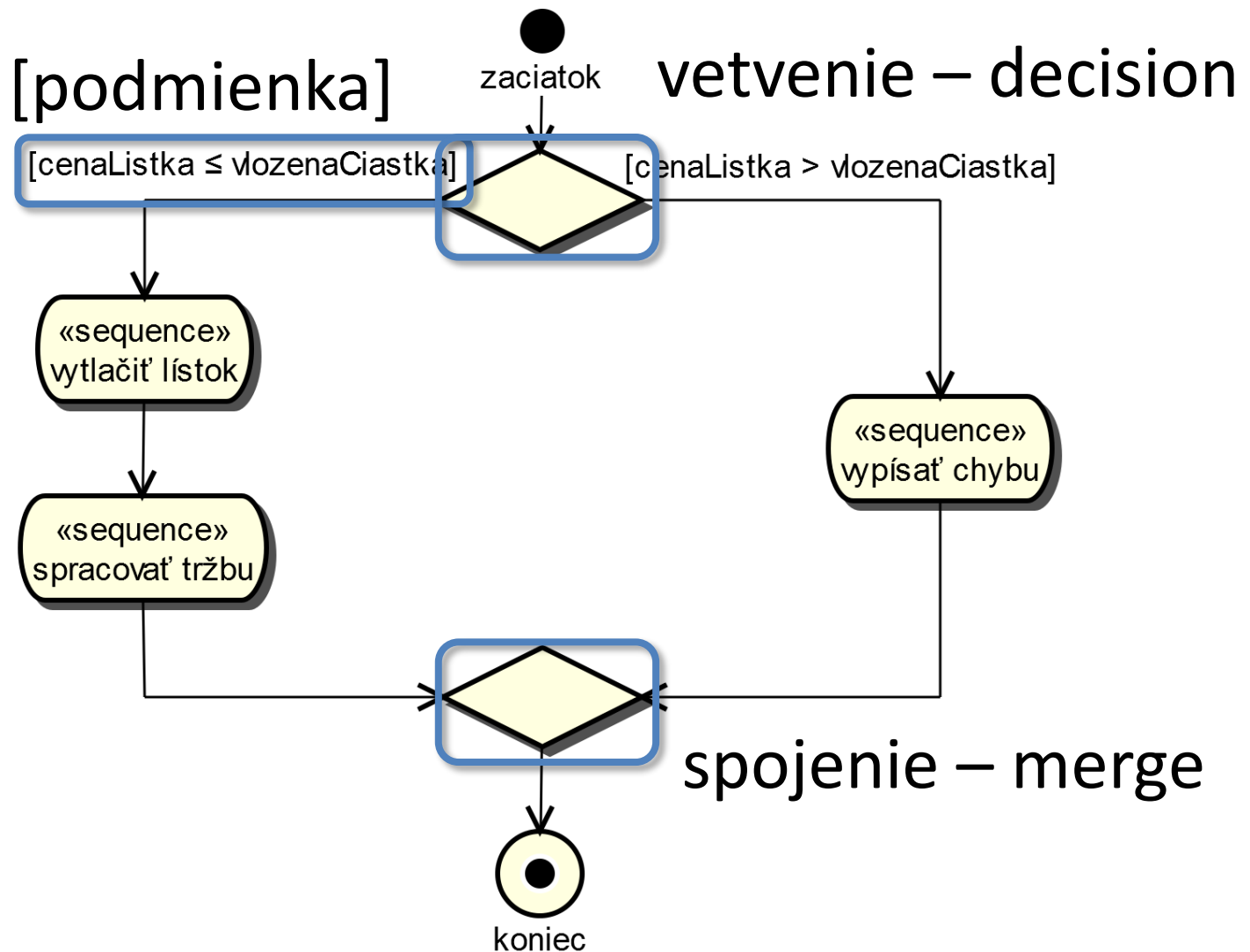


# Riešenie nedostatkov – 1

*chýba kontrola, či vložená čiastka pokrýva cenu lístka*

- treba sa rozhodnúť
  - ak vložená čiastka dosahuje cenu lístka
    - lístok sa vytlačí
  - ak vložená čiastka nedosahuje cenu lístka
    - lístok sa nevytlačí, vypíše sa dôvod
- riešenie: vetvenie

# Vetvenie v diagrame aktivít



# Telo metódy tlačListok

```
if (this.cenaListka <= this.vlozenaCiastka) {  
    // tlač listka – vynechane prikazy  
    this.trzba = this.trzba + this.cenaListka;  
    this.vlozenaCiastka = this.vlozenaCiastka  
                                – this.cenaListka;  
} else {  
    System.out.println("Ciastka je mensia ako cena.");  
}
```

# Vetvenie v jazyku Java – príkaz if

- úplný príkaz if

```
if (podmienka) {  
    // príkazy vetvy, ak podmienka platí (true)  
} else {  
    // príkazy vetvy, ak podmienka neplatí (false)  
}  
// príkazy, ktoré sa vykonajú vždy
```

# Vetvenie v jazyku Java – príkaz if

- neúplný príkaz if

```
if (podmienka) {  
    // príkazy vetvy, ak podmienka platí (true)  
}  
// príkazy, ktoré sa vykonajú vždy
```

# Formulácia podmienky

- podobná matematickej forme
- logický výraz – jeho hodnota je typu boolean
- príklad:

```
this.hodnotaMince > 0
```

- > (väčší ako) – relačný operátor
- ak je relácia splnená, výsledok má hodnotu true
- inak má hodnotu false

# Relačné operátory

- sú vždy binárne

prvýOperand operátor druhýOperand

- oba operandy sú aritmetické výrazy
- priorita relačných operátorov je nižšia ako priorita aritmetických operátorov
- výsledok logického výrazu je vždy typu boolean

# Relačné operátory

matematika	Java
$x > y$	<code>x &gt; y</code>
$x \geq y$	<code>x &gt;= y</code>
$x < y$	<code>x &lt; y</code>
$x \leq y$	<code>x &lt;= y</code>
$x = y$	<code>x == y</code>
$x \neq y$	<code>x != y</code>



- časť kódu programu uzavretá do dvojice zátvoriek {}
- bloky sa do seba vnášajú
- telo triedy obsahuje bloky – telá konštruktorov a metód
- vetva v príkaze if tiež môže byť blok

# Príkaz if bez bloku

**if** (podmienka)

// jeden príkaz – podmienka platí (true)

**else**

// jeden príkaz – podmienka neplatí (false)

// tu už začína nasledujúci príkaz

**if** (podmienka)

// jeden príkaz – podmienka platí (true)

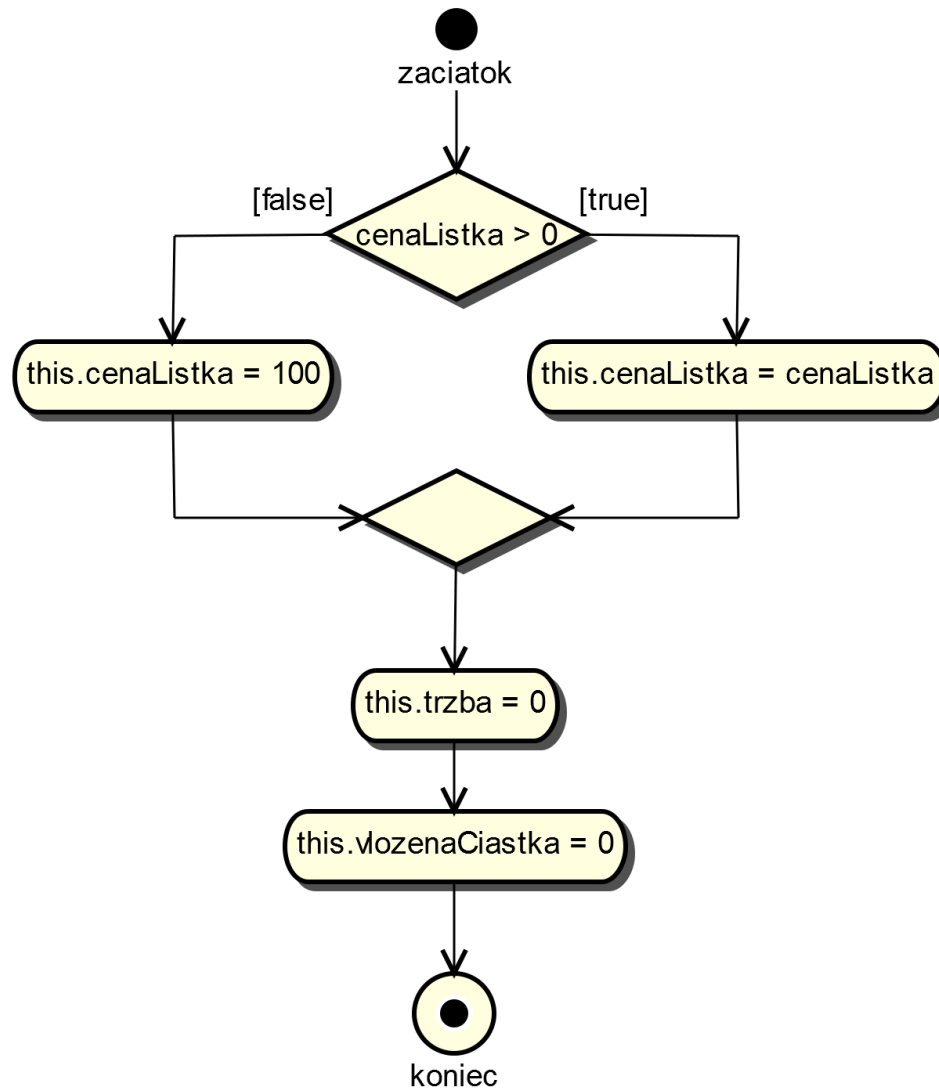
// tu už začína nasledujúci príkaz

# Riešenie nedostatkov – 2

*chýba kontrola, či je zadaná cena lístka kladná*

- použijeme vetvenie
  - 1. vetva
    - podmienka: cena lístka  $> 0$
    - cena sa použije na inicializáciu atribútu
  - 2. vetva
    - podmienka: cena lístka  $\leq 0$
    - ???
    - atribút sa inicializuje preddefinovanou hodnotou

# Diagram aktivít – konštruktor



# Telo konštruktora AutomatMHD

```
if (cenaListka > 0) {  
    this.cenaListka = cenaListka;  
} else {  
    this.cenaListka = 100;  
}  
this.vlozenaCiastka = 0;  
this.trzba = 0;
```

# Riešenie nedostatkov – 3

*automat nevráti preplatok, ak sa vloží väčšia čiastka, ako je cena lístka*

- rozhranie automatu rozšírime o správu vratZostatok
- triedu doplníme o novú metódu

# Automat vráti zostatok<sub>(1)</sub>

- ako automat pozná, koľko ma vrátiť?
- aVlozenaCiastka
- je metóda správna?

```
public int vratZostatok() {  
    return this.vlozenaCiastka;  
}
```

- treba nulovať vlozenaCiastka.

# Automat vráti zostatok<sub>(2)</sub>

- je teraz metóda správna?

```
public int vratZostatok() {  
    return this.vlozenaCiastka;  
    this.vlozenaCiastka = 0;  
}
```

- preklad je s chybou
- nulovanie sa nemôže vykonať



# Automat vráti zostatok<sub>(3)</sub>

- je teraz metóda správna?

```
public int vratZostatok() {  
    this.vlozenaCiastka = 0;  
    return this.vlozenaCiastka;  
}
```

- preklad je bez chýb
- automat nevráti nič
  - metóda vráti nulu

# Automat vráti zostatok<sub>(4)</sub>

```
public int vratZostatok() {  
    int zostatok;  
    zostatok = this.vlozenaCiastka;  
    this.vlozenaCiastka = 0;  
    return zostatok;  
}
```

# Lokálna premenná

- definícia lokálnej premennej

```
int zostatok;
```

- definícia a inicializácia lokálnej premennej

```
int zostatok = this.vlozenaCiastka;
```

- rovnako ako

```
int zostatok;  
zostatok = this.vlozenaCiastka;
```

# Premenná – spoločné vlastnosti

- atribúty, parametre a lokálne premenné sú miesta v pamäti, v ktorých sú uložené hodnoty
- atribúty, parametre a lokálne premenné majú vždy definovaný typ hodnoty, ktorú uchovávajú
- pokiaľ budeme o atribútoch, parametroch a lokálnych premenných hovoriť všeobecne, budeme hovoriť o premenných

# Atribúty, parametre a lokálne premenné<sub>(1)</sub>

- účel, poslanie
  - atribút – uchováva stav objektu
  - parameter – prenos spresňujúcej informácie
  - lokálna premenná – dočasné uloženie určitej hodnoty v rámci bloku

# Atribúty, parametre a lokálne premenné<sub>(2)</sub>

- definícia
  - atribút – telo triedy, mimo konštruktorov a metód
  - parameter – hlavička konšuktora alebo metódy
  - lokálna premenná – v bloku (v tele konšuktora alebo metódy)

# Atribúty, parametre a lokálne premenné<sub>(3)</sub>

- inicializácia
  - atribút – v konštruktore v čase vytvárania inštancie
  - parameter – v odosielanej správe ako skutočný parameter
  - lokálna premenná – v bloku (v tele konštruktora alebo metódy)

# Atribúty, parametre a lokálne premenné<sub>(4)</sub>

- rozsah platnosti (viditeľnosť, použiteľnosť)
  - atribút – v každom konštruktore alebo metóde
  - parameter – v tele daného konštruktora alebo metódy
  - lokálna premenná – v bloku od miesta definície po koniec bloku definície aj vo všetkých vnorených blokoch.



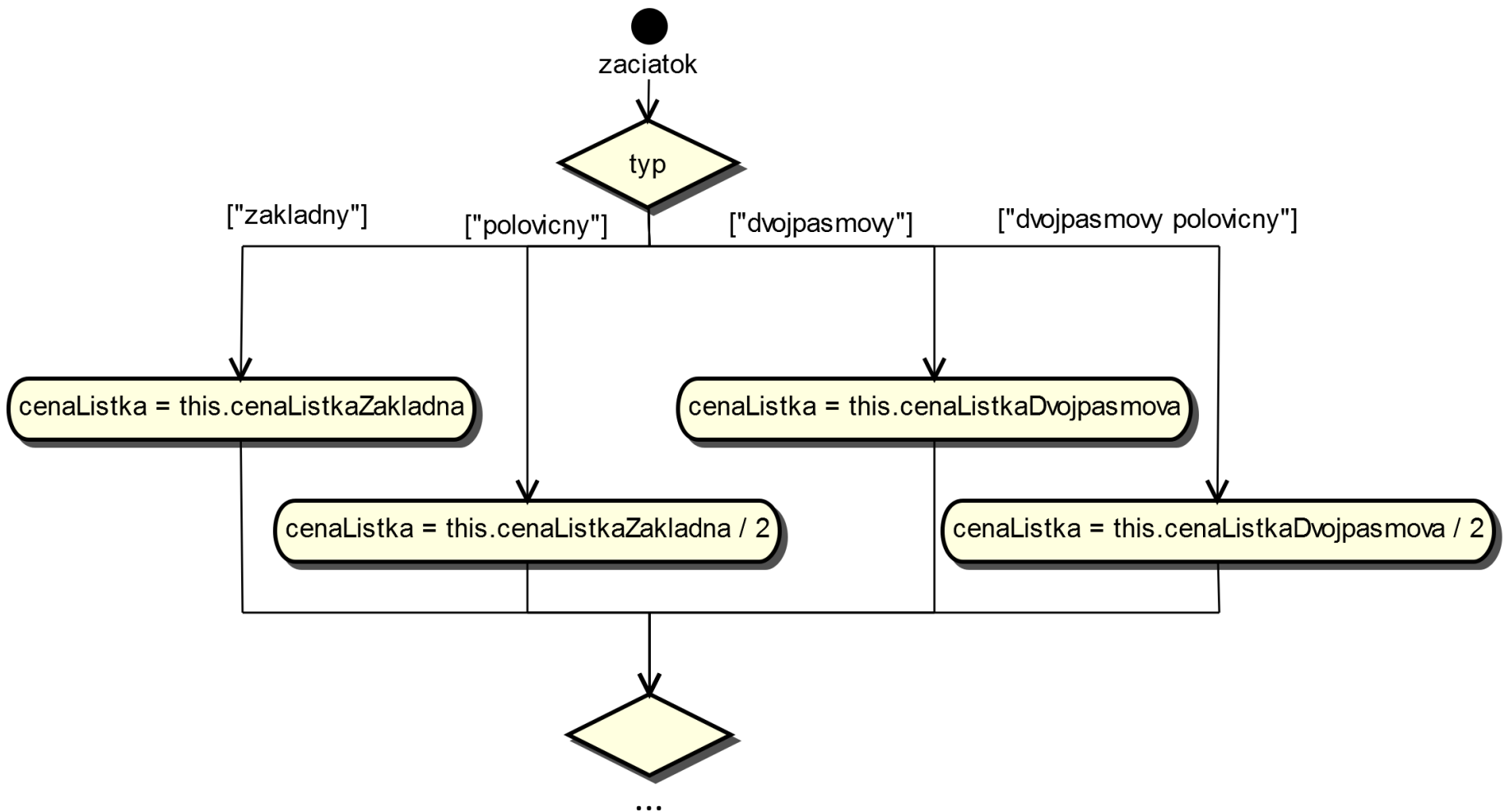
- životný cyklus – existencia
  - atribút – životný cyklus inštancie
  - parameter – v čase vykonávania konštruktora alebo metódy
  - lokálna premenná – v čase vykonávania bloku od miesta definície po koniec bloku, v ktorom bola definovaná

# Riešenie nedostatkov – 5

*automat vydáva len lístky jednej ceny*

- chceme napr. polovičné, alebo dvoj pásmové
- rozhodovanie medzi viac možnosťami
- viaccestné vetvenie

# Získanie ceny lístka<sub>(1)</sub>



# Získanie ceny lístka<sub>(2)</sub>

```
public void tlacListok(String typ) {  
    int cenaListka;  
    switch (typ) {  
        case "zakladny":  
            cenaListka = this.cenaListkaZakladna;  
            break;  
        case "studentsky":  
        case "polovicny":  
            cenaListka = this.cenaListkaZakladna / 2;  
            break;
```

...

# Získanie ceny lístka<sub>(2)</sub>

...

```
case "dvojpasmovy":
```

```
    cenaListka = this.cenaListkaDvojpasmove;
```

```
    break;
```

```
case "studentsky dvojpasmovy":
```

```
case "polovicny dvojpasmovy":
```

```
    cenaListka = this.cenaListkaDvojpasmove / 2;
```

```
    break;
```

```
default:
```

```
    System.out.println("Nespravny typ listka.");
```

```
    return;
```

```
}
```

...

# Príkaz switch

- príkaz switch – viaccestné vetvenie

```
switch (výraz) {  
    // možnosti a vetvy  
}
```

- výraz
  - musí byť buď celočíselný výraz
    - Java 7 – aj reťazce
  - porovnáva s možnosťami

# Príkaz switch – case (návestie)

- možnosti musia byť konštantné celočíselné výrazy
  - konvertovateľné na typ výrazu v príkaze switch
  - Java 7 – aj reťazce

```
case moznost1:
```

```
case moznost2:
```

```
...
```

```
// príkazy vetvy
```

# Ukončenie vetvy

- každá vetva má byť ukončená
  - príkaz [return](#) – ukončenie vykonávania metódy
  - príkaz [break](#) – ukončenie vykonávania príkazu switch
  - chýbajúce ukončenie
    - prekladač jazyka Java [neupozorní](#)
    - vykonávanie pokračuje ďalšou vetvou



# Príkaz switch – default (návestie)

- default – vetva, ktorá sa uplatní, ak sa nenájde príslušná možnosť
- obdoba else v príkaze if

default:

// príkazy vetvy

# Vďaka za pozornosť