6. Programování v jazyce JAVA: vlastnosti a koncepce jazyka. Principy objektového programování.

PREVIOUS

Chapter 1 Objektové programování

Objektově orientované programování (zkracováno na OOP) je metodika vývoje softwaru, založená na následujících myšlenkách, koncepci:

1. Objekty – jednotlivé prvky modelované reality (jak data, tak související funkčnost) jsou v programu seskupeny do entit, nazývaných objekty. Objekty si pamatují svůj stav a navenek poskytují operace (přístupné jako metody pro volání).
2. Abstrakce – programátor, potažmo program, který vytváří, může abstrahovat od některých detailů práce jednotlivých objektů. Každý objekt pracuje jako černá skříňka, která dokáže provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje.t
3. **Zapouzdření** – zaručuje, že objekt nemůže přímo přistupovat k „vnitřnostem“ jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek zpřístupňuje rozhraní, pomocí kterého (a nijak jinak) se s objektem pracuje.
4. Skládání – Objekt může obsahovat jiné objekty.
5. Delegování – Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.
6. **Dědičnost** – objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).
7. **Polymorfismus** – odkazovaný objekt se chová podle toho, jaké třídy je instancí. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným

způsobem, ale jejich konkrétní chování se liší podle implementace. U polymorfismu podmíněného dědičností to znamená, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy, neboť rozhraní třídy je podmnožinou rozhraní podtřídy. U polymorfismu nepodmíněného dědičností je dostačující, jestliže se rozhraní (nebo jejich požadované části) u různých tříd shodují, pak jsou vzájemně polymorfní.

Objektový přístup programování trochu jinak:

- Modelování problému jako systému spolupracujících tříd
- Třída modeluje jeden koncept
- Třídy umožní generování instancí, objektů příslušné třídy
- Jednotlivé objekty spolu spolupracují, „posílají si zprávy“,
- Třída je „vzorem“ pro strukturu a vlastnosti generovaných objektů
- Každý objekt je charakteristický specifickými hodnotami svých atributů a společnými vlastnostmi třídy

1.1 Třída

Třída je základem uživatelského programu v jazyku Java (nebo obecně v OOP). Třída = koncept. Každá třída je reprezentována svými prvky, objekty dané třídy. Každá třída je charakterizována svými vlastnostmi, svými funkčními možnostmi a svými parametry. Třída tedy popise nějaký “objekt” reálného problému a udržuje jeho parametry a poskytuje metody pro práci s ním. Třída v bodech:

- musí být deklarována hlavní funkce main (NOTE: bylo v přednášce, ale není to pravda, třídy bez třídy main se běžně používá, např. jako knihovna). Funkce main musí být statická, volá se ještě před vytvořením nějaké instance, slouží pro spuštění programu, testování.
- jsou deklarovány další (static) funkce (či procedury) třídy, případně i v jiných třídách: projekt, package
- mohou být deklarovány statické proměnné, které jsou použitelné jako nelokální proměnné ve funkcích dané třídy
- program probíhá spuštěním příkazů metody main

Třída je definována množinou identifikátorů, které mají třídou definovaný význam:
– data: proměnné, konstanty (členské proměnné, datové složky, atributy)
– metody: pracovně funkce a procedury

1.1.1 Třída = předpis vs. Objekt = instance třídy

Třída jako šablona pro generování konkrétních instancí třídy, tzv. objektů. Jednotlivé instance třídy (objekty) mají stejné metody, ale nacházejí se v různých stavech – Stav objektu je určen hodnotami instančních, členských proměnných. Schopnosti

objektu jsou dány instančními metodami třídy. V jazyku Java lze objekty (instance tříd) vytvářet pouze dynamicky pomocí operátoru new a přistupovat k nim pomocí referenčních proměnných (podobně jako u pole). Třída bez vytvořené instance (objektu) může „pracovat“ pouze „staticky“ (mohou být použity jen její statické metody či proměnné – procedurální přístup). Statické proměnné = společné pro všechny instance třídy.

1.1.2 Konstruktor třídy

Konstruktory = speciální metody pro generování instancí tříd, konkrétních objektů.

- vytvoří objekt
- případně nastaví vlastnosti objektu
- jméno konstruktoru je totožné se jménem třídy (jediná metoda začínající velkým písmenem)
- volání pomocí operátoru new, např. malyObdelnik = new Obdelnik(2,5);
- neosahuje návratový typ - nic nevrací, vytváří objekt
- není-li konstruktor vytvořen, je vygenerován implicitní konstruktor s prázdným seznamem parametrů – je-li konstruktor deklarován, implicitní zaniká

1.1.2.1 Přetěžování (nejen konstruktoru třídy)

Umožňuje objektům volání jedné metody se stejným jménem, ale s jinou implementací. Provádí se to tak, že se deklaruje více metod se stejným názvem, které se mohou lišit různým počtem, typem argumentů popř. jejich pořadím. Umožňuje volat vytvoření instance s různými parametry - různým počtem i typem parametrů.
Př.:

```
public class Obdelnik {
// vsechny konstruktory lze pouzit zvenci pro vytvoreni objekt
u
// kontruktor pro volani se vsemi tremi vlastnostmi,
Obdelnik(int s, int v, Color c){
sirka = s;
vyska = v;
barva = c;
}
// kontruktor pouze s rozmery - vyuziva prvni konstruktor pomo
ci this
Obdelnik(int s, int v){
this(s,v,Color.BLACK);
```

```
}
// prazdny konstruktor
Obdelnik() {
this(0,0,Color.BLACK);
}
}
```

Konstruktor dále:

- Jméno konstruktoru je totožné se jménem třídy
- Konstruktor nemá návratovou hodnotu (ani void)
- Předčasně lze ukončit činnost konstruktoru return
- Konstruktor má parametrovou část jako metoda - může mít libovolný počet a typ parametrů
- V těle konstruktoru použít operátor this, odkaz na příslušný konstruktor s týmž počtem, pořadím a typem parametrů - nepíše se jméno třídy
- Konstruktor je zpravidla vždy public (!) – //třída java.lang.Math jej má private – proč? Protože je designována jako utilitní třída, která poskytuje statické metody (a atributy) a nevyžaduje vytvoření instance

1.1.3 Třída jako datový typ

Příkladem třídy jako datového typu je třída Complex

- hodnotami typu jsou komplexní čísla tvořená dvojicemi čísel typu double (reálná a imaginární část)
- množinu operací tvoří obvyklé operace nad komplexními čísly (absolutní hodnota, sčítání, odčítání, násobení a dělení)

1.1.4 Statické vs. Instanční metody

Třída může definovat dva druhy metod:

- statické metody – metody třídy, procedury a funkce
- instanční metody – metody objektů

Metody obou druhů mohou mít parametry a mohou vracet výsledek nějakého typu. Statická metoda označuje operaci (dílčí algoritmus, řešení dílčího podproblému), jejíž vyvolání (provedení) obsahuje jméno třídy, jméno metody a seznam skutečných parametrů jméno_třídy.jméno_metody(seznam skutečných parametrů). Statickým metodám třídy odpovídají v jiných jazycích procedury (nevracejí žádnou hodnotu) a funkce (vracejí hodnotu nějakého typu). Instanční metoda označuje operaci nad objektem (instancí (!)) dané třídy, jejíž vyvolání obsahuje referenční proměnnou objektu, jméno metody a seznam skutečných parametrů referenční_proměnná.jméno_metody(seznam skut. parametrů).

- Statickým metodám třídy budeme i nadále říkat procedury a funkce
- Instančním metodám budeme zkráceně říkat metody

1.1.5 Operátor this

Každý objekt má implicitní operátor this, který obsahuje odkaz na "svou" instanci

- Hodnotou operátoru this je odkaz na objekt pro který byla metoda zavolána (implicitní parametr odkazu na objekt)
- Umožňuje přístup k vlastním instančním proměnným v instančních metodách
- Používá se v přetížených konstruktorech

Podobně funguje operátor this i pro metody:

- pokud je instanční metoda volána z jiné instanční metody té samé třídy, potom se volá pomocí operátoru this (operátor se může vynechat)
- pokud se volá metoda z jiného kontextu, uvádí se před jejím jménem přístup k příslušné instanci (tečka notace) např. předané parametrem
- nelze volat ze statické metody - u ní by nebylo jasné, kam this odkazuje

1.2 Objekt

Objekt - datový prvek, instance třídy, dynamicky vytvořen podle „vzoru“- třídy (viz sekce Třída). Objekt si pamatuje svůj stav (v podobě dat čili atributů) a zveřejněním některých svých operací (nazývaných metody) poskytuje rozhraní, jak s ním pracovat. Objekt je strukturován tzn. skládá se z jednotlivých položek tzv. atributů. Objekt = heterogenní objekt skládající se z položek různého typu (x pole, pole se skládá z položek stejného typu). Složky objektu:

- atributy objektu (datové složky, členské proměnné (member variables)) - proměnné objektu, definují typ a jména vlastností objektu
- metody

Hodnota objektu je strukturovaná, tzn. skládá se dílčích hodnot, které mohou být obecně různého typu (heterogenní datová struktura – na rozdíl od pole). Objekt je tedy abstrakcí paměťového místa skládajícího se z částí, ve kterých jsou uloženy dílčí hodnoty - nazývají se položkami objektu (složkami, atributy, instančními proměnnými, fields, attributes). Položky objektu jsou označeny jmény, která mohou (ale nemusí) být třídou zveřejněna, zásadně se nezveřejňují. Pro manipulaci se zavádějí settery, gettery.

1.3 Zapouzdření

Zapouzdření v objektech znamená, že k obsahu objektu se nedostane nikdo jiný, než sám vlastník. Navenek se objekt projeví jen svým rozhraním (operacemi, metodami) a

komunikačním protokolem. (Př. private proměnné -> pro manipulaci metody: nejjednodušší např. settery, gettery). Důležité pojmy:

- **Skládání objektů:** udržování odkazů na jiné objekty (objekt obsahuje jiné objekty).
- **Delegování:** Využívání služeb jiných objektů.

Zapoudření = Daný stav objektu je přístupný nebo měnitelný pouze prostřednictvím rozhraní poskytovaného objektem. Důsledkem zapouzdření je autorizovaný přístup k datům, při kterém zajistíme, že s daty objektu nebude možné z vnějšku třídy manipulovat jinak než pomocí metod této třídy, které tvoří komunikační rozhraní třídy. Zapouzdření je zajištěno pomocí modifikátorů:

1. public - lze je volat odkudkoli
2. protected - lze volat ze stejného package nebo odovozené třídy
3. neurčený - lze volat ze stejného package; nelze volat z odvozené třídy ležící v jiném package
4. private - lze je volat pouze z metod téže třídy

1.4 Dědičnost

Dědičnost je mechanismus umožňující

- rozšiřovat datové položky tříd (také modifikovat)
- rozšiřovat či modifikovat metody tříd

Dědičnost umožní

- vytvářet hierarchie tříd
- „předávat“ datové položky a metody k rozšíření a úpravě
- specializovat, „upřesňovat“ třídy

Dvě základní výhody dědění

- Dědění má praktický význam v znuvupoužitelnosti programového kódu
- Dědičnost je základem polymorfismu

Java dědí pouze od jediného předka. Mnohonásobné dědění se řeší pomocí rozhraní. Nejvyšší třída je Object, všechny třídy dědí třídu Object (metody: equals (standardní implementace neporovnává objekty, ale reference), toString, hashCode (stejné objekty by měly generovat stejný hashCode), clone..)

1.4.1 Příklad: Obdélník je případ kvádru (ne naopak)

Obdélník je „kvádrem“ s nulovou hloubkou

- Potomek se deklaruje pomocí klauzule extends
- Obdelnik převezme proměnné sirka, vyska, hloubka metody hodnotaSirky, delkaUhlopricky
- Konstruktor se dědí, parametr hloubka se nastaví do nuly

- Objekty Obdelnik mohou využívat proměnné sirka, vyska a hloubka, metody hodnotaSirky a delkaUhlopricky

```
public class Kvadr2 {
public int sirka;
public int vyska;
public int hloubka;
public Kvadr2(int sirka, int vyska, int hloubka) {
this.hloubka = hloubka;
this.sirka = sirka;
this.vyska = vyska;
}
public int hodnotaSirky() {
return sirka;
}
public double delkaUhlopricky() {
double pom = (sirka * sirka) + (vyska*vyska) + (hloubka * hlou
bka);
return Math.sqrt(pom);
}
}
class Obdelnik2 extends Kvadr2{
public Obdelnik2(int sirka, int vyska) {
super(sirka, vyska, 0);
}
}
```

1.4.2 Hiearchie tříd

Třída Tpod, která je podtřídou třídy Tnad, dědí vlastnosti nadtřídy Tnad a rozšiřuje je o nové vlastnosti; některé zděděné vlastnosti mohou být v podtřídě modifikovány. Pro instanční metody to znamená:

1. každá metoda třídy Tnad je i metodou třídy Tpod, v podtřídě však může mít jinou implementaci (může být zastíněna - override = Podtřída obsahuje metodu se

stejným názvem i stejnými parametry, metody nadtřídy zastíní vlastní implementací. Př. typicky toString())
2. v podtřídě mohou být definovány nové metody

Pro strukturu objektu to znamená:

- instance třídy Tpod mají všechny členy třídy Tnad a případně další

Pro referenční proměnné to znamená:

- proměnné typu Tnad může být přiřazena reference na objekt typu Tpod
- na objekt referencovaný proměnnou typu Tnad lze vyvolat pouze metodu deklarovanou ve třídě Tnad; jde-li však o objekt typu Tpod, metoda se provede tak, jak je dáno třídou Tpod
- hodnotu referenční proměnné typu Tnad lze přiřadit referenční proměnné typu Tpod pouze s použitím přetypování, které zkontroluje, zda referencovaný objekt je typu Tpod

Vztah nadtřída – podtřída je tranzitivní = jestliže je x nad třídou y a y je nad třídou z, pak je x nadtřídou z

1.5 Kompozice

Obsahuje-li deklarace třídy členské proměnné objektového typu, pak mluvíme o kompozici objektů. Kompozice vytváří hierarchii objektů, nikoli však dědičnost. Například třída, která obsahuje jako členskou proměnnou integer, ale hlavně i například Datum (další, jiný objekt). (Kompozice označována jako struktura HAS ("má", dědičnost jako ISE "je"). Kompozice objektů je tvořena atributy objektového typu, pouze je skládá

1.6 Polymorfismus

V programovacím jazyce se jedná o možnost volat stejné metody u různých objektů, aniž bychom věděli, jakého přesně jsou typu. Navíc může mít stejná metoda u různých objektů odlišný význam. To je možné díky tomu, že vždy známe společného předka těchto různých objektů. Tím může být třída, abstraktní třída nebo rozhraní.

```
// soubor Osoba.java
public class Osoba {
public int fce () {
return 10;
}
}
// soubor Zamestnanec.java
```

```
public class Zamestnanec extends Osoba {
public int fce () {
return 20;
}
}
// jiny soubor
Zamestnanec a = new Zamestnanec();
Osoba b = new Zamestnanec();
Osoba c = new Osoba();
System.out.println(a.fce()); // vypíše 20
System.out.println(b.fce()); // vypíše 20
System.out.println(c.fce()); // vypíše 10
```

1.7 Abstraktní třídy

V některých situacích je výhodné vytvořit jedinou bázovou třídu pro více tříd odpovídajících konkrétním objektům, i když tato samotná bázová třída žádnému konkrétnímu objektu neodpovídá. Může ovšem nést některá data a poskytovat metody, které jsou odvozeným třídám společné. Taková třída se pak nazývá abstraktní a je označena klíčovým slovem abstract. Překladač jazyka Java pak zajistí, že instanci abstraktní třídy nelze operátorem new přímo vytvořit, mohou se vytvářet pouze instance konkrétních tříd.
Abstraktní třída může deklarovat některé společné metody a poskytovat jejich základní implementaci. Pokud odvozená třída takovou metodu nepředefinuje, pak se pro její instance použije implementace poskytnutá v bázové třídě.
Mohou však nastat i situace, kdy skutečně vyžadujeme, aby odvozené třídy určitou metodu vždy definovaly. Takovou metodu pak také nazýváme abstraktní a označujeme klíčovým slovem abstract, navíc u ní není uvedeno tělo a hlavička metody je zakončena středníkem. Pokud odvozená třída některou abstraktní metodu neimplementuje, musí být také označena jako abstraktní. Tím je zajištěno, že instance konkrétních tříd mají všechny metody implementované.

```
abstract class Obrazec {
public Obrazec(double x, double y) {
this.x = x;
```

```
this.y = y;
}
public abstract double obvod();
public abstract double obsah();
protected double x;
protected double y;
}
```

1.8 Rozhraní

Druhou velmi podobnou konstrukcí jsou rozhraní (klíčové slovo interface). Základním rozdílem je, že interface obsahuje pouze konstanty a metody bez těla (v tomto případě je neoznačujeme slovem abstract). Rozhraní je kontrakt, který specifikuje operace, které má třída splňovat, a který se již nezabývá tím, jak toho bude konkrétně dosaženo.
Velkou výhodou rozhraní oproti abstraktním třídám je, že každá třída může implementovat až mnoho rozhraní, avšak vždy maximálně jednu třídu.
Zatímco pro dědění tříd (a dědení interfaců mezi sebou) využíváme v hlavičce klíčové slovo extends, tak pro implementaci rozhraní používáme slovo implements.

```
public interface Visualni {
public void vykresli (Graphics kam);
}
```

1.8.1 Rozhraní jako typ proměnné

V Javě lze definovat proměnnou typu reference na rozhraní, ve které může být uložena libovolná třída, která toto rozhraní implementuje. Jména rozhraní lze používat jako referenční datové typy stejným způsobem jako jména tříd.

```
Visualni visualniObjekt = new VisualniKruznice();
visualniObjekt = new VisualniCtverec();
```

1.9 Výčtové typy - enum

Výčtové typy jsou speciální třídy zavedené pro větší bezpečí a pohodlí, v nejjednodušší variantě se definují (příklad):

```
enum Den {SUN, MON, TUE, WED, THU, FRI, SAT;}
for ( Den d : Den.values( ) )
System.out.println( d.ordinal( )+ " " +d.name( ) );
```

PREVIOUS

Chapter 1 JAVA - struktura jazyka

Většina této části PRG2 byla zpracována pro otázky pro PRG1 (5, 6, 7 -> struktura tříd a programu). Proto zde vynecháno, nebo stručně doplněno co chybělo.

1.1 Typy programovacích jazyků

- deklarativní - nepopisují jak se co má dělat, ale co má být výsledkem (př. HTML, Prolog: nepopisuje kroky výpočtu, ale fakta o problému a požadovaný výsledek)
- imperativní - zbytek (JAVA, C...), viz. imperativní programování otázka č. 5

1.2 Programovací styly

- Naivní
- Procedurální
- Objektově orientovaný
- Návrhové vzory
- SOA - service-oriented architecture

Chapter 2 JAVA GUI

(Tato kapitola = pouze stručný přehled <= není to přímo vypsáno jako otázka).
Vizuální a interaktivní komunikaci počítač-člověk podporují balíčky:

2.1 Knihovny pro GUI

Základní knihovny:

2.1.1 AWT - Abstract Window ToolKit

- první, těžké(heavyweight)
- vykreslení zajišťuje platforma – rychlejší, ne vždy vše funguje vše stejně

2.1.2 SWING

- doporučené
- nové komponenty (tree-view, list box,...),
- robustní
- Look and Feel - na platformě nezávislý a vypadá stejně na všech platformách a přitom respektuje i18n
- důsledné oddělení modelu od pohledu a řadiče

2.1.3 (SWT-Standard Widget Toolkit, Eclipse IBM)

- podobné AWT (platformově závislé vykreslení)
- mnoho rozšiřujících vlastností

2.2 Základní součásti GUI

Velmi stručně, jen pro úvod do problematiky, v zadání otázky není vypsáno.

2.2.1 Komponenty (dialogové prvky) - v knihovně javax.swing

- tlačítka, seznamy, jezdci, textová pole, zatrhávací tlačítka, rádio tlačítka, ...
- společné metody pro velikost, barvu, umístění textu, ...

2.2.2 Kontejnery (v oknech) - v knihovně javax.swing

- Kontejnery se vkládají do oken
- komponenty musí být umístěny v kontejnerech
- dva základní typy kontejnerů:
  - **JPanel** – nejjednodušší, přidělí se komponenty (také JApplet),
  - **JFrame** – složitější, ale více možností

2.2.3 Správce rozmístění (Layout Manager) - v knihovně javax.swing a java.awt

- definuje pozici komponent v kontejneru
- postupně, pevná pozice, podle mřížky, sdružování, ..
- vzhled a chování celé aplikace

```
// pr. border layout
class Okno4_1 extends JFrame{
public Okno4_1 () {
...
Container kon = getContentPane();
kon.setBackground(Color.green);
BorderLayout srb = new BorderLayout();
kon.setLayout(srb);
JButton tl1 = new JButton("Test1");
kon.add(tl1,srb.WEST);
JButton tl2 = new JButton("Test2");
kon.add(tl2,srb.EAST);
JButton tl3 = new JButton("Test3");
kon.add(tl3,srb.NORTH);
setContentPane(kon);
}
}
```

Chapter 3 Obsluha událostí

Mechanismus, který umožní zpracovávat vstupní informace programu, predávané nejčastěji přes GUI. Mechanismus reakce na akci uživatele: stisk tlačítka, zadání textu, stisk tlačítka myši, ...

1. Zpracování vstupní informace v GUI je realizováno:
   a. vysláním „události“ na jedné straně „producentem“ (producent = třída)
   b. zachycením této „události“ (událost = objekt této třídy) „posluchačem“ (posluchač = naše třída)
2. Pro každou komponentu je třeba:
   a. deklarovat typ zachycované události, kterou je zájem zpracovat
   b. určit „posluchače“, který má událost obsloužit
3. Akcí uživatele vznikne událost
   a. událost je objektem Javy!
4. Události jsou zachyceny
   a. události jsou zpracovány (obslouženy) „posluchači“ (listener) – třídami s uživatelskými metodami pro reakci na událost
   b. „posluchači“ jsou třídy, které implementují rozhraní naslouchání – musejí mít schopnost „naslouchání“

3.1 Událost

Událost je objekt, který vznikne změnou stavu zdroje - důsledek interakce uživatele střídícími grafickými elementy GUI

1. Událost vznikne:
   a. kliknutím na tlačítko
   b. stiskem klávesy
   c. posunem kurzoru, atd.
2. Události jsou produkovány tzv. producenty což jsou:
   a. tlačítka
   b. rámy
   c. ostatními grafické prvky

3.2 Zpracování události

Informace o jedné události (zdroj události, poloha kurzoru, atd.) jsou shromážděny v objektu, jehož třída určuje charakter události, napr.:

- ActionEvent ~ událost generovaná tlacítkem
- WindowEvent ~ událost generovaná oknem

- MouseEvent ~ událost generovaná myší

Všechny třídy událostí jsou **následníky třídy event** a jsou umísteny v **java.awt.event.***
Základní princip zpracování událostí:

1. Události jsou generovány zdroji událostí (jsou to objekty, které nesou informaci o události)
2. Události jsou přijímány ke zpracování posluchači událostí (to jsou opět objekty tříd s metodami schopnými událost zpracovat)
3. Zdroj události rozhoduje o tom, který posluchač má reagovat (registruje si svého posluchače)

```
class Okno extends JFrame{
public Okno (){
...
FlowLayout srb = new FlowLayout();
kon.setLayout(srb);
JButton aTlac = new JButton("Pred stiskem");
kon.add(aTlac);
// registrace posluchace
aTlac.addActionListener(new Posluchac0());
setContentPane(kon);
}
}
// posluchac
class Posluchac0 implements ActionListener {
public void actionPerformed(ActionEvent e) {
//urcen objekt události
JButton o=(JButton)e.getSource();
//reakce na udalost
o.setLabel(“Po stisku");
}
}
```

Pozn. Jedna třída může být producentem i posluchačem (addActionListener(**this**), třída zároveň i implemetuje příslušné rozhranní a potřebnou metodu). Pro zpracování události se často používá i vnitřní třída (přehlednost, efektivita kódu, zapouzdření).

```
class Okno6 extends Okno3 {
JLabel lab1;
// vnitrni trida, obsluha udalosti
class Udalost implements ActionListener {
String vypis;
```

```
public Udalost(String vypis) {
this.vypis = vypis;
}
// oblsuha udalosti
public void actionPerformed(ActionEvent e) {
lab1.setText(vypis);
}
}
Udalost aUD, nUD;
// konstruktor
Okno6() {
lab1 = new JLabel("Nazdar6");
kon.add(lab1);
aUD = new Udalost("AHOJ");
// registrace 1. posluchace
aTlac.addActionListener(aUD);
nUD = new Udalost("NAZDAR");
// registrace 2. posluchace
bbTlac.addActionListener(nUD);
}
}
Anonymní třída jako posluchač:
aTlac.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
lab.setText("AHOJ");
}
});
```

3.3 Model šíření události

1. Události jsou předávány posluchačům, které si nejprve musí producent zaregistrovat – např. metodami:
   a. addActionListener()
   b. addWindowListener()
   c. addMouseListener()
   d. ...
2. Producent vysílá jen těm posluchačům, které si sám zaregistroval.

3. Posluchač musí implementovat některé z posluchačských rozhraní (!) (schopnost naslouchat):
   a. ActionListener
   b. WindowListener
   c. MouseListener
   d. ...

3.4 Implementace modelu události

Posluchac události musí implementovat príslušné rozhraní (interface), tj. implementovat príslušné abstraktní metody rozhraní. Pro každý druh události je definována **abstraktní metoda (handler)**, která tuto událost ošetruje:

- actionPerformed
- mouseClicked
- windowClosing, atd.

Handlery jsou deklarovány v rozhraních - posluchaci:

- ActionListener
- MouseListener
- WindowListener, atd.

Předání události posluchači ve skutečnosti znamená vyvolání činnosti handleru. Objekt události je předán jako skutečný parametr handleru. Producent registruje posluchače zavoláním registrační metody:

- addActionListener
- addMouseListener
- addWindowListener, atd.

Vazba mezi producentem a posluchacem je vztah N:M. Jeden posluchac muže být registrován u více producentů, u jednoho producenta muže být registrováno více posluchačů. Událost se předá všem posluchačům, avšak pořadí zpracování není zaručeno.

3.5 Více zdrojů události - jeden posluchač

Následuje ukázka zpracování události a rozlišení producenta:

```
// rozliseni popisem zdroje
public void actionPerformed(ActionEvent e) {
String s = e.getActionCommand();
String napis = (s.equals(aTlac.getLabel()))?"PRVNI":"DRUHE";
lab.setText(napis);
}
```

```
// rozliseni objektem zdroje
public void actionPerformed(ActionEvent e) {
Object o = e.getSource();
String napis = (o.equals(aTlac)) ? "PRVNI!":"DRUHE!";
lab.setText(napis);
}
```

3.6 Jeden zdroj událost - více posluchačů

Zajistí se zkrátka vícenásobným voláním metody addActionListener(new Posluchac()) u příslušného elementu. Každá třída=posluchač má svojí metodu, která se po události provede. Pořadí provedení není určeno.

3.7 Zrušení posluchače

removeActionListener(ActionListener posluchac)

Chapter 4 Výjimky

Vyjímka je „nestandardní situace“:

1. Situace, které jsou nestandardní, či které my považujeme za nestandardní, měli bychom reagovat a mužeme a dokážeme reagovat (RuntimeException)
   a. Pokus o čtení z prázdného zásobníku EmptyStackException
   b. Dělení nulou, indexování mimo rozsah pole, špatný formát císel AritmeticException, NumberFormatException
2. Situace, na které musíme reagovat, Java nás přinutí (Exception, IOException)
   a. Odkaz na chybějící soubor FileNotFoundException
3. Chyba v hardware, závažné chyby, nemůžeme reagovat (Error), (OutOfMemoryError,UnknownError)
   a. Chyba v JVM
   b. HW chyba

Obecne chyba vzniká pri porušení sémantických omezení jazyka Java. Bezpečnostní prvek Javy: zpracování chyb a nestandardních stavů není ponecháno jen na vůli programátora! Reakce na očekávané chyby se vynucuje na úrovni překladu, při nerespektování se překlad nepodaří.

4.1 Reakce na výjimky

Chyba při provádení programu v jazyku Java nemusí znamenat ukončení programu – chybu lze ošetrit a pokracovat dál. Při vzniku výjimky je automaticky vytvořen objekt,

který nese informace o vzniklé výjimce. Mechanismus výjimek umožní přenést řízení z místa, kde výjimka vznikla do místa, kde bude zpracována. Oddělení "výkonné" části (try) od části "chybové" - catch.

4.1.1 Úplné neošetření výjimky

Žádné použití throws nebo try-catch způsobí ukončení programu - CHYBA. Java sama ohlásí při překladu, které části jsou kritické a je je třeba ošetrit. Nejsou-li, pak minimálně "vyhozením" na vyšší úroven pomocí klauzule throws, jinak nedojde k překladu.

4.1.2 Neošetření výjimky, ale předání výše

public static void main(String[] args) **throws IOException** { ... }

Závisí-li chod dalšího programu na korektní funkci metody, nemá cenu ji ošetrovat a přitom by činnost programu stejně nemohla pokračovat (proste tu hodnotu musíme mít a ne jen "nespadnutý" program!)

4.1.3 Ošetření výjimky a předání výš - throw

```
public static int XctiInt() throws Exception {
try {
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
return i;
} catch (Exception e) {
System.out.println("Chyba v udaji");
throw e; // predani vyse
}
}
```

Volající metoda musí opět použít try-catch, nebo výjimku throws výše.

4.1.4 Kompletní Ošetření výjimky - try-catch

Kompletní ošetření výjimky se provádí konstrukcí try - catch:

try { ... } catch (Exception e) { ... }

Klauzulí catch muže být i více pro ruzné typy výjimek. Pokud výjimka vyhovuje jedné větvi catch, tato se provede a ostatní už ne. Existuje také blok finally, ten se provede vždy, ať už výjimka nastane nebo ne.

4.2 Mechanismus šíření výjimek

Jestliže vznikne výjimka, potom JVM hledá odpovídající klauzuli, která je schopná výjimku ošetrit (tj. převzít rízení):

1. pokud výjimka vznikla v bloku příkazu try, hledá se odpovídající klauzule catch v tomto příkazu, další príkazy bloku try se neprovedou a řízení se předá konstrukci

ošetrující výjimku daného typu do místa ošetrení výjimky (tzv. handler = catch blok)
2. pokud výjimka vznikne mimo příkaz try, předá se řízení do místa volání metody a pokračuje se podle předchozího bodu
3. pokud taková konstrukce v těle funkce (metody, konstruktoru) není, skončí funkce nestandardně a výjimka se šírí na dynamicky nadřazenou úroven
4. není-li výjimka ošetrena ani ve funkci main, vypíše se (na výstup) a program skončí

Pro rozlišení různých typů výjimek je v jazyku Java zavedena řada knihovních tříd, výjimky jsou instancemi těchto tříd.

4.3 Typy výjimek

Začněme obrázkem:

4.3.1 Kontrolované

Kontrolované výjimky musí být na rozdíl od nekontrolovaných explicitne deklarovány v hlavicce metody, ze které se mohou šírit, jedná se o výjimky trídy Exception, je nutné je povinne obsloužit. Oznacují se též jako výjimky synchronní:

```
void m() throws Exception {
if (...) throw new Exception();
}
```

Potomek třídy Exception muže ošetrovat i "naše" výjimky, jednoznačne synchronní, vznikají na námi specifikovaném míste. Vyžadují povinné ošetrení, jinak se ohlásí!. Třída Exception je nadtřída výjimek, které prevážne vznikají vlastní chybou aplikace a má smysl je ošetrovat (typicky ošetrení chyb vstupu/výstupu (IOException)).

4.3.2 Nekontrolované

Nekontrolované výjimky jsou takové, které se mohou šírit z vetšiny metod a proto by jejich deklarování obtežovalo, tzv. asynchronní výjimky.

- bežný uživatel není schopen výjimku ošetrit – ze třídy Error. Třída Error je nadtřída všech výjimek, které převážne vznikají v důsledku softwarových či hardwarových chyb výpocetního systému a které vetšinou nelze v aplikaci smysluplně ošetrit.
  - MemoryOverflowError - přetecení paměti
  - ClassFormatError - chybný formát byte-kódu
- chyby, které ošetrujeme podle potřeby, prekladac nekontroluje, zda tyto výjimky jsou ošetreny - podtřídy třídy RuntimeException. Nemusíme na ně reagovat, ale mužeme je predat "výše", překladač nás k reakci nenutí.
  - ArithmeticException - delení 0
  - IndexOutOfBoundsException - indexace mimo meze

- NumberFormatException - nedovolený prevod znaku na číslo, prectení nenumerické hodnoty
- NegativeArraySizeException - vytváření pole se zápornou délkou
- NullPointerException - dereference odkazu null

4.4 Vlastní výjimky

Ve vlastních třídách muže nastat stav, který chceme ošetrit standardně výjimecným stavem. Vlastní výjimka je potomkem třídy Exception. Jedná se o tzv.synchronní výjimku, vzniká na přesne definovaném míste. Vetšinou se jedná o výjimku, na kterou chceme reakci uživatele. Reakci na vlastní výjimky systém vyžaduje.
throw new Exception(); // generujeme vyjímku

4.4.1 Vytvoření vlastní výjimky

```
class MojeVyjimka extends Exception {
int n;
int d;
MojeVyjimka(int i, int j) {
n = i;
d = j;
}
public String toString() {
return "Hodnota " + n + "/" + d +" není integer.";
}
}
//priklad pouziti
throw new MojeVyjimka(numer[i], denom[i]);
```