

Seminář Java

V

Radek Kočí

Fakulta informačních technologií VUT

Březen 2011

- Pole
- Dokumentační komentáře
- Výjimky
- Vnořené třídy, Anonymní třídy, Výčtový typ (enums), Proměnný počet parametrů (varargs)

Pole

- Pole v Javě je speciálním objektem.
- Můžeme mít pole jak primitivních, tak objektových hodnot
 - pole primitivních hodnot tyto hodnoty obsahuje
 - pole objektů obsahuje odkazy na objekty
- Kromě pole v Javě existují i jiné objekty na ukládání více hodnot – *kontejnery* (bude později ...)

Před použitím je nutné pole

- deklarovat
- vytvořit
- inicializovat (naplnit)

Syntaxe deklarace

- **typ [] identifikator**
- na rozdíl od C/C++ nikdy neuvádíme při deklaraci počet prvků pole – ten je podstatný až při vytvoření objektu pole

Vytvoření pole

- jako u jiného objektu – voláním konstruktoru:
 - `nazevPole = new typ[velikost];`
 - `int[] pole = new int[10];`
- nebo inicializací při deklaraci:
 - `int[] nazevPole = { 1, 2, 3 };`

Syntaxe přístupu k prvkům

- `pole[i]`
- `pole[i] = 20;`
- `int j = pole[2];`

```
Ucet [] ucty;           // deklarace pole
ucty = new Ucet[5];     // vytvoření pole

// vytvoření objektu
// a inicializace 1. prvku pole
ucty[0] = new Ucet("Franta");

ucty[0].vypisInfo();   // přístup k prvku pole
```

- V poli `ucty` je naplněn 1. prvek odkazem na objekt
- Ostatní prvky zůstaly naplněny prázdnými odkazy `null`.

Co když vynecháme vytvoření pole?

```
Ucet [] ucty;  
ucty[0] = new Ucet("Franta");  
    // chyba, pole neexistuje
```

Co když vynecháme inicializaci pole?

```
Ucet [] ucty;  
ucty = new Ucet[5];  
ucty[0].vypisInfo();  
    // chyba, prvek neexistuje
```

Přiřazení proměnné objektového typu (a tedy i polí) vede pouze k duplikaci odkazu, nikoli celého odkazovaného objektu.

```
Ucet [] ucty = new Ucet [5];  
Ucet [] ucty2;  
ucty2 = ucty;
```

Proměnná **ucty2** obsahuje odkaz na stejné pole jako **ucty**.


```
Ucet [] ucty2 = new Ucet[5];  
System.arraycopy(ucty, 0, ucty2, 0,  
                 ucty.length);
```

- Proměnná **ucty2** obsahuje kopii původního pole.
- Také **arraycopy** však do cílového pole zduplikuje jen odkazy na objekty, nevytvoří kopie objektů!

Výpis argumentů programu

```
public class Pole {  
    public static void main(String[] args) {  
        for(int i=0; i<args.length; i++)  
            System.out.println(args[i]);  
    }  
}
```

Základní typy komentářů (podobně jako např. v C/C++)

- *řádkové* od značky `//` do konce řádku
- *blokové* (na libovolném počtu řádků) začínají `/*` pak je text komentáře, končí `*/`
- *dokumentační* (na libovolném počtu řádků) od značky `/**` po značku `*/` Každý další řádek může začínat mezerami či *, hvězdička se v komentáři neprojeví.

Dokumentace

- je generována nástrojem **javadoc**
 - z dokumentačních komentářů
 - a ze samotného zdrojového textu
- *je tedy možné dokumentovat (základním způsobem) i program bez vložených komentářů!*
- má standardně podobu HTML stránek (s rámy i bez)
- chování **javadoc** můžeme změnit volbami (options) při spuštění

Dokumentační komentáře uvádíme:

- před hlavičkou třídy (komentuje třídu jako celek)
- před hlavičkou metody nebo proměnné (komentuje příslušnou metodu nebo proměnnou)

Značky pro dokumentační komentáře

Nástroj **javadoc** můžeme podrobněji instruovat pomocí značek vkládaných do dokumentačních komentářů, např.:

@author	specifikuje autora API/programu
@version	označuje verzi API, např. "1.4.2"
@deprecated	informuje, že prvek je zavrhováný
@exception	popisuje informace o výjimce, kterou metoda propouští ("vyhazuje")
@param	popisuje jeden parametr metody
@since	uvedeme, od kdy (od které verze pg.) je věc podporována/přítomna
@see	uvedeme odkaz, kam je také doporučeno nahlédnout (související věci)

Ukázka použití dokumentačních komentářů

```
package ija1.ucty;

/**
 * Trida ucet
 * @author R. Koci
 */
public class Ucet {
    /** Majitel uctu */
    protected String majitel;
    protected double zustatek;

    public void pridej(double castka) {
        zustatek += castka;
    }
    ...
}
```

Co a k čemu jsou výjimky

- výjimka je mechanismus umožňující psát robustní, spolehlivé programy
- robustní ve smyslu odolné proti chybám "okolí" – uživatele, systému, ...
- výjimkami *v žádném případě* neošetřujeme chyby programu samotného! (hrubé zneužití)
- režie spojená s vyvoláním výjimky je vysoká!

Reprezentace výjimky

- výjimka (exception) je objekt třídy `java.lang.Exception`
- Objekty (výjimky) jsou vytvářeny (vyvolávány) buďto
 - automaticky běhovým systémem Javy, nastane-li nějaká běhová chyba, např. dělení nulou, nebo
 - jsou vytvořeny samotným programem, zdetekuje-li nějaký chybový stav, na nějž je třeba reagovat – např. do metody je předán špatný argument

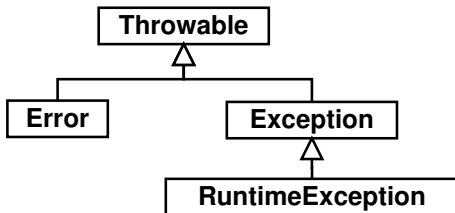
Zpracování výjimky

- Vzniklý objekt výjimky je předán buďto:
 - v rámci metody, kde výjimka vznikla, do bloku **catch** ⇒ výjimka je v bloku **catch** tzv. zachycena
 - výjimka "propadne" do nadřazené (volající) metody, kde je buďto v bloku **catch** zachycena nebo opět propadne atd.
- Výjimka tedy "putuje programem" tak dlouho, než je zachycena
- ⇒ pokud není, běh JVM skončí s hlášením o výjimce

```
try {  
    //zde může vzniknout výjimka  
}  
catch (TypVýjimky proměnnáVýjimky) {  
    // zde je výjimka ošetřena  
    // je možné přistupovat k proměnnéVýjimky  
}
```

-
- Bloku **try** se říká hlídaný blok, protože výjimky (příslušného hlídaného typu) zde vzniklé jsou zachyceny.
 - V bloku **catch** jsou zachycené výjimky ošetřeny

package `java.lang`



- **Throwable** – pouze objekty této třídy (a podtříd) mohou být generovány jako výjimky
- **Error** – vážné chyby JVM (*Out Of Memory, Stack Overflow, ...*)
- **Exception** – hlídané výjimky (checked exceptions)
- **RuntimeException** – běhové (runtime, nehlídané – unchecked) výjimky, takové výjimky nemusejí být zachytávány

Hlídané výjimky

- `java.lang.Exception`
- indikuje podmínky (stavy), které může aplikace chtít ošetřovat
- musí se explicitně uvádět
- hlídaná výjimka musí být zpracována!
- např. `java.io.FileNotFoundException`

Propuštění hlídané výjimky

- někdy není nutné či vhodné ošetřovat výjimku na místě
- metoda může deklarovat, že *propouští* výjimku (**throws**)
- klauzule **throws** říká, že během zpracování metody může být vygenerovaná uvedená výjimka, která není ošetřena
- volající metoda musí výjimku zpracovat (zachytit nebo propustit)

```
public FileReader(String fileName)
    throws FileNotFoundException;

public void close() throws IOException;
```

```
import java.io.*;
public class OtevreniSouboru {
    public static void main(String[] args) {
        String jmeno = args[0];
        FileReader r;
        System.err.println("Otviram soubor "+jmeno);
        r = new FileReader(jmeno);
        System.err.println("Soubor otevren");
        r.close();
    }
}
```

unreported exception

`java.io.FileNotFoundException`; must be caught
or declared to be thrown

```
    r = new FileReader(jmeno);
```

unreported exception `java.io.IOException`; must
be caught or declared to be thrown

```
    r.close();
```

- výjimku jsou hierarchicky uspořádané podle dědičnosti jejich tříd
- nadřazená výjimka pokrývá všechny odvozené výjimky
- př.: **FileNotFoundException** je speciálním případem **IOException**
- zpracováním **IOException** zpracujeme i **FileNotFoundException**

Zachycení výjimky

```
public static void main(String[] args) {  
    String jmeno = args[0];  
    FileReader r;  
    System.err.println("Otviram soubor "+jmeno);  
  
    try {  
        r = new FileReader(jmeno);  
        System.err.println("Soubor otevren");  
        r.close();  
    } catch (IOException ex) {  
        System.err.println("Chyba pri manipulaci  
                               se souborem.");  
    }  
}
```

Je možné zpracovat každou výjimku zvlášť.
Pozor na řazení podle hierarchie!

Zachycení výjimky

```
public static void main(String[] args) {
    String jmeno = args[0];
    FileReader r;
    System.err.println("Otviram soubor "+jmeno);
    try {
        r = new FileReader(jmeno);
        System.err.println("Soubor otevren");
        r.close();
    } catch (FileNotFoundException ex) {
        ...
    } catch (IOException ex) {
        ...
    }
}
```

Zachycení výjimky – chyba

```
public static void main(String[] args) {
    String jmeno = args[0];
    FileReader r;
    System.err.println("Otviram soubor "+jmeno);
    try {
        r = new FileReader(jmeno);
        System.err.println("Soubor otevren");
        r.close();
    } catch (IOException ex) {
        ...
    }
    // Nasledujici blok se nikdy neprovede !!!
    catch (FileNotFoundException ex) {
        ...
    }
}
```

Propuštění výjimky

```
public FileReader(String fileName)
    throws FileNotFoundException;
public void close() throws IOException;
```

```
public class OtevreniSouboru {
    static void otevri(String jmeno) {
        System.err.println("Otviram soubor "+jmeno);
        FileReader r = new FileReader(jmeno);
        r.close();
    }
    public static void main(String[] args) {
        otevri(args[0]);
        System.err.println("Soubor otevren");
    }
}
```

Propuštění výjimky

```
static void otevri(String jmeno)
```

```
    throws IOException
```

```
{
    System.err.println("Otviram soubor "+jmeno);
    FileReader r = new FileReader(jmeno);
    r.close();
}
```

```
public static void main(String[] args) {
    try {
        otevri(args[0]);
        System.err.println("Soubor otevren");
    } catch (IOException ioe) {
        System.err.println("Nelze otevrit soubor");
    }
}
```

Nehlídané výjimky

- `java.lang.RuntimeException`
- výjimky, které mohou být generovány během standardních operací JVM
- nemusí se explicitně uvádět
- nemusí se zachytávat
- např.
`java.lang.ArrayIndexOutOfBoundsException`,
`java.lang.NullPointerException`

Generování výjimky

- výjimku lze generovat (klíčové slovo `throw`)

```
public abstract class Reader ... {
    public void mark(int limit)
        throws IOException
    {
        throw new IOException("mark() not supported");
    }

    public InputStreamReader(.., String charsetName)
    {
        if (charsetName == null)
            throw new NullPointerException("charsetName")
        }
    }
}
```


Klauzule (blok) `finally`:

- může následovat ihned po bloku `try` nebo až po blocích `catch`
- slouží k "úklidu v každém případě", tj.
 - když je výjimka zachycena blokem `catch`
 - i když je vygenerovaná jiná než ošetřovaná výjimka
 - i když je výjimka propuštěna do volající metody
- Používá se typicky pro uvolnění systémových zdrojů – uzavření souborů ...

Vlastní výjimky

- typy (tj. třídy) výjimek si můžeme definovat sami
- bývá zvykem končit názvy tříd (výjimek) na **Exception**
- *je lepší využívat standardní výjimky!*

```
class MyException extends Exception {
    protected int pocetParametru;

    public MyException(int pocet) {
        pocetParametru = pocet;
    }

    public int getPocetParametru() {
        return pocetParametru;
    }
}
```

Ukázka vlastní výjimky a klauzule finally

```
public static void main(String[] args) {
    int pocetParametru = args.length;
    try {
        if (pocetParametru < 2)
            throw new MyException(pocetParametru);
        System.out.println("Spravny pocet: "
            + pocetParametru);
    } catch (MyException me) {
        System.out.println("Malo parametru: "
            + me.getPocetParametru());
    } finally {
        System.out.println("Konec");
    }
}
```

Jak můžeme reagovat?

- Napravit příčiny vzniku chybového stavu – např. znovu nechat načíst vstup
- Poskytnout za chybný vstup náhradu – např. implicitní hodnotu
- Operaci neprovést ("vzdát") a sdělit chybu výše tím, že výjimku "propustíme" z metody

Výjimková pravidla:

- Vždy nějak reagujeme! Neignorujeme, nepotlačujeme, tj.
- blok **catch** nenecháme prázdný, přinejmenším vypíšeme `e.printStackTrace()`
- Nelze-li reagovat na místě, propustíme výjimku výše (a popíšeme to v dokumentaci...)

Ukázka nesprávného použití výjimky

```
public static void main(String[] args) {  
    try {  
        otevri(args[0]);  
    }  
    catch (ArrayIndexOutOfBoundsException e) {  
        System.err.println("Neni zadan argument.");  
    }  
}
```

```
public static void main(String[] args) {  
    if (args.length == 0) {  
        System.err.println("Neni zadan argum.");  
        return;  
    }  
    otevri(args[0]);  
}
```

Ukázka nesprávného použití výjimky

```
try {
    int i = 0;
    while(true) {
        a[i++] = i;
    }
} catch (ArrayIndexOutOfBoundsException e) {}
```

```
int length = a.length;
for(int i = 0; i < length; ) {
    a[i++] = i;
}
```

Třídy nejvyšší úrovně (top-level classes)

- "normální třídy" – jsou přímými členy nějakého balíku

```
package balik;
```

```
public class TopLevel1 {  
    ...  
}
```

Lokální třídy

- vnořené v jiné třídě (na úrovni lokálních proměnných)
- uváděné uvnitř bloku (platné pouze v uvedeném bloku)
- nesmí být *public*, *private* a *protected*

Vnořené třídy (inner classes)

```
public class TopLevel1 {
    private String text = "interni promenna";
    public void test() {
        {
            class A {
                public A() {
                    System.out.println(text);
                }
            }
            A a = new A();
        }
        // tady uz neni trida A dostupna
    }
}
```

Vnořené třídy (inner classes)

Členské třídy (member classes)

- vnořené v jiné třídě (na úrovni vlastností objektu)
- mohou být *public*, *private* a *protected*

```
public class TopLevel1 {  
    private String text = "interni promenna";  
    class Inner {  
        public Inner() {  
            System.out.println("Inner: " + text);  
        }  
    }  
    public m() {  
        Inner i = new Inner();  
        ...  
    }  
}
```

Vnořené top-level třídy

- členské třídy s modifikátorem **static**
- vnořená rozhraní
- mohou být *public*, *private* a *protected*

Vnořené třídy (inner classes)

```
public class TopLevel1 {
    private String text = "interni promenna";
    static public class TopLevel2 {
        public TopLevel2() {
            TopLevel1 t = new TopLevel1();
            System.out.println(t.text);
        }
    }
    interface Cool {
        ...
    }
}
```

Vnořené top-level třídy

- používá se k seskupení souvisejících tříd bez nutnosti vytvářet nový balík
- přístup k vnořeným top-level třídám (rozhraním)
`new TopLevel1.TopLevel2 () ;`

Anonymní třídy (anonymous classes)

- zvláštní případ vnořené třídy

```
new Typ ( parametry ) {  
    tělo anonymní třídy  
}
```

- Typ představuje
 - jméno konstrukturu rodičovské(!) třídy, od které je anonymní třída odvozena (následují jeho parametry)
 - jméno rozhraní – anonymní třída jako jediná může přímo instanciovat rozhraní (zde se parametry neuvádějí)

```
class NejakaTrida {  
    Runnable r = new Runnable() {  
        public void run() {  
            // ...  
        }  
    }  
}
```

Reprezentace výčtového typu (Java < 5.0)

- int Enum pattern

```
public static final int SEASON_WINTER = 0;  
public static final int SEASON_SPRING = 1;  
public static final int SEASON_SUMMER = 2;  
public static final int SEASON_FALL = 3;
```

- není typově bezpečný
- hodnoty nejsou informativní (vždy int; "jiné" názvy, stejná hodnota)

Výčtový typ (Java \geq 5.0)

- typově bezpečný
- `enum Season {WINTER, SPRING, SUMMER, FALL}`
- plnohodnotná třída!
- rozšiřuje třídu `java.lang.Enum`

```
class EnumsDemo {  
    public enum Season { WINTER, SPRING, SUMMER,  
                        FALL };  
    private Season season;  
  
    public static void main(String[] args) {  
        new EnumsDemo();  
    }  
    public EnumsDemo() {  
        season = Season.WINTER;  
        System.out.println(season);  
        for (Season s : Season.values())  
            System.out.println("Value of Season: "  
                                + s);  
    }  
}
```

```
public enum Planet {  
    MERCURY (3.303e+23, 2.4397e6),  
    VENUS   (4.869e+24, 6.0518e6),  
    EARTH   (5.976e+24, 6.37814e6);  
  
    private final double mass;    // in kilograms  
    private final double radius; // in meters  
  
    Planet(double mass, double radius) {  
        this.mass = mass;  
        this.radius = radius;  
    }  
}
```

```
public static final double G = 6.67300E-11;

double surfaceGravity() {
    return G * mass / (radius * radius);
}

double surfaceWeight(double otherMass) {
    return otherMass * surfaceGravity();
}
}
```

```
public static void main(String[] args) {
    double earthWeight = Double.parseDouble(args[0]);
    double mass =
        earthWeight/Planet.EARTH.surfaceGravity();

    for (Planet p : Planet.values())
        System.out.printf("Your weight on %s is %f%n",
                           p, p.surfaceWeight(mass));
}
```

<http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html>

Varargs

- metoda s proměnným počtem parametrů
- pole objektů třídy **Object**

```
Object[] arguments = {  
    new Integer(7),  
    new Date(),  
    "a disturbance in the Force"  
};
```

```
String result = MessageFormat.format(  
    "At {1,time} on {1,date}, there was {2} " +  
    "on planet {0,number,integer}.", arguments);
```

Varargs (Java \geq 5.0)

- vlastnost skrývající nutnost obalení polem
- poslední argument: **Object...** **name**

```
class Args {  
  
    public static void main(String[] args) {  
        Args a = new Args();  
        a.print("Hi ", "hou ", "ha");  
    }  
  
    public void print(String s, Object... args) {  
        String str = "";  
        System.out.println(args.length);  
        for (Object o : args)  
            str += (String) o;  
        System.out.println(s + str);  
    }  
}
```