JAVA
VÝPISKY

00_Obsah

00_Obsah
01_Základy
02_Datové typy a proměnné
03_Výrazy a příkazy
04_Řídicí struktury
05_Pole
06_Metody
07_Třídy
08_Dědičnost
09_Výjimky
10_Vlákna
11_Soubory a proudy
12_GUI

Použitá literatura:

[1] KEOGH, James. Java bez předchozích znalostí : Průvodce pro samouky. [s.l.] : Computer Press, 2005. 280 s. ISBN 80-251-0839-2.

01_Základy

Stručná historie vysokoúrovňových jazyků:

50. léta – Fortran, Cobol
60. léta – BCPL, B (v něm byl napsán 1. Unix)
1972 – C
1980 – C++ (1. objektový jazyk)
1991 – Java [džava] (původně Oak)
1995 – oficiální uvedení Javy společností Sun Microsystems (java.sun.com)

Standardy:

ANSI, ISO

Edice:

Java 2 Standard Edition (J2SE): tvorba běžných aplikací
Java 2 Enterprise Edition (J2EE): tvorba podnikových aplikací
Java 2 Micro Edition (J2ME): tvorba mobilních aplikací

Nastavení Path a Classpath:

tento počítač – vlastnosti – upřesnit – proměnné prostředí – systémové proměnné
proměnná: Path, Hodnota: cesta k JDK;
proměnná: Classpath, Hodnota: cesta k JDK; vlastní cesty;

Kompilace a spuštění programu v příkazovém řádku:

```
javac program.java
```

program zkompiluje zdrojový kód ve formátu .java do bajtového kódu a vznikne objektový soubor .class

```
java program
```

virtuální stroj Javy přeloží bajtový kód do strojového jazyka, proto je Java multiplatformní

První program:

```
class Program
{
     public static void main (String arg[])
     {
          System.out.print(„Nazdar chlape!!!“);
     }
}
```

kód se musí uložit do souboru program.java, název souboru se musí shodovat s názvem třídy

Komentáře:

komentář na 1 řádku: `// komentář`
komentář s určitou délkou: `/* komentář`
`komentář */`

02_Datové typy a proměnné

Kódování znaků:

ASCII (128 znaků), Unicode (1 114 112 znaků)

Vyhledání hodnoty Unicode:

```
class Program
{
     public static void main (String arg[])
     {
          char x = 'J';
          double a = x;
          System.out.println(„Kod Unicode je “ + a);
     }
}
```

Literály:

celočíselné – decimální: 0,1,2,3,4,5,6,7,8,9
celočíselné – oktalové: 00,01,02,03,04,05,06,07
celočíselné – hexadecimální: 0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9,0xA,0xB,0xC,0xD,0xE,0xF

desetinné – s tečkou: např. 3.1415
desetinné – exponenciální – jednoduše přesná (7 míst) – velmi velká: např. 5e+20 nebo 5E20
desetinné – exponenciální – jednoduše přesná (7 míst) – velmi malá: např. 5e-20 nebo 5E-20
desetinné – exponenciální – dvojnásobně přesná (15 míst) – velmi velká: např. 5e+20f nebo 5E20F
desetinné – exponenciální – dvojnásobně přesná (15 míst) – velmi malá: např. 5e-20f nebo 5E-20F

booleovské: true, false

znakové: např. ‘a’, ‘A’, ‘1’

escape znaky: \n (nový řádek), \t (tabulátor), \r (návrat vozíku), \f (posuv formuláře), \b (zpětná mezera), \\ (zpětné lomítko), \“ (uvozovky)

řetězcové: např. „Petr Novák“

Názvy některých znaků:

#	haš, šarp, mřížka, křížek
&	ampersand
%	modulo
~	tilda

Datové typy:

typ	velikost	rozsah hodnot
byte	8 bitů	-128 až 127
short	16 bitů	-32 768 až 32 767
int	32 bitů	-2 147 483 648 až 2 147 483 647
long	64 bitů	-9 223 372 036 854 775 808 až 9 223 372 036 854 775 807
char	16 bitů (Unicode)	65 000 (Unicode)
float	32 bitů	3.4e-038 až 3.4e+038
double	64 bitů	1.7e-308 až 1.7e+308
boolean	1 bit	0 nebo 1

Konverze datových typů (přetypování):

```
class Program
{
     public static void main (String arg[]) throws java.io.IOException
     {
          char volba = (char) System.in.read();
     }
}
```

Převod číselné hodnoty řetězce String na typ int:

```
class Program
{
     public static void main (String arg[])
     {
          try
          {
               String s = "123";
               int i = Integer.parseInt(s);
               System.out.println("String: " + s + "\nint: " + i);
          }
          catch(Exception e)
          {
               System.out.println("Doslo k vyjimce: " + e);
          }
     }
}
```

Alokování paměti (deklarace) a inicializace proměnné:

```
int znamka;
int znamka1, znamka2, znamka3;

int pocet = 10;
int pocet = 1 + 3 + 4 + 2;
int pocet1 = 10, pocet2 = -20, pocet3 = 0;
int celkem_pocet = pocet1 + pocet2 + pocet3;
```

popis: datový typ, identifikátor (název), středník

Java rozlišuje malá a velká písmena

identifikátor nesmí začínat číslicí, nesmí obsahovat mezery, nesmí být klíčovým slovem jazyka Java:

```
abstract, assert, boolean, break, byte, case, catch, char, class,
const, continue, default, do, double, else, extends, final, finally,
float, for, goto, if, implements, import, instanceof, int, interface,
long, native, new, package, private, protected, public, return,
short, static, strictfp, super, switch, synchronized, this, throw,
throws, transient, try, void, volatile, while
```

rozsah platnosti proměnné je v bloku kódu, kde je proměnná deklarována či ve vnořeném bloku

03_Výrazy a příkazy

Výrazy:

např. a + b – 10
výrazy jsou tvořeny operandy (čísla a proměnné), operátory a klíčovými slovy

Priorita operátorů:

priorita	typ	operátor
1	postfixové	[].(*parametry*) *výraz*++ *výraz*--
2	unární	++*výraz* –*výraz* +*výraz* –*výraz* ~ !
3		vytvoření nebo přetypování
4		new
5	multiplikativní	* / %
6	součtové	+ -
7	posuvu	<< >> >>>
8	relační	< > <= >= instanceof
9	rovnosti	== !=
10	bitové AND	&
11	bitové exkluzivní OR	^
12	bitové inkluzivní OR	|
13	logické AND	&&
14	logické OR	||
15	ternární	?:
16	přiřazovací	= += -= *= /= %= &= ^= |= <<= >>= >>>=

operátory s vyšší prioritou se vyhodnocují dříve, pokud je priorita stejná, vyhodnocují se zleva doprava, prioritu operátorů u složených výrazů lze upravovat pomocí závorek

Operátory:

unární operátory: vyžadují jeden operand (např. inkrementace: a++)
binární operátory: vyžadují dva operandy (např. součet: a + b)

aritmetické operace	**operátor**	**popis**
sčítání	+	např. a + b
odčítání a unární minus	-	např. a – b
násobení	*	např. a * b
dělení	/	např. a / b
modulo	%	např. a % b, vrací zbytek při dělení
přiřazení	=	např. a = b
sčítání a přiřazení	+=	např. a += b
odčítání a přiřazení	-=	např. a -= b
násobení a přiřazení	*=	např. a *= b
dělení a přiřazení	/=	např. a /= b
modulo a přiřazení	%=	např. a %= b
inkrementace	++	prefixová: např. ++a, postfixová: např. a++
dekrementace	--	prefixová: např. --a, postfixová: např. a--

relační operace	**operátor**	**popis**
rovno	==	např. a == b
nerovno	!=	např. a != b
větší než	>	např. a > b
menší než	<	např. a < b
větší nebo rovno	>=	např. a >= b
menší nebo rovno	<=	např. a <= b

logické operace	**operátor**	**popis**
AND	&&	nevyhodnotí výraz celý není-li třeba, např. a==b && b==c
OR	||	nevyhodnotí výraz celý není-li třeba, např. a==b || b==c
AND	&	vyhodnotí výraz celý, např. a==b & b==c
OR	|	vyhodnotí výraz celý, např. a==b | b==c
ternární if-then-else	?:	např. String hodnoceni = body>80 ? „prospel“ : „neprospel“

bitové operace	**operátor**	**popis**
bitové AND	&	např. a & b
bitové inkluzivní OR	|	např. a | b
bitové exkluzivní OR	^	např. a ^ b
posun vlevo	<<	např. a << b
posun vpravo	>>	např. a >> b
posun vpravo bez znaménka	>>>	např. a >>> b
bitový komplement (doplněk)	~	např. a ^ b ~ a

Příkazy:

příkazy jsou tvořeny výrazy, musí být zakončeny středníkem

04_Řídicí struktury

Výběrové řídicí příkazy:

příkaz if else:

```
if (a == true)
{
     c++;
}
else
{
     c--;
}
```

když a je true, proveď c++, jinak proveď c--, lze napsat jen if(a)

příkaz else if:

```
if (a == b)
{
     c++;
}
else if (a > b)
{
     c--;
}
```

když a je rovno b, proveď c++, jinak když a je větší než b proveď c--

příkaz switch:

```
switch (a)
{
     case 1:   b = 1;
               break;
     case 2:   b = 2;
               break;
     case 3:   b = 3;
     default:  b += 10;
}
```

když a je rovno 1, proveď b=1 a ukonči příkaz switch
když a je rovno 2, proveď b=2 a ukonči příkaz switch
když a je rovno 3, proveď b=3 a pokračuj v příkazu switch
v ostatních případech proveď b+=10

Iterační řídicí příkazy:

cyklus for:

```
for (int a = 0; a < 10; a++)
{
     b++;
}
```

popis: int a = 0 je inicializační výraz, a < 10 je řídicí podmíněný výraz

další alternativy inicializačního výrazu:

```
int a;
for (a = 0; a < 10; a++)
{
     b++;
}

int a = 0;
for (; a < 10; a++)
{
     b++;
}
```

další alternativy podmíněného výrazu:

```
boolean x = true;
for (int a = 0; x; a++)
{
     b++;
     if (a == 10) x = false;
}

for (int a = 0; ; a++)
{
     b++;
     if (a == 10) break;
}
```

cyklus while:

```
while (a < 10)
{
      a++;
      b++;
}
```

pokud a je menší než 10, prováděj a++, b++

další alternativa:

```
while (true)
{
      if (a == 10) break;
      a++;
      b++;
}
```

cyklus do while:

```
do
{
      a++;
      b++;
}
while (a < 10);
```

Skokové příkazy:

příkaz break:

přikazuje opustit blok kódu daného cyklu či příkazu switch

příkaz continue:

přenese řízení programu na začátek daného cyklu jako kdyby program došel na konec cyklu

příkaz return:

v metodě vrací řízení zpět příkazu, který metodu volal

05_Pole

Alokování paměti (deklarace) a inicializace pole:

```
int hodnoceni[] = new int[10];
int []hodnoceni = new int[10];
int hodnoceni = new int[10];

int hodnoceni;
hodnoceni = new int[10];
```

popis: int je datový typ, hodnocení je odkaz (reference) na pole celých čísel, new je operátor, který rezervuje paměť pro uložení 10 celých čísel

```
int hodnoceni[] = {95,87,93,84,79,100,95,96,84,89};
int []hodnoceni = {95,87,93,84,79,100,95,96,84,89};
```

Přístup k prvku pole a přiřazení hodnoty:

```
hodnoceni[0] = 95;
hodnoceni[0] = hodnoceni[1];
hodnoceni[0] = promenna;
```

popis: hodnocení je název pole, [0] je index pole, první index pole je 0, proměnná musí být stejného datového typu jako pole

Vícerozměrná pole a nesouměrná pole:

```
int hodnoceni[][] = new int [3][2];
int [][]hodnoceni = new int [3][2];

int hodnoceni[][] = new int[3][];
hodnoceni[0] = new int[50];
hodnoceni[1] = new int[50];
hodnoceni[2] = new int[50];
```

popis: pole má deklarován jen jeden rozměr, dále je deklarován druhý rozměr

Datový člen length:

```
int x = mojePole.length;
```

popis: vrací délku pole

Předávání pole do metody:

```
public static void main(String arg[])
{
     float hodnoceni;
     float vysledekTestu[] = {70,100};
     hodnoceni = vypocetHodnoceni(vysledekTestu);
}

static float vypocetHodnoceni(float test[])
{
     return (test[0] / test[1]) * 100;
}
```

Vrácení pole z metody:

```
public static void main(String arg[])
{
     float vysledekTestu[];
     vysledekTestu = dataTestu();
}

static float[] dataTestu()
{
     float vysledekTestu[] = {70,100};
     return vysledekTestu;
}
```

Třída Arrays (balíček java.util):

equals():

```
Arrays.equals(pole1, pole2);
```

popis: metoda vrací booleanovskou hodnotu true, jestliže se dvě pole rovnají

fill():

```
Arrays.fill(mojePole, 0);
```

popis: metoda inicializuje všechny prvky pole na hodnotu 0

```
Arrays.fill(mojePole, 10, 21, 0);
```

popis: metoda inicializuje prvky 10 až 20 pole na hodnotu 0

sort():

```
Arrays.sort(mojePole);
```

popis: metoda setřídí hodnoty pole podle abecedy

binarySearch():

```
Arrays.binarySearch(mojePole, „Petr“);
```

popis: metoda vrací index pole s hodnotou „Petr“, záporné číslo znamená, že žádný prvek pole hodnotu neobsahuje, před voláním metody by se mělo pole setřídit metodou Arrays.sort()

06_Metody

Typy metod:

nestatické metody: lze volat jen prostřednictvím instance dané členské třídy
statické metody: lze volat aniž bychom museli deklarovat instanci dané členské třídy

Definice a volání metody:

definici metody tvoří hlavička (tvořená názvem metody a seznamem argumentů) a tělo metody

seznam argumentů: je tvořen datovým typem a názvem argumentu, data vstupující do metody musí být stejného datového typu jako data specifikovaná na seznamu argumentů, musí být stejné i jejich pořadí a počet

```
public static void main(String arg[])
{
     int cislo1 = 5, cislo2 = 10;
     System.out.println(„Vysledek je: “ + vypocet(cislo1, cislo2));
}

static int vypocet(int x, int y)
{
     return x + y;
}
```

popis: příkaz return vrací návratovou hodnotu z metody

```
public static void main(String arg[])
{
     zprava();
}

static void zprava()
{
     System.out.println(„Ahoj!!!“);
}
```

popis: metoda typu void nevrací žádnou hodnotu, metoda může mít i prázdný seznam argumentů

Argumenty příkazového řádku:

argumenty z příkazového řádku se postupně ukládají do pole arg[] metody main()

```
public static void main (String arg[])
```

předání argumentů při spuštění programu: `java program Petr „Jan Pavel“ \“Karel\“`

popis: do pole arg[] metody main() třídy program se uloží 3 hodnoty: Petr, Jan Pavel, „Karel“

Polymorfismus (přetěžování metod):

metody lze přetížit pouze pokud mají rozdílné signatury, signaturu metody tvoří název a seznam argumentů, který jednoznačně odlišuje metodu od ostatních, seznam argumentů se může lišit: různým počtem argumentů, různými datovými typy

```
public static void main(String arg[])
{
     zobrazitChybu();
     zobrazitChybu(„Dosel papir!!!“);
     zobrazitChybu(„Doslo k chybe cislo: “, 3);
     zobrazitChybu(4, „Doslo k chybe cislo: “);
}

static void zobrazitChybu()
{
     System.out.println(„Doslo k chybe!!!“);
}

static void zobrazitChybu(String zprava)
{
     System.out.println(zprava);
}

static void zobrazitChybu(String zprava, int cislo)
{
     System.out.println(zprava + cislo);
}

static void zobrazitChybu(int cislo, String zprava)
{
     System.out.println(zprava + cislo);
}
```

07_Třídy

Specifikátory přístupu:

členy třídy (členská data / atributy / proměnné a členské metody) jsou zapouzdřeny uvnitř definice třídy

public: ke členu třídy lze přistupovat odkudkoliv
private: ke členu třídy mohou přistupovat jen členové stejné třídy
protected: ke členu třídy mohou přistupovat jen členové stejné či děděné třídy a třídy ze stejného balíčku

pokud není specifikátor uveden, člen třídy bude veřejný (public)

Definice třídy a konstruktor:

```
class MojeTrida
{
     private int student;
     mojeTrida()
     {
          student = 12345;
     }
}
```

popis: každá instance dané třídy má svoje vlastní instanční proměnné, ale sdílí členské metody třídy
parametrizovaný konstruktor: má seznam argumentů
uvnitř definice jedné třídy může být vnořena jiná třída

Deklarace instance třídy (objektu):

```
MojeTrida x = new MojeTrida();
```

popis: x je odkaz (reference) na instanci třídy MojeTrida, new je operátor, který rezervuje paměť pro všechny proměnné instance

```
MojeTrida x;
x = new MojeTrida();
```

popis: nejprve dojde k deklaraci odkazu a poté k přiřazení instance do odkazu

```
MojeTrida x,y,aktualni;
x = new MojeTrida();
y = new MojeTrida();
aktualni = x;
aktualni = y;
```

popis: ukládání různých instancí do stejného odkazu (odkaz nesmí být deklarován jako final), operátor new dynamicky (za běhu) alokuje paměť pro instanci třídy

Přístup ke členům třídy:

```
x.metoda();
x.promenna = 0;
```

popis: x je odkaz na instanci dané třídy

Přetěžování členských metod a konstruktoru:

```
class Program
{
     public static void main(String arg[])
     {
          Student novak = new Student();
          Student svoboda = new Student(128);
          novak.tisk();
          svoboda.tisk();
          novak.tisk(„Petra Novaka: “);
          svoboda.tisk(„Pavla Svobody: “);
     }
}

class Student
{
     private int id;

     Student()
     {
          id = 0;
     }
     Student(int cislo)
     {
          id = cislo;
     }
     public void tisk()
     {
          System.out.println(„Identifikacni cislo: „ + id);
     }
     public void tisk(String jmeno)
     {
          System.out.println(„Identifikacni cislo: „ + jmeno + id);
     }
}
```

Klíčové slovo this:

lokální proměnná: proměnná deklarovaná uvnitř členské metody

```
class Program
{
	public static void main(String arg[])
	{
		MojeTrida x = new MojeTrida();
		x.zobrazit();
	}
}

class MojeTrida
{
	int id;

	MojeTrida()
	{
		id=0;
	}
	public void zobrazit()
	{
		int id = 128;
		System.out.println(id);
		System.out.println(this.id);
	}
}
```

popis: lokální proměnná má přednost před instanční, klíčové slovo this odkazuje na instanční proměnnou id třídy MojeTrida

Metoda finalize():

Java uvolňuje paměť (odstraňuje instanci z paměti) automaticky

```
protected void finalize()
{
	//prikazy;
}
```

popis: metoda finalize() se volá automaticky bezprostředně před uvolněním paměti

Balíčky:

```
package mujBalicek;
package rodicovskyBalicek.dcerinnyBalicek;
```

popis: vytvoření balíčku, všechny třídy programu se uloží do balíčku, který se uloží do stejného adresáře jako je název balíčku

```
import mujBalicek.MojeTrida;
import mujBalicek.*;
```

popis: importování balíčku

```
mujBalicek.MojeTrida.metoda();
```

popis: volání metody bez importování balíčku

08_Dědičnost

Dědění tříd:

z objektu A lze odvodit objekt B pokud je objekt A objektem B

```
class A
{
     //prikazy;
}

class B extends class A
{
     //prikazy;
}
```

popis: třída B (dceřinná třída / podtřída) dědí třídu A (rodičovská třída / nadtřída), dceřinná třída má přístup k veřejným (public) a chráněným (protected) členům rodičovské třídy, rodičovská třída však nemá přístup ke členům dceřinné třídy

Volání konstruktorů a klíčové slovo super:

implicitní konstruktor: se automaticky volá při vytváření instance třídy, nepřijímá žádné argumenty

při deklaraci instance dceřinné třídy se volá konstruktor rodičovské a ihned poté i dceřinné třídy, volání konstruktoru rodičovské třídy proběhne jen pokud v konstruktoru dceřinné třídy explicitně nevoláme konstruktor rodičovské třídy

```
super();
super(555);
```

popis: volání kontruktoru rodičovské třídy z konstruktoru dceřinné třídy

```
super.zobrazit();
```

popis: volání metody zobrazit() rodičovské třídy uvnitř kódu metody zobrazit() dceřinné třídy

```
class Program
{
     public static void main (String arg[])
     {
          PostStudent ps = new PostStudent();
          ps.zobrazit();
     }
}

class Student
{
     private int ID;

     Student ()
     {
          ID = 12354;
     }

     Student (int cislo)
     {
          ID = cislo;
     }

     protected void zobrazit()
     {
          System.out.println("ID je: " + ID);
     }
}

class PostStudent extends Student
{
     PostStudent ()
     {
          super(555);
     }

     public void zobrazit()
     {
          super.zobrazit();
     }
}
```

Víceúrovňová dědičnost a přetížení členských metod:

```
class Program
{
     public static void main (String arg[])
     {
          PostStudent ps = new PostStudent();
          ps.zobrazit();
     }
}

class Osoba
{
     protected String jmeno;

     Osoba()
     {
          jmeno = "Petr Novak";
     }

     void zobrazit()
     {
          System.out.println("Trida Osoba: " + jmeno);
     }
}

class Student extends Osoba
{
     protected int ID;

     Student ()
     {
          ID = 12354;
     }

     void zobrazit()
     {
          System.out.println("Trida Student: " + ID);
     }
}

class PostStudent extends Student
{
     void zobrazit()
     {
          System.out.println("Trida PostStudent: ");
          System.out.println("Jmeno: " + jmeno);
          System.out.println("ID: " + ID);
     }
}
```

Dynamické odbavení metody:

```
Osoba test;
Osoba o = new Osoba();
Student s = new Student();
PostStudnet ps = new PostStudent();

test = o;
test.zobrazit();
test = s;
test.zobrazit();
test = ps;
test.zobrazit();
```

popis: vychází z předchozího příkladu, proměnná test neodkazuje na nic dokud do ní neuložíme obsah odkazu o, v tu chvíli budou oba odkazy test a o ukazovat na stejnou instanci

Klíčové slovo final:

```
class Osoba
{
     final void varovnaZprava()
     {
          System.out.println(„Error“);
     }
}
```

popis: klíčové slovo final zabrání překrytí dané členské metody třídy

```
final class Osoba
{
     void varovnaZprava()
     {
          System.out.println(„Error“);
     }
}
```

popis: klíčové slovo final zabrání dědění dané třídy

Abstraktní třídy:

```
class Program
{
     public static void main (String arg[])
     {
          Student s = new Student();
          PostStudent ps = new PostStudent();
          s.zobrazit();
          ps.zobrazit();
     }
}

abstract class Osoba
{
          abstract void zobrazit();
}

class Student extends Osoba
{
     protected int ID;

     Student ()
     {
          ID = 12354;
     }

     void zobrazit()
     {
          System.out.println("Trida Student: " + ID);
     }
}

class PostStudent extends Student
{
     void zobrazit()
     {
          System.out.println("Trida PostStudent: ");
          System.out.println("ID: " + ID);
     }
}
```

popis: abstraktní (obecná) členská metoda uvnitř abstarktní třídy představuje definici metody, kterou tvoří název metody, seznam argumentů a návratová hodnota, ale neobsahuje tělo metody, každá dceřinná třída poskytne své vlastní tělo metody, všechny abstraktní třídy musí být rodičovské, nelze deklarovat instanci abstraktní třídy, ale lze deklarovat odkaz na abstraktní třídu, abstraktní třída může obsahovat i neabstraktní členy

Třída Object:

členská metoda	**popis**
Object clone()	vytvoří nový objekt z klonovaného objektu
boolean equals(Object obj)	zjistí, zda se dva objekty rovnají
void finalize()	volá se těsně před tím, než úklidový systém odstraní objekt
Class getClass()	zjistí za chodu programu třídu objektu
int hashCode()	vrací kód hash objektu
void notify()	obnoví provádění vlákna čekajícího na volaný objekt
void notifyAll()	obnoví provádění všech vláken čekajících na volaný objekt
String toString()	vrací řetězec objektu
void wait()	čeká na další vlákno, dříve než zavolá objekt
void wait(long milisekundy)	čeká na další vlákno, dříve než zavolá objekt
void wait(int nanosekundy)	čeká na další vlákno, dříve než zavolá objekt

09_Výjimky

Základy ošetření výjimek:

```
class Program
{
     public static void main(String arg[])
     {
          try
          {
               int a = 10, b = 0, c;
               c = a/b;
          }
          catch (ArithmeticException v)
          {
               System.out.println(„Chyba: „ + v);
          }
     }
}
```

popis: program vygeneruje výjimku „dělení nulou“ a přiřadí ji do proměnné v, blok catch musí následovat bezprostředně po bloku try

Vícenásobné bloky catch:

```
class Program
{
     public static void main (String arg[])
     {
          try
          {
               int a[] = new int[3];
               a[0] = 10;
               a[1] = 0;
               a[2] = a[0]/a[3];
          }
          catch (ArithmeticException v)
          {
               System.out.println(„Chyba: „ + v);
          }
          catch (ArrayIndexOutOfBoundsException v)
          {
               System.out.println(„Chyba: „ + v);
          }
     }
}
```

popis: kdyby první blok catch výjimku zachytil, byla by ošetřena a do druhého bloku by se program už nedostal, uvnitř jednoho bloku try může být vnořen další blok try

Blok finally:

```
class Program
{
     public static void main(String arg[])
     {
          try
          {
               int a[] = new int[3];
               a[0] = 10;
               a[1] = 0;
               a[2] = a[0]/a[3];
          }
          catch (ArithmeticException v)
          {
               System.out.println(„Chyba: „ + v);
          }
          catch (ArrayIndexOutOfBoundsException v)
          {
               System.out.println(„Chyba: „ + v);
          }
          finally
          {
               System.out.println(„Koncovy blok finally“);
          }
     }
}
```

popis: příkazy v bloku finally se provedou vždy bez ohledu na to zda byla či nebyla vygenerována výjimka

Standardní obslužný kód:

```
class Program
{
     public static void main (String arg[])
     {
          int a = 10, b = 0, c;
          c = a/b;
     }
}
```

popis: program zachytí výjimku a vypíše hlášení popisující výjimku a výpis ze zásobníku, ukazující na místo, kde se stala chyba

Vyvolání výjimky:

```
class Program
{
     public static void main(String arg[])
     {
          int a=10, b=0, c;
          try
          {
               if (b == 0)
               {
                    throw new ArithmeticException („Deleni nulou!“);
               }
               else
               {
                    c = a/b;
               }
          }
          catch (ArithmeticException v)
          {
               System.out.println(„Chyba: „ + v);
               a=0;
          }
          System.out.println(„a = „ + a);
     }
}
```

popis: příkaz throw explicitně vyvolá výjimku, používá se také k testování bloků catch

Metody, které neošetřují výjimky:

```
class Program
{
     static void mojeMetoda() throws ArithmeticException,
     ArrayIndexOutOfBoundsException
     {
          int a[] = new int[3];
          a[0] = 10;
          a[1] = 0;
          a[2] = a[0]/a[1];
          System.out.println(„Uvnitr metody mojeMetoda.“);
     }
     public static void main(String arg[])
     {
          try
          {
               mojeMetoda();
          }
          catch (ArithmeticException v)
          {
               System.out.println(„Chyba: „ + v);
          }
     }
}
```

popis: příkaz throws specifikuje seznam výjimek, metoda mojeMetoda() může vygenerovat dvě výjimky, ale nezachycuje žádnou, metoda main() zachycuje jednu výjimku, ale o druhou se stará standardní obslužný kód

Kontrolované a nekontrolované výjimky:

nekontrolované výjimky: kompilátor neověřuje, jestli bude blok catch výjimky zachycovat

třída nekontrolované výjimky	**popis**
ArithmeticException	aritmetická chyba, např. dělení nulou
ArrayIndexOutOfBoundsException	index pole mimo dosah
ArrayStoreException	přiřazení prvku nekompatibilního typu do pole
ClassCastException	neplatná konverze
IllegalArgumentException	při volání metody byl použit neplatný argument
IllegalMonitorStateException	neplatná monitorující operace, např. čekání na odemknuté vlákno
IllegalStateException	prostředí nebo aplikace se nacházejí v nesprávném stavu
IllegalThreadStateException	operace není kompatibilní s momentálním stavem vlákna
IndexOutOfBoundsException	některý typ indexu je mimo rozsah
NegativeArraySizeException	pokus o vytvoření pole se zápornou velikostí
NullPointerException	neplatné použití odkazu null
NumberFormatException	neplatná konverze řetězce na číselný formát
SecurityException	pokus o narušení bezpečnosti
StringIndexOutOFBounds	pokus o přístup mimo rozsah řetězce
UnsupportedOperationException	program provedl nepodporovanou operaci

kontrolované výjimky: se musí explicitně ošetřit buď zachycením (catch), nebo se musí deklarovat v hlavičce metody jako generované (throws), kompilátor ověřuje, jestli bude blok catch výjimky zachycovat

třída kontrolované výjimky	**popis**
ClassNotFoundException	třída nebyla nalezena
CloneNotSupportedException	pokus klonovat objekt bez implementovaného rozhraní Cloneable
IllegalAccessException	přístup ke třídě zamítnut
InstantiationException	pokus vytvořit objekt abstraktní třídy nebo rozhraní
InterruptedException	jedno vlákno bylo přerušeno jiným
NoSuchFieldException	požadovaný soubor neexistuje
NoSuchMethodException	požadovaná metoda neexistuje
IOException	během vstupního/výstupního zpracování došlo k výjimce
SQLException	při komunikaci s databázovým systémem (SQL) došlo k výjimce

Odvozování od třídy Exception:

od třídy Exception, která je odvozena od třídy Throwable, lze odvozovat své vlastní výjimky

metoda třídy Throwable	**popis**
Throwable fillInStackTrace()	vrací vyvolatelný objekt, obsahující výpis zásobníku
String getLocalizedMessage()	vrací lokalizovaný popis výjimky
String getMessage()	vrací popis výjimky
void printStackTrace(PrintWriter stream)	posílá výpis zásobníku do specifického proudu (stream)
String toString()	vrací objekt String, obsahující popis výjimky, metodu volá funkce println() při odesílání objektu Throwable na výstup

```
class DeleniNulou extends Exception
{
     private int podrobnosti;
     DeleniNulou()
     {
          podrobnosti = 0;
     }
     DeleniNulou(int a)
     {
          podrobnosti = a;
     }
     public String toString()
     {
          return „DeleniNulou[„ + podrobnosti + „] „;
     }
}
class Program
{
     static void vypocet(int a) throws DeleniNulou
     {
          int b = 10, c;
          if(a == 0)
          {
               throw new DeleniNulou (a);
          }
          else
          {
               c = b/a;
               System.out.println(„Vysledek: „ + a);
          }
     }
     public static void main(String arg[])
     {
          try
          {
               vypocet(1);
               vypocet(0);
          }
          catch(DeleniNulou v)
          {
               System.out.println(„Chyba: „ + v);
          }
     }
}
```

10_Vlákna

Paralelní zpracování úloh:

na bázi procesů: probíhá více programů (např. Word a Total Commander)
na bázi vláken: zpracování v rámci jednoho procesu (např. psaní textu a kontrola pravopisu)
režie: práce potřebná k řízení paralelního zpracování
vlákna: části programu, které běží paralelně, vlákna se zpracovávají asynchronně, nezávisle na ostatních vláknech, vlákna lze synchronizovat definováním synchronizační metody, vlákna mají svoji prioritu

pravidla pro přepínání kontextu (přepínání mezi procesy resp. vlákny): jedno vlákno může dát přednost druhému (s nejvyšší prioritou), vlákno s vyšší prioritou může přerušit vykonávání vlákna s nižší prioritou bez ohledu na to co právě provádí (preemptivní paralelní zpracování), vlákna se stejnou prioritou se zpracovávají podle pravidel operačního systému (např. Windows používá dávkování času, tzn. že se provádění cyklicky střídá po několika milisekundových intervalech)

stavy vlákna	**popis**
běžící	vlákno se provádí
pozastavené	provádění vlákna je pozastaveno a může se obnovit od místa, kde bylo zastaveno
blokované	ke zdroji nelze přistoupit, protože jej používá jiné vlákno
ukončené	provádění bylo zastaveno a nelze jej obnovit

Hlavní vlákno:

každý program má hlavní vlákno, z něj vznikají dceřinná vlákna

```
class Program
{
     public static void main(String arg[])
     {
          Thread v = Thread.currentThread();
          System.out.println(„Aktualni vlakno: „ + v);
          v.setName(„Hlavni vlakno“);
          System.out.println(„Prejmenovane vlakno: „ + v);
     }
}
```

popis: deklaruje se odkaz na hlavní vlákno, které se přejmenuje, na obrazovku se vypíše:
Thread[Hlavni vlakno,5,main], kde Hlavni vlakno je název vlákna, 5 je priorita vlákna (kde 1 je nejnižší a 10 nejvyšší priorita) a main je jméno skupiny vláken

Vytvoření vlastního vlákna:

implementací rozhraní Runnable:

```
class MojeVlakno implements Runnable
{
     Thread v;
     MojeVlakno()
     {
          v = new Thread(this,“Moje vlakno“);
          v.start();
     }
     public void run()
     {
          System.out.println(„Dcerinne vlakno spusteno“);
          System.out.println(„Dcerinne vlakno ukonceno“);
     }
}
class Program
{
     public static void main(String arg[])
     {
          new MojeVlakno();
          System.out.println(„Hlavni vlakno spusteno“);
          System.out.println(„Hlavni vlakno ukonceno“);
     }
}
```

popis: metoda start() volá metodu run(), ve které jsou definovány příkazy nově vytvořeného vlákna

odvozením třídy od třídy Thread:

```
class MojeVlakno extends Thread
{
     MojeVlakno()
     {
          super(„Moje vlakno“);
          start();
     }
     public void run()
     {
          System.out.println(„Dcerinne vlakno spusteno“);
          System.out.println(„Dcerinne vlakno ukonceno“);
     }
}
class Program
{
     public static void main(String arg[])
     {
          new MojeVlakno();
          System.out.println(„Hlavni vlakno spusteno“);
          System.out.println(„Hlavni vlakno ukonceno“);
     }
}
```

popis: při odvozování od třídy Thread se musí překrýt metoda run(), ve které jsou definovány příkazy nově vytvořeného vlákna

je-li metoda run() jedinou metodou, kterou budeme ve třídě Thread překrývat, měli bychom implementovat rozhraní Runnable, odvozovat třídu od třídy Thread bychom měli jen v případě, že budeme překrývat více metod definovaných v této třídě

metoda třídy Thread	**popis**
getName()	vrací název vlákna
getPriority()	vrací prioritu vlákna
isAlive()	zjišťuje, zda vlákno běží
join()	pozastaví provádění až do ukončení vlákna
run()	vstupní bod do vlákna
sleep()	pozastaví vlákno, lze specifikovat interval, po který je vlákno pozastaveno
start()	spustí vlákno

Použití více vláken:

```
class MojeVlakno implements Runnable
{
     String JmenoVlakna;
     Thread vlakno;
     MojeVlakno (String threadName)
     {
          JmenoVlakna = threadName;
          vlakno = new Thread (this, JmenoVlakna);
          vlakno.start();
     }
     public void run()
     {
          try
          {
               System.out.println(„Spusteni Vlakna: “ + JmenoVlakna);
               Thread.sleep(2000);
          }
          catch(InterruptedException v)
          {
               System.out.println(„Vyjimka: vlakno „ + JmenoVlakna +
               „preruseno“);
          }
          System.out.println(„Ukonceni Vlakna: “ + JmenoVlakna);
     }
}
class Program
{
     public static void main(String arg[])
     {
          new MojeVlakno(„1“);
          new MojeVlakno(„2“);
          new MojeVlakno(„3“);
          new MojeVlakno(„4“);
          try
          {
               Thread.sleep(10000);
          }
          catch(InterruptedException v)
          {
               System.out.println(„Vyjimka: Hlavni vlakno preruseno“);
          }
          System.out.println(„Ukonceni Vlakna: hlavni vlakno“);
     }
}
```

popis: vytvoříme 4 instance jednoho vlákna, metoda sleep(2000) pozastaví vlákno na 2000 milisekund tzn. 2 sekundy

Metody isAlive() a join():

```
class MojeVlakno implements Runnable
{
     String JmenoVlakna;
     Thread vlakno;
     MojeVlakno (String threadName)
     {
          JmenoVlakna = threadName;
          vlakno = new Thread (this, JmenoVlakna);
          vlakno.start();
     }
     public void run()
     {
          try
          {
               System.out.println(„Spusteni Vlakna: “ + JmenoVlakna);
               Thread.sleep(2000);
          }
          catch(InterruptedException v)
          {
               System.out.println(„Vyjimka: vlakno „ + JmenoVlakna +
               „preruseno“);
          }
          System.out.println(„Ukonceni Vlakna: “ + JmenoVlakna);
     }
}
class Program
{
     public static void main(String arg[])
     {
          MojeVlakno vlakno1 = new MojeVlakno(„1“);
          MojeVlakno vlakno2 = new MojeVlakno(„2“);
          MojeVlakno vlakno3 = new MojeVlakno(„3“);
          MojeVlakno vlakno4 = new MojeVlakno(„4“);
          System.out.println(„Stav vlakna: isAlive:“);
          System.out.println(„Vlakno 1: “ +vlakno1.vlakno.isAlive());
          System.out.println(„Vlakno 2: “ +vlakno2.vlakno.isAlive());
          System.out.println(„Vlakno 3: “ +vlakno3.vlakno.isAlive());
          System.out.println(„Vlakno 4: “ +vlakno4.vlakno.isAlive());
          try
          {
               System.out.println(„Vlakno: join“);
               vlakno1.vlakno.join();
               vlakno2.vlakno.join();
               vlakno3.vlakno.join();
               vlakno4.vlakno.join();
          }
          catch(InterruptedException v)
          {
               System.out.println(„Vyjimka: Hlavni vlakno preruseno“);
          }
```

```
            System.out.println(„Stav vlakna: isAlive: “);
            System.out.println(„Vlakno 1: “ +vlakno1.vlakno.isAlive());
            System.out.println(„Vlakno 2: “ +vlakno2.vlakno.isAlive());
            System.out.println(„Vlakno 3: “ +vlakno3.vlakno.isAlive());
            System.out.println(„Vlakno 4: “ +vlakno4.vlakno.isAlive());
            System.out.println(„Ukonceni Vlakna: hlavni vlakno“);
      }
}
```

popis: metoda isAlive() vrací booleanovskou hodnotu true, jestliže dané vlákno stále běží, metoda join() čeká, dokud se dceřinný proces neukončí a nepřidá se k hlavnímu vláknu (lze určit čas, po který má metoda čekat na ukončení dceřinného procesu)

Priorita vlákna:

```
class MojeVlakno implements Runnable
{
      Thread v;
      private volatile boolean spusteno = true;
      public MojeVlakno (int p, String tName)
      {
            v = new Thread(this, tName);
            v.setPriority(p);
      }
      public void run()
      {
            System.out.println(v.getName() + „ bezi“);
      }
      public void stop()
      {
            spusteno = false;
            System.out.println(v.getName() + „ zastaveno“);
      }
      public void start()
      {
            System.out.println(v.getName() + „ spusteno“);
            v.start();
      }
}
class Program
{
      public static void main(String arg[])
      {
            Thread.currentThread().setPriority(10);
            MojeVlakno lowPriority=new MojeVlakno(3,„Vlakno s nizkou
            prioritou“);
            MojeVlakno highPriority=new MojeVlakno(3,„Vlakno s vysokou
            prioritou“);
            lowPriority.start();
            highPriority.start();
```

```
            try
            {
                 Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {
                 System.out.println(„Hlavni vlakno preruseno“);
            }
            lowPriority.stop();
            highPriority.stop();
            try
            {
                 lowPriority.v.join();
                 highPriority.v.join();
            }
            catch(InterruptedException v)
            {
                 System.out.println(„Vyjimka InterruptedException“);
            }
      }
}
```

popis: prioritu nastavujeme pomocí metody setPriority() třídy Thread, argumentem je číslo reprezentující prioritu (kde 1 je nejnižší a 10 nejvyšší priorita), nebo lze použít konstanty MIN_PRIORITY, MAX_PRIORITY, NORM_PRIORITY, prioritu zjistíme pomocí metody getPriority(), vlákno s vyšší prioritou obírá vlákno s nižší prioritou o zdroje, tzn. že vlákno s nižší prioritou čeká, dokud vlákno s vyšší prioritou neukončí používání zdroje

Synchronizace vláken:

vlákna se synchronizují za použití monitoru (semaforu), který vláknu zpřístupňuje zdroj, monitor může v daný okamžik používat jen jedno vlákno, vlákno smí vlastnit monitor jen tehdy, jestliže jej nevlastní žádné jiné vlákno, pokud je monitor k dispozici, vlákno si jej přivlastní a získá tak přístup ke zdroji

pomocí synchronizované metody:

```
class Zavorky
{
     synchronized void zobrazit(String s)
     {
          System.out.print(„(„);
          try
          {
               Thread.sleep(1000);
          }
          catch(InterruptedException e)
          {
               System.out.println(„Preruseno“);
          }
          System.out.print(s);
```

```
          try
          {
               Thread.sleep(1000);
          }
          catch(InterruptedException e)
          {
               System.out.println(„Preruseno“);
          }
          System.out.println(„)„);
     }
}
class MojeVlakno implements Runnable
{
     String s1;
     Zavorky z1;
     Thread v;
     public MojeVlakno(Zavorky z2, String s2)
     {
          z1 = z2;
          s1 = s2;
          v = new Thread(this);
          v.start();
     }
     public void run()
     {
          z1.zobrazit(s1);
     }
}
class Program
{
     public static void main(String arg[])
     {
          Zavorky z3 = new Zavorky();
          MojeVlakno jmeno1 = new MojeVlakno(z3, „Petr“);
          MojeVlakno jmeno2 = new MojeVlakno(z3, „Pavel“);
          try
          {
               jmeno1.v.join();
               jmeno2.v.join();
          }
          catch(InterruptedException v)
          {
               System.out.println(„Preruseno“);
          }
     }
}
```

popis: synchronizací metody zobrazit() předejdeme tomu, že dvě vlákna budou současně používat jeden zdroj (metodu zobrazit()), jestliže modifikujeme hodnotu proměnné ve dvou různých vláknech, měl by být kód manipulující s proměnnou umístěn do synchronizované metody nebo do synchronizovaného příkazu

pomocí synchronizovaného příkazu:

```
class MojeVlakno implements Runnable
{
     String s1;
     Zavorky z1;
     Thread v;
     public MojeVlakno(Zavorky z2, String s2)
     {
          z1 = z2;
          s1 = s2;
          v = new Thread(this);
          v.start();
     }
     public void run()
     {
          synchronized(z1)
          {
               z1.zobrazit(s1);
          }
     }
}
```

popis: vychází z předchozího příkladu, místo synchronizované metody použijeme synchronizovaný příkaz v metodě run()

Meziprocesová komunikace (komunikace mezi vlákny):

```
class Fronta
{
     int vymenovanaHodnota;
     boolean obsazeno = false;
     synchronized int vyjmout()
     {
          if(!obsazeno)
          {
               try
               {
                    wait();
               }
               catch(InterruptedException v)
               {
                    System.out.println(„Vyjmout:
                    InterruptedException“);
               }
          }
          obsazeno = false;
          System.out.println(„Vyjmout: „ + vymenovanaHodnota);
          notify();
          return vymenovanaHodnota;
     }
     synchronized void vlozit(int vymenovanaHodnota)
     {
          if(obsazeno)
          {
               try
               {
                    wait();
               }
               catch(InterruptedException v)
               {
                    System.out.println(„Vlozit:
                    InterruptedException“);
               }
          }
          this.vymenovanaHodnota = vymenovanaHodnota;
          obsazeno = true;
          System.out.println(„Vlozit: „ + vymenovanaHodnota);
          notify();
     }
}
```

```
class Vydavatel implements Runnable
{
     Fronta f;
     Vydavatel(Fronta f)
     {
          this.f = f;
          new Thread(this, „Vydavatel“) .start();
     }
     public void run()
     {
          for(int i = 0; i < 5; i++)
          {
               f.vlozit(i);
          }
     }
}
class Zakaznik implements Runnable
{
     Fronta f;
     Zakaznik (Fronta f)
     {
          this.f = f;
          new Thread(this, „Zakaznik“) .start();
     }
     public void run()
     {
          for(int i = 0; i < 5; i++)
          {
               f.vyjmout();
          }
     }
}
class Program
{
     public static void main(String arg[])
     {
          Fronta f = new Fronta();
          new Vydavatel(f);
          new Zakaznik(f);
     }
}
```

popis: komunikaci mezi vlákny zajišťují metody wait(), notify() a notifyAll(), metody se volají uvnitř synchronizované metody, metoda wait() zajišťuje uvolnění monitoru a pozastavení činnosti vlákna (lze specifikovat interval, po který je vlákno v čekacím stavu), metoda notify() obnoví chod vlákna, které bylo pozastaveno metodou wait(), metoda notifyAll() probudí všechna vlákna čekající na monitor a vlákno s nejvyšší prioritou monitor získá, pozor: když se všechna vlákna dostanou do čekacího stavu, dojde k zablokování programu

Pozastavení a obnovení činnosti vlákna:

```
class MojeVlakno implements Runnable
{
     Thread v;
     boolean zastaven;
     MojeVlakno()
     {
          v = new Thread(this, „Vlakno“);
          zastaven = false;
          v.start();
     }
     public void run()
     {
          try
          {
               for(int i = 0; i < 10; i++)
               {
                    System.out.println(„Vlakno: „ + i);
                    Thread.sleep(200);
                    synchronized(this)
                    {
                         while(zastaven)
                         {
                              wait();
                         }
                    }
               }
          }
          catch(InterruptedException e)
          {
               System.out.println(„Vlakno: preruseno“);
          }
          System.out.println(„Vlakno ukonceno“);
     }
     void zastavitChod()
     {
          zastaven = true;
     }
     synchronized void obnovitChod()
     {
          zastaven = false;
          notify();
     }
}
```

```
class Program
{
     public static void main(String arg[])
     {
          MojeVlakno v1 = new MojeVlakno();
          try
          {
               Thread.sleep(1000);
               v1.zastavitChod();
               System.out.println(„Vlakno: chod zastaven“);
               Thread.sleep(1000);
               v1.obnovitChod();
               System.out.println(„Vlakno: chod obnoven“);
          }
          catch(InterruptedException v)
          {
          }
          try
          {
               v1.v.join();
          }
          catch(InterruptedException v)
          {
               System.out.println(„Hlavni vlakno: preruseno“);
          }
     }
}
```

11_Soubory a proudy

Třída File (balíček java.io):

```
File soubor1 = new File(String adresar);
File soubor1 = new File(String adresar, String jmenoSouboru);
File soubor1 = new File(File adresarovyObjekt, String jmenoSouboru);
```

popis: ukazatele na adresář nebo soubor

metoda třídy File	**popis**
isFile()	vrací booleanovskou hodnotu true, jestliže je objektem soubor
isAbsolute()	vrací booleanovskou hodnotu true, má-li soubor absolutní cestu
boolean renameTo(File noveJmeno)	přejmenuje adresář nebo soubor
delete()	odstraní soubor z disku
void deleteOnExit()	Odstraňuje soubor při ukončení javového virtuálního stroje
boolean isHidden()	vrací booleanovskou hodnotu true, je-li adresář nebo soubor skrytý
boolean setLastModified(long ms)	nastaví časové razítko souboru (datum a čas v milisekundách)
boolean setReadOnly()	nastaví soubor jen ke čtení
compareTo()	porovná dva soubory
length()	vrací délku souboru v bajtech
is Directory()	vrací booleanovskou hodnotu true, jestliže je objektem adresář
canRead()	vrací booleanovskou hodnotu true, má-li objekt povolení ke čtení
canWrite()	vrací booleanovskou hodnotu true, má-li objekt povolení k zápisu
exists()	vrací booleanovskou hodnotu true, jestliže objekt existuje
getParent()	vrací jméno rodičovského adresáře, který obsahuje podadresář
getAbsolutePath()	vrací absolutní cestu
getPath()	vrací cestu do adresáře
getName()	vrací jméno objektu (adresářové cesty nebo souboru)

```
import java.io.*;
public class Program
{
     public static void main(String arg[])
     {
          File soubor = new File(„\\test\\zkouska“);
          System.out.println(„Jmeno: „ + soubor.getName());
          System.out.println(„Cesta: „ + soubor.getPath());
          System.out.println(„Absolutni cesta: „ +
          soubor.getAbsolutePath());
          System.out.println(„Rodic: „ + soubor.getParent());
          System.out.println(„Existuje: „ + soubor.exists());
          System.out.println(„Zapis: „ + soubor.canWrite());
          System.out.println(„Cteni: „ + soubor.canRead());
          System.out.println(„Adresar: „ + soubor.isDirectory());
          System.out.println(„Soubor: „ + soubor.isFile());
          System.out.println(„Absolutni: „ + soubor.isAbsolute());
          System.out.println(„Delka: „ + soubor.length());
     }
}
```

popis: systém Windows používá \ zatímco Linux /, java převádí lomítka v případě potřeby

Výpis souborů obsažených v adresáři:

```
import java.io.*;
public class Program
{
     public static void main(String arg[])
     {
          String adresar = „/test“;
          File soubor1 = new File(adresar);
          if(soubor1.isDirectory())
          {
               System.out.println(„Adresar: „ + adresar);
               String str[] = soubor1.list();
               for(int i = 0;i<str.length;i++)
               {
                    File soubor2 = new File(adresar + „/“ + str[i]);
                    if(soubor2.isDirectory())
                    {
                         System.out.println(„Adresar: „ + str[i]);
                    }
                    else
                    {
                         System.out.println(str[i]);
                    }
               }
          }
          else
          {
               System.out.println(„Nejedna se o adresar“);
          }
     }
}
```

popis: obsah adresáře získáme voláním metody list(), metoda vrací pole řetězců s názvy souborů

Proudy:

program ukládá data třemi způsoby: jako samostatnou část dat, která není zapouzdřena do třídy, jako data, která jsou zapouzdřena do třídy nebo jako data uložená v databázi (viz. Databáze)

data, která nejsou zapouzdřena do třídy:

zápis do souboru:

```
import java.io.*;
import java.util.*;
public class Program
{
     public static void main(String arg[])
     {
          String jmeno = "Petr";
          String prijmeni = " Novak";
          String hodnoceni = "A";
          try
          {
               PrintWriter vystup = new PrintWriter(new
FileOutputStream("Student.dat"));
               vystup.print(jmeno);
               vystup.println(prijmeni);
               vystup.print(hodnoceni);
               vystup.close();
          }
          catch (IOException v)
          {
               System.out.println(v);
          }
     }
}
```

popis: pro zápis do souboru je třeba vytvořit souborový výstupní proud pomocí konstruktoru třídy FileOutputStream, kterému předáme název souboru (příp. plnou cestu k souboru), konstruktor vrací odkaz na souborový výstupní proud, data zapisujeme pomocí třídy PrintWriter, vytvoříme její instanci a do konstruktoru předáme jako argument odkaz na výstupní proud, konstruktor vrátí odkaz na instanci třídy PrintWriter, odkaz použijeme pro volání metod print() a println(), které zapisují do souboru, nakonec voláme metodu close(), která soubor uzavře

čtení ze souboru:

```
import java.io.*;
import java.util.*;
public class Program
{
     public static void main(String arg[])
     {
          String radek;
          try
          {
               BufferedReader vstup = new BufferedReader(new
               FileReader(„Student.dat“));
               while((radek = vstup.readLine()) != null)
               {
                    System.out.println(radek);
               }
               vstup.close();
          }
          catch (IOException v)
          {
               System.out.println(v);
          }
     }
}
```

popis: soubor, ze kterého čteme, otevřeme vytvořením instance třídy FileReader a pomocí konstruktoru předáme název a cestu k souboru, vytvořením instance třídy BufferedReader alokujeme úsek paměti jako vyrovnávací paměť, do které se ukládají data načtená z disku, podobnou techniku lze použít i pro zápis využitím třídy BufferedWriter

zápis na konec souboru:

```
import java.io.*;
import java.util.*;
public class Program
{
     public static void main(String arg[])
     {
          String jmeno = "Petr";
          String prijmeni = " Novak";
          String hodnoceni = "A";
          try
          {
               PrintWriter vystup = new PrintWriter(new
               FileOutputStream("Student.dat", true));
               vystup.print(jmeno);
               vystup.println(prijmeni);
               vystup.print(hodnoceni);
               vystup.close();
          }
          catch (IOException v)
          {
               System.out.println(v);
          }
     }
}
```

popis: pro zápis dat až za poslední bajt v souboru nastavíme druhý argument ve volání konstruktoru třídy FileOutputStream na hodnotu true

data, která jsou zapouzdřena do třídy:

```
import java.io.*;
import java.util.*;
public class Program
{
     public static void main(String arg[])
     {
          Student[] zapisStudentInfo = new Student[3];
          Student[] cteniStudentInfo = new Student[3];
          zapisStudentInfo[0] = new Student(„Petr“,“Novak“,“B“);
          zapisStudentInfo[1] = new Student(„Anna“,“Svobodova“,“A“);
          zapisStudentInfo[2] = new Student(„Pavel“,“Cech“,“B“);
          try
          {
               ObjectOutputStream vystup = new ObjectOutputStream(new
               FileOutputStream(„Objekt.dat“,true));
               for(int x = 0; x < 3; x++)
               {
                    vystup.writeObject(zapisStudentInfo[x]);
               }
               vystup.close();
               ObjectInputStream vstup = new ObjectInputStream(new
               FileInputStream(„Objekt.dat“));
               for(int y = 0; y < 3; y++)
               {
                    cteniStudentInfo[y] = (Student)
               vstup.readObject();
               }
               vstup.close();
               for(int z = 0; z < 3; z++)
               {
                    System.out.println(cteniStudentInfo[z].
                    studentJmeno);
                    System.out.println(cteniStudentInfo[z].
                    studentPrijmeni);
                    System.out.println(cteniStudentInfo[z].
                    studentHodnoceni);
               }
          }
          catch (Exception v)
          {
               System.out.println(v);
          }
     }
}
```

```
class Student implements Serializable
{
     String studentJmeno, studentPrijmeni, studentHodnoceni;
     public Student() {};
     public Student(String jmeno, String prijmeni, String hodnoceni)
     {
          studentJmeno = jmeno;
          studentPrijmeni = prijmeni;
          studentHodnoceni = hodnoceni;
     }
}
```

popis: instance třídy Student se uloží do souboru tak, že se uloží všechny instanční proměnné třídy, třídě Student implementujeme rozhraní Serializable, které převádí (serializuje) instance třídy na proud bajtů, který lze zapisovat na disk nebo přenášet po síti, při čtení objektu se proud bajtů naopak deserializuje, objekt zapíšeme do souboru pomocí metody writeObject() třídy ObjectOutputStream a předáme do ní odkaz na instanci objektu, statické proměnné se neukládají, nakonec voláme metodu close() a soubor uzavřeme, objekt načteme voláním metody readObject() třídy ObjectInputStream, která vrací instanci třídy Object, která se konvertuje na typ Student

12_GUI

Uživatelské rozhraní:

```
import java.io.*;
public class Program
{
     public static void main(String arg[])
     {
          BufferedReader stdin = new BufferedReader(new
          InputStreamReader(System.in));
          try
          {
               System.out.print("Zadejte ID studenta: ");
               String IDstudenta = stdin.readLine();
               System.out.println("ID: " + IDstudenta);
          }
          catch(IOException v)
          {
               System.out.println("Vyjimka: " + v);
          }
     }
}
```

popis: metoda System.in se používá při odkazování se na standardní vstup počítače, tedy klávesnici, program používá objekty InputStreamReader a BufferedReader ke zlepšení procesu čtení informací z klávesnice, informace se ukládají do posloupnosti bajtů a objekt InputStreamReader bajty převádí na znaky, třída BufferedReader zajistí čtení znaků z objektu třídy InputStreamReader a ukládání do vyrovnávací paměti pro rychlé zpracování informací, metoda readLine() třídy BufferedReader pak přečte z paměti celý řádek najednou

Jednoduché rozhraní GUI a zprávy:

první GUI (graphical user interface) vyvinuli vývojáři firmy Xerox
kolekce tříd rozhraní GUI obsahují balíčky: javax.swing (Java Foundation Classes – JFC), java.awt (Abstract Window Toolkit – AWT)

```
JOptionPane.showMessageDialog(null, “Zprava“, “Nadpis okna“,
JOptionPane.PLAIN_MESSAGE);
```

popis: zobrazí dialogové okno, první argument je odkaz na rodiče, který volá metodu, druhý argument je zpráva, která se zobrazí, třetí argument je nadpis okna a čtvrtý argument je typ zprávy

konstanta určující typ zprávy	**popis**
JOptionPane.PLAIN_MESSAGE	obecná zpráva (bez ikony)
JOptionPane.ERROR_MESSAGE	chybová zpráva
JOptionPane.INFORMATION_MESSAGE	informační zpráva
JOptionPane.WARNING_MESSAGE	varující zpráva
JOptionPane.QUESTION_MESSAGE	dotazovací zpráva

```
String str = JOptionPane.showInputDialog(„Zadej text: „);
```

popis: zobrazí vstupní dialogové okno, argumentem je zpráva, která se zobrazí, metoda vrací řetězec typu String, zobrazené okno obsahuje tlačítko OK, které potvrdí zadanou informaci a Cancel, které okno uzavře

```
import javax.swing.*;
public class Program
{
     public static void main (String arg[])
     {
          String str;
          str = JOptionPane.showInputDialog("Zadejte ID studenta: ");
          JOptionPane.showMessageDialog(null, "ID studenta je:\n" +
          str, "Identifikacni kod", JOptionPane.INFORMATION_MESSAGE);
          System.exit(0);
     }
}
```

Vytvoření okna:

```
import javax.swing.*;
import java.awt.*;
public class Program
{
     public static void main(String arg[])
     {
          Okno mojeOkno = new Okno();
     }
}
class Okno extends JFrame
{
     public Okno()
     {
          super(„Nadpis okna“);
          setSize(640, 480);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
          Container kon = getContentPane();
          kon.setBackground(Color.orange);
     }
}
```

popis: okno vytvoříme vytvořením instance třídy odvozené od třídy JFrame z knihovny JFC, titulek okna se vytvoří při volání konstruktoru třídy JFrame pomocí klíčového slova super, argumentem je nadpis okna, metoda setSize() třídy JFrame definuje velikost okna, metoda setDefaultCloseOperation() definuje chování okna při jeho zavření uživatelem a metoda setVisible() zajišťuje zobrazení okna, dále je potřeba vytvořit kontejner, aby bylo možné do okna vkládat prvky GUI, kontejner je definován třídou Container z knihovny AWT, instanci třídy Container deklarujeme uvnitř třídy okna voláním metody getContentPane() třídy JFrame, metoda vrací odkaz na kontejner, který se uloží do proměnné typu Container, metoda setBackground definuje barvu kontejneru, argumentem je konstanta třídy Color určující barvu

konstanta barvy	barva
Color.white	bílá
Color.lightGray	světle šedá
Color.gray	šedá
Color.black	černá
Color.blue	modrá
Color.green	zelená
Color.cyan	tyrkysová
Color.magenta	purpurová
Color.orange	oranžová
Color.pink	růžová
Color.red	červená
Color.yellow	žlutá

Správci rozvržení:

Flow Layout Manager:

tento správce rozmisťuje prvky GUI zleva do prava a potom na další řádek, správce dokáže zarovnávat prvky GUI na střed, doleva nebo doprava předáním určité konstanty (FlowLayout.LEFT, FlowLayout.RIGHT) do konstruktoru třídy, výchozí nastavení je zarovnání prvků na střed, dále lze definovat horizonální a vertikální vzdálenost mezi prvky tak, že při volání konstruktoru zadáme číslo, představující počet pixelů

```
import java.awt.*;
import javax.swing.*;
public class Program
{
     public static void main(String arg[])
     {
          Okno mojeOkno = new Okno();
     }
}
class Okno extends JFrame
{
     public Okno()
     {
          super("Nadpis okna");
          setSize(400, 100);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
          Container kon = getContentPane();
          kon.setBackground(Color.yellow);
          FlowLayout rozvrzeni = new FlowLayout();
          kon.setLayout(rozvrzeni);
          JButton tlacitko = new JButton(„Test“);
          kon.add(tlacitko);
          setContentPane(kon);
     }
}
```

Border Layout Manager:

tento správce rozděluje kontejner na pět oblastí: sever, jih, západ, východ, střed, prvky GUI se vkládají do kontejneru pomocí metody add() třídy Container, metoda add() má dva argumenty: odkaz na prvek GUI a konstantu určující polohu prvku v kontejneru, metoda setContentPane() umístí obsah kontejneru do okna, konstanty určující polohu:

```
BorderLayout.NORTH
BorderLayout.SOUTH
BorderLayout.WEST
BorderLayout.EAST
BorderLayout.CENTER

import java.awt.*;
import javax.swing.*;
public class Program
{
     public static void main(String arg[])
     {
          Okno mojeOkno = new Okno();
     }
}
class Okno extends JFrame
{
     public Okno()
     {
          super(„Nadpis okna“);
          setSize(400, 100);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
          Container kon = getContentPane();
          kon.setBackground(Color.red);
          BorderLayout rozvrzeni = new BorderLayout();
          kon.setLayout(rozvrzeni);
          JButton tlacitko1 = new JButton(„Test 1“);
          kon.add(tlacitko1, rozvrzeni.WEST);
          JButton tlacitko2 = new JButton(„Test 2“);
          kon.add(tlacitko2, rozvrzeni.EAST);
          setContentPane(kon);
     }
}
```

Grid Layout Manager a Grid Bag Layout Manager:

správce Grid Layout dělí kontejner na řádky a sloupce, lze nastavit počet řádků a sloupců pomocí dvou argumentů v konstruktoru třídy, všechny buňky mají stejnou velikost, prvky se do buněk rozmisťují zleva doprava

```
GridLayout rozvrzeni = new GridLayout();
```

správce Grid Bag Layout umožňuje umístit prvek GUI do specifické buňky, deklarujeme instanci třídy GridBagConstraints, která zajistí přesné určení polohy prvku v mřížce, nastavíme atributy gridx a gridy, poté zavoláme metodu add() a předáme do ní odkaz na prvek GUI a odkaz na instanci třídy GridBagConstraints

```
import java.awt.*;
import javax.swing.*;
public class Program
{
     public static void main(String arg[])
     {
          Okno mojeOkno = new Okno();
     }
}
class Okno extends JFrame
{
     public Okno()
     {
          super(„Nadpis okna“);
          setSize(400, 100);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
          Container kon = getContentPane();
          kon.setBackground(Color.blue);
          GridBagLayout rozvrzeni = new GridBagLayout();
          GridBagConstraints umisteni = new GridBagConstraints();
          kon.setLayout(rozvrzeni);
          JButton tlacitko1 = new JButton(„Test 1“);
          umisteni.gridx = 1;
          umisteni.gridy = 0;
          kon.add(tlacitko1, umisteni);
          JButton tlacitko2 = new JButton(„Test 2“);
          umisteni.gridx = 1;
          umisteni.gridy = 1;
          kon.add(tlacitko2, umisteni);
          setContentPane(kon);
     }
}
```

Tlačítka:

```
JButton tlacitko = new JButton(„Nápis“);
kontejner.add(tlacitko);
```

popis: třída JButton má v konstruktoru jeden argument: nápis na tlačítku

Popisky a textová pole:

```
JLabel popisek = new JLabel(„Informace o studentovi:“);
kontejner.add(popisek);
```

popis: třída JLabel má v konstruktoru jeden argument: nápis, který se zobrazí

```
JTextField text = new JTextField(„Jméno“, 25);
kontejner.add(text);
```

popis: třída JButton má v konstruktoru dva argumenty: text, který se objeví v textovém poli a délku pole

Přepínače a zaškrtávací políčka:

```
JCheckBox policko = new JCheckBox(„Student“);
kontejner.add(policko);
```

popis: třída JCheckBox má v konstruktoru jeden argument: text, který se objeví vedle políčka

```
ButtonGroup skupina = new ButtonGroup();
JRadioButton prepinac1 = new JRadioButton(„Muž“);
skupina.add(prepinac1);
kontejner.add(prepinac1);
JRadioButton prepinac2 = new JRadioButton(„Žena“);
skupina.add(prepinac2);
kontejner.add(prepinac2);
```

popis: pro přepínače je třeba vytvořit skupinu pomocí třídy ButtonGroup, třída JButton má v konstruktoru jeden argument: text, který se objeví vedle přepínače, dále je potřeba přepínače vložit do skupiny přepínačů, kterou jsme vytvořili, pomocí metody add() třídy ButtonGroup

Rozevírací seznam:

```
JComboBox seznam = new JComboBox();
seznam.addItem(„Jedna“);
seznam.addItem(„Dvě“);
seznam.addItem(„Tři“);
kontejner.add(seznam);
```

popis: pro rozevírací seznam je třeba definovat položky pomocí metody addItem() třídy JComboBox, metoda má v konstruktoru jeden argument: text položky

Textová oblast:

```
JTextArea oblast = new JTextArea(3, 30);
kontejner.add(oblast);
```

popis: třída JTextArea má v konstruktoru dva argumenty: počet řádků a délku oblasti

```
JTextArea oblast = new JTextArea(„Výchozí text“, 3, 30);
oblast.setEditable(false);
kontejner.add(oblast);
```

popis: třída JTextArea má v konstruktoru tři argumenty: text který se objeví v oblasti, počet řádků a délku oblasti, metoda setEditable() třídy JTextArea určuje, zda je možné text editovat

Rolovací panel:

```
JScrollPane panel = new JScrollPane(prvek,
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
```

popis: rolovací panel umožňuje posouvat vertikálně a horizontálně jiný prvek GUI pomocí rolovacích lišt, třída JScrollPane má v konstruktoru tři argumenty: odkaz na prvek GUI a konstanty určující chování vertikální a horizontální rolovací lišty

konstanta rolovacího panelu	**popis**
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED	panel se zobrazí, je-li třeba
JScrollPane.VERTICAL_SCROLLBAR_NEVER	panel se nepoužije
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS	panel se zobrazí vždy
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED	panel se zobrazí, je-li třeba
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER	panel se nepoužije
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS	panel se zobrazí vždy

Získávání dat z prvků GUI:

kód pro získávání dat z prvků GUI obsahuje posluchače, kteří monitorují rozhraní GUI a čekají až se přihodí nějaká událost (např. stisknutí tlačítka), když událost nastane, posluchač ji detekuje a zavolá metodu události, pro použití posluchače je třeba implementovat rozhraní posluchače a přiřadit posluchače ke konkrétnímu prvku GUI

posluchač (rozhraní zachytávající události)	**prvek GUI**
ActionListener	tlačítka
ItemListener	zaškrtávací políčka, přepínače, rozevírací seznamy
KeyListener	vstup z klávesnice
MouseListener	činnost myši
MouseMotionListener	pohyb myši

Obsluha příkazového tlačítka:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class Program
{
     public static void main(String arg[])
     {
          Okno mojeOkno = new Okno();
     }
}
class Okno extends JFrame implements ActionListener
{
     JTextArea oblast1 = new JTextArea("Výchozí text", 5, 25);
     JTextArea oblast2 = new JTextArea(5, 25);
     JButton tlacitko = new JButton("Kopírovat");
     public Okno()
     {
          super("Nadpis okna");
          setSize(800, 150);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
          Container kon = getContentPane();
          FlowLayout rozvrzeni = new FlowLayout();
          kon.setLayout(rozvrzeni);
          tlacitko.addActionListener(this);
          kon.add(tlacitko);
          kon.add(oblast1);
          kon.add(oblast2);
          setContentPane(kon);
     }
     public void actionPerformed(ActionEvent udalost)
     {
          oblast2.setText(oblast1.getText());
     }
}
```

popis: program zkopíruje pomocí tlačítka text z jednoho pole do druhého, je třeba vytvořit posluchače tak, že implementujeme rozhraní ActionListener, dále je třeba k rozhraní přidružit tlačítko zavoláním metody addActionListener () třídy JButton, posluchač zavolá metodu události actionPerformed(), metoda přijímá jeden argument: odkaz na událost, příkazy metody actionPerformed() se provedou po stisknutí kteréhokoliv tlačítka v programu, pro identifikaci konkrétního tlačítka v programu by posloužila metoda udalost.getSource(), která by vrátila odkaz na tlačítko, dále by šel zjistit typ činnosti, kterou uživatel stisknutím tlačítka provedl, k tomu by posloužila metoda udalost.getActionCommand()

Obsluha přepínačů a zaškrtávacích políček:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class Program
{
     public static void main(String arg[])
     {
          Okno mojeOkno = new Okno();
     }
}
class Okno extends JFrame implements ItemListener
{
     JTextArea oblast = new JTextArea("Výchozí text", 5, 25);
     JCheckBox policko1 = new JCheckBox(„Policko 1“);
     JCheckBox policko2 = new JCheckBox(„Policko 2“);
     public Okno()
     {
          super("Nadpis okna");
          setSize(350, 150);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
          Container kon = getContentPane();
          FlowLayout rozvrzeni = new FlowLayout();
          kon.setLayout(rozvrzeni);
          policko1.addItemListener(this);
          policko2.addItemListener(this);
          kon.add(oblast);
          kon.add(policko1);
          kon.add(policko2);
          setContentPane(kon);
     }
     public void itemStateChanged(ItemEvent udalost)
     {
          int stav = udalost.getStateChange();
          if(stav == ItemEvent.SELECTED)
          {
               if(udalost.getItem() == policko1)
                    oblast.setText(„Políčko 1 je zaškrtnuto“);
               if(udalost.getItem() == policko2)
                    oblast.setText(„Políčko 2 je zaškrtnuto“);
          }
          if(stav == ItemEvent.DESELECTED)
          {
               if(udalost.getItem() == policko1)
                    oblast.setText(„Políčko 1 není zaškrtnuto“);
               if(udalost.getItem() == policko2)
                    oblast.setText(„Políčko 2 není zaškrtnuto“);
          }
     }
}
```

popis: program zobrazí informaci o tom zda je políčko zaškrnuto, je třeba vytvořit posluchače tak, že implementujeme rozhraní ItemListener, dále je třeba k rozhraní přidružit políčka zavoláním metody addItemListener() třídy JCheckBox, posluchač zavolá metodu události itemStateChanged(), která zjišťuje změny ohledně zaškrtnutí, metoda přijímá jeden argument: odkaz na událost, metoda getStateChange() vrací celé číslo, které se porovnává s konstantami ItemEvent.SELECTED a ItemEvent.DESELECTED

Obsluha rozevíracího seznamu:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class Program
{
     public static void main(String arg[])
     {
          Okno mojeOkno = new Okno();
     }
}
class Okno extends JFrame implements ItemListener
{
     JTextArea oblast = new JTextArea("Výchozí text", 5, 25);
     JComboBox seznam = new JComboBox();
     public Okno()
     {
          super("Nadpis okna");
          setSize(350, 200);
          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          setVisible(true);
          Container kon = getContentPane();
          FlowLayout rozvrzeni = new FlowLayout();
          kon.setLayout(rozvrzeni);
          seznam.addItemListener(this);
          seznam.addItem(„Volba 1“);
          seznam.addItem(„Volba 2“);
          kon.add(oblast);
          kon.add(seznam);
          setContentPane(kon);
     }
     public void itemStateChanged(ItemEvent udalost)
     {
          String polozka = udalost.getItem().toString();
          oblast.setText(polozka);
     }
}
```

popis: program zobrazí informaci o vybrané položce ze seznamu, je třeba vytvořit posluchače tak, že implementujeme rozhraní ItemListener, dále je třeba k rozhraní přidružit seznam zavoláním metody addItemListener() třídy JComboBox, posluchač zavolá metodu události itemStateChanged(),která zjišťuje změny ohledně přepínání položek, metoda přijímá jeden argument: odkaz na událost, metoda getItem() vrací text, který převedeme na řetězec pomocí metody toString()

Nepřístupné a přístupné prvky GUI:

```
prvek.setEnabled(true);
```

popis: prvek je přístupný

```
prvek.setEnabled(false);
```

popis: prvek je nepřístupný (nelze použít)