

JAVA

Komentáře

```
//poznámka

/* poznámka */

/**
 * popis třídy, metody
 * @param args - popis parametrů
 */
```

Zápis identifikátorů

String, StringBuffer	... zápis třídy, rozhraní
pocet, pocetPrvku	... proměnné
start(), getSize()	... metody
PI, MAX_VALUE	... konstanty
java.lang	... balíky (všechno malým, slova oddělena tečkami)

Anotace

Píší se před definicí metody

@Deprecated	... označení zastaralého kódu
@Override	... překrytí metody
@SuppressWarnings("warning1 warning2")	... vynechat varovné hlášení

Hlavní program

Soubor musí mít název stejný jako hlavní třída.

Hlavní metoda musí mít název *main* dle následujícího příkladu:

```
public class Prvni {
    public static void main (String[] args) {
        int i = 5;
        System.out.println("i = " + i);
        boolean b = false;
        System.out.println("b = " + b);
    }
}
```

Primitive data types

byte, short, int, long	... celá čísla
float, double	... desetinná čísla
boolean	... logická proměnná
char	... znak

JAVA

Zápisy celých čísel

Desítkové nezačínají nulou: 85, 16, 0, 1
Osmičkové začínají nulou: 0126, 015, 0, 01
Šestnáctkové začínají 0x: 0x56, 0x3A, 0x0, 0xCD, 0x1

Konstanty jsou implicitně typu int (32 bitů). Chceme-li ji mít long (64 bitů), je potřeba dát za číslo *L*:

```
long k;  
k = 1234567890123L;
```

Zápisy reálných čísel

15.2, 5e6, -7E+23

Konstanty jsou implicitně typu double (64 bitů). Chceme-li ji mít float (32 bitů), je potřeba dát za číslo *F*.
Třídy float a double mají některé speciální konstanty:

MIN_VALUE, MAX_VALUE	... minimální a maximální rozsah daného typu
POSITIVE_INFINITY, NEGATIVE_INFINITY	... hodnoty nekonečno

Při dělení nulou nebo při překročení rozsahu není zaznamenána výjimka, ale tyto výsledné hodnoty:
Infinity, *-Infinity* (nekonečna) nebo *NaN* (not a number, dělení 0/0).

Je možné je testovat metodami `isInfinite()` a `isNaN()`:

```
double vysledek = Double.MAX_VALUE / 0.0;  
if (Double.isInfinite(vysledek) == true)  
    System.out.println("nekone\u010Dno");
```

Zápisy znaků

Klasicky:	'a', '%', '1', 'S'
Unicode \uXXXX, kde XXXX je šestnáctková číslice:	'\u00E1', '\u000A', '\u0158'
Osmičkovým zápisem \XXX:	'\007', '\011'
Escape sekvence:	
nová řádka	'\n'
návrat na začátek řádky	'\r'
tabulátor	'\t'
zpětné lomítko	'\\'
apostrof	'\''
uvozovky	'\"'
nová stránka	'\f'

Zápis řetězců je obdobný, jen se znaky uzavírají do uvozovek.

Implicitní hodnoty proměnných

Není-li proměnné přiřazena hodnota, je u čísel automaticky přiřazena 0, u typu char hodnota '\u0000' a u logických proměnných hodnota *false*.

JAVA

Konstanty

```
final int MAX = 10;

public class TridaSKonstantou {
    public static final double MAX_HODNOTA = 20.8;
    ... }
```

Výčtový typ (enumerate)

```
public class PouzitiEnum {
    enum Vyucujici {ASISTENT, DOKTOR, DOCENT, PROFESOR};
    public static void main (String[] args) {
        Vyucujici vyucujici = Vyucujici.DOCENT;
        System.out.println("Vyucujici" + vyucujici);
    }
}
```

Přiřazení je výraz (assignment is expression)

```
int j, i=5; //toto ale není vhodný programátorský zápis
if (i == (j=5)) //zde je j přiřazena 5 a pak porovnáno i(5) a j(5)
    System.out.println(i==j);
```

Přetypování a konverze

```
char c = 'A';
int i = (int) c;

(double) i + j //přetypuje se jen proměnná i
```

Operátory ++ a --

Tyto operátory musí být buď před nebo za *l-hodnotou* (l-value). L-hodnota je něco, co má adresu v paměti. Pokud jsou před, hodnota se nejprve upraví a pak přiřadí, po samozřejmě opačně.

```
int i = 5, j = 4;
i++;           //správně
i = 45++;     //špatně, 45 není l-value
i = --j;      //j se o jednu sníží (na 3) a to se přiřadí i
```

Dělení modulo

Jedná se o zbytek po celočíselném dělení

```
int i = 5, j = 13;
j = j / 4;      //celočíselné dělení! j bude 3
j = i % 3;      //j bude 2
```

JAVA

Zkrácený zápis přiřazení a operátorů

```
j += 5;  
j = j + 5;
```

Jedná se o identitickou instrukci.

Relační operátory

==	rovná se
!=	nerovná se
&&	AND
	OR
!	NOT
<, <=, >, >=	menší, menší nebo rovno, větší, větší nebo rovno

Bitové operace

&	AND
	OR
^	XOR
<<	posun bitů doleva
>>	posun bitů doprava znaménkově (nejlevější bit si ponechává hodnotu)
>>>	posun bitů doprava neznaménkově
~	negace bit po bitu, unární operátor

```
byte x = -16, y = -16;  
x >>= 2;    // 1111 0000 se změní na 1111 1100 (-4)  
y >>>= 2;    // 1111 0000 se změní na 0011 1100 (60)
```

Formátování textu výstupu

```
int i = 4, j = 7;  
System.out.println("Soucet je " + (i+j)); //vypíše 11  
System.out.print("Znak % není \"lomitkovy\" znak" + "\n");  
System.out.print("Znak % není 'lomitkovy' znak");
```

```
int i = 5;  
System.out.println(i + '\n'); //vypíše 15!
```

```
System.out.format("i = %d, g = %f%n", i, g); //%n je odřádkování  
System.out.format("i = %7d", i); //doplnění na 7 znaků zleva  
System.out.format("i = %-7d", i); //doplnění na 7 znaků zprava  
System.out.format("i = %+7d", i); //vynucené znaménko + ve výpisu
```

```
int i = 1234;  
System.out.format("i = %,7d", i); //použije oddělovač tisíců
```

V posledním případě je zrada. Formátování nepoužije mezery, ale znak á.

JAVA

Formátování textu výstupu

```
int i = 10;
System.out.format("i = %02X", i);    //vypíše 0A

double i = 123.567;
System.out.format("i = %5.1f", i);    //vypíše 123.6

String s = "Dobry den";
System.out.format("s = %s", s);        //vypíše Dobry den
System.out.format("s = %S", s);        //vypíše DOBRY DEN
System.out.format("s = %-11s", s);    //vypíše Dobry den a 2 mezery
```

Podmínka

```
if (booleanový výraz) příkaz1; else příkaz2;

if (c=ctiznak() != 'A') d = c;    //špatně
if ((c=ctiznak()) != 'A') d = c;  //správně
```

Složený příkaz

Uzavřený do složených závorek { }. Za uzavírací složenou závorkou se nepíše středník.

Ternární operátor

```
booleanový výraz ? výraz1 : výraz2;

System.out.print(i + ((i % 10 == 0) ? "\n" : " "));
```

Cykly

```
while (booleanový výraz) příkaz;
do { příkazy; } while (booleanový výraz); //repeat s opačnou podmínkou
for (výraz_start; výraz_stop; výraz_iterace) příkaz;

while (true) {
    if ((c = sc.nextLine().charAt(0)) < 'a')
        continue;    //skok na konec smyčky, vynucení dalšího průchodu
    if (c == 'z')
        break;        //ukončení cyklu
}

do {
    i--;
} while (i > 0); //cyklus se opustí, až nebude i větší než 0

for (int i = 1; i <= 10; i++) sum += i;
```

JAVA

Cyklus *for* odpovídá této konstrukci *while*:

```
výraz_start;
while (výraz_stop) {
    příkaz;
    výraz_iterace;
}
```

V cyklu *for* můžeme příkazy ve výrazech oddělit čárkou:

```
for (int i=1, sum=0; i < 10; sum += i, i++) ; //je to ale zvěrstvo
System.out.println(sum); //chyba, sum je definované jen pro cyklus
```

Návěští

Za návěštím musí být buď blok příkazů nebo cyklus.

```
hopsem:
{
    for (int i = 1; i < 10; i++) {
        if (i == 5) break hopsem;
        System.out.print(i + " ");
    }
    System.out.print("za cyklem"); //nevypíše se. Jen 1 2 3 4
}
```

```
navesti:
for (int n = 0; n < 4; n++) { //for n = 0 to 3
    for (int m = 0; m < 2; m++) { //for m = 0 to 1
        if ((n == 2) && (m == 0)) continue navesti;
        System.out.print(n + "-" + m + " ");
    }
} //vypíše 0-0 0-1 1-0 1-1 3-0 3-1
```

Příkaz switch

```
switch (znak) { //výraz v závorkách může být jen typu int nebo char
    case 'a' :
    case 'b' :
    case 'c' :
        System.out.println("1");
        break;

    case 'd' :
        System.out.println("2");
        break;

    default :
        System.out.println("3");
        break; //nemusí být
}
```

JAVA

Metody

Metody nemůžou být vnořené jedna do druhé a deklarace nemusí předcházet jejímu použití. Dělí se na statické (metody třídy, klíčové slovo *static*) a metody instance. I když nemá metoda parametry, musí se vypisovat prázdné kulaté závorky při jejím volání i definici. Vždy musí být uveden typ metody (typ návratové hodnoty). Výstupní hodnota se předává příkazem:

return *výraz*;

Po tomto příkazu je metoda ukončena. Návratový typ metody může být *void* (prázdný datový typ). Taková metoda nemusí být ukončena příkazem *return*. Jedná se o proceduru.

```
static int secti(int a, b) {           //špatný zápis  
static int secti(int a, int b) {       //správně
```

Je možné použití přetížení – stejně pojmenované funkce s rozdílnými parametry. Není ale možný proměnný počet parametrů (známé z C++) ani volání odkazem (klíčové slovo *var* v Pascalu).

Proměnné statické a lokální

Statická proměnná je globální proměnná v rámci jedné třídy. Je uvozená klíčovým slovem *static* a nachází se někde mimo těla metod. Doporučená konvence je však psát je před definicemi metod.

Lokální proměnná je definovaná někde uvnitř těla metody a platí jen pro danou metodu nebo blok příkazů ve kterém je deklarovaná. Lokální proměnné nejsou automaticky inicializovány na *0* nebo *false*.

Referenční proměnné

Java má dva nepřimitivní datové typy – pole a objekty. Proměnné těchto typů jsou označovány jako referenční (u polí někdy jako *jméno pole*). Odkazem (reference) však není míněna adresa v paměti. Hodnota neplatného odkazu je *null*.

Pole

```
int[] poleInt;           //deklarace bez přidělení paměti  
poleInt = new int[20];   //délka pole bude 20, odkazováno 0..19
```

Prvky pole jsou automaticky nastaveny na *0* nebo *false*, u referenčních proměnných na *null*.

```
int[] poleInt = new int[20];  
for (int i = 0; i < poleInt.length; i++) //použití atributu length  
    poleInt[i] = i + 1;
```

Pole lze také inicializovat výčtem:

```
int[] cisla = {1, 2, 3, 5, 7};
```

Definovali jsme tak pole pěti čísel typu *int*. Nejedná se ale o klasický výčet, prvky nejsou konstantní.

JAVA

Konstrukce for-each

```
for (int hodnota : poleInt) {  
    suma += hodnota;  
}
```

Jednotlivé prvky pole *poleInt* se budou postupně ukládat do pomocné proměnné *hodnota* (typu *int*) a s ní se bude provádět zadaný příkaz.

Dvourozměrná pole

```
int[][] a = new int[5][4];  
System.out.println("Počet řádků pole: " + a.length);  
System.out.println("Počet sloupců pole: " + a[0].length);
```

Toto pole polí ale nemusí být alokované najednou a nemusí mít stejný počet sloupců:

```
int[][] a = new int[4][];  
for (int i = 0; i < a.length; i++) a[i] = new int[i+1];
```

Inicializace výčtem:

```
int[][] b = {{1, 2, 3}, {11,12}, {21,31,41}};
```

Podobně jako dvourozměrná lze vytvářet i trojrozměrná a vícerozměrná pole.

Konstruktor

Konstruktor je metoda, která má stejný název jako třída a nemá návratovou hodnotu. Je vyvolaná při vytváření instance třídy.

```
public class Obdelnik {  
    public int sirka;           //atribut (member variable, field)  
    public int vyska;           //public = možné zavolat i odjinud  
  
    public Obdelnik(int pomSirka, int pomVyska) {  
        sirka = pomSirka;  
        this.vyska = pomVyska; //ukázka použití this  
    }  
  
    public int obsah() {         //metoda instance  
        return (sirka * vyska);  
    }  
}
```

A pak můžeme někde zavolat:

```
Obdelnik obd = new Obdelnik(5, 3);
```

I konstruktor je možné přetížit nebo uvést bez parametrů.

Konstruktor není povinný, není-li žádný uveden, použije se implicitní (volaný bez parametrů).

JAVA

Speciální metoda `this()`

V konstruktoru můžeme zavolat jiný konstruktor pomocí *this()*. Tento příkaz však musí být prvním příkazem v těle konstruktoru.

```
public Obdelnik(Obdelnik o) {  
    this(o.sirka, o.vyska);  
}
```

Statické proměnné a metody v třídě a jejich použití pro instance třídy

```
public class Zakaznik {  
    private static int pocetZakazniku = 0; //promenná třídy, neveřejná  
    public int utratil = 0;                //proměnná instance  
  
    public Zakaznik() {  
        Zakaznik.pocetZakazniku++;        //ukázky správných volání  
        this.utratil = 0;  
    }  
  
    public static int kolikZakazniku() {    //metoda třídy  
        return pocetZakazniku;  
    }  
  
    public void platba(int cena) {         //metoda instance  
        utratil += cena;  
    }  
}
```

Statický inicializační blok

Je to složený příkaz uvedený klíčovým slovem *static*. Proveďte se při prvním použití třídy. Slouží pouze k prvotnímu nastavení statických proměnných nebo konstant.

Garbage collector

Proces běžící na pozadí programu. Odstraňuje z paměti objekty na které neukazuje žádná proměnná. Lze jej mimořádně vyvolat příkazem `System.gc()` ;

Finalizer

```
protected void finalize() throws Throwable { //povinný tvar  
    pocetZakazniku--;  
    super.finalize();                        //taky musí být  
}
```

Finalizer je metoda třídy vyvolávaná při likvidaci objektu. Lze ji vynutit zavoláním:

```
System.runFinalization();
```

JAVA

Řetězce

Pro konstantní řetězce je definována třída *String*, pro měnitelné řetězce pak třída *StringBuffer*.

```
String ret = "Ahoj";  
int i = ret.length(); //vrátí 4, neplést length a length()
```

```
String ret1;  
StringBuffer buf = new StringBuffer("Dobrý den");  
ret1 = new String(buf);
```

```
char[] znaky = {'N', 'a', 'z', 'd', 'a', 'r'};  
String ret2 = new String(znaky, 2, 3); //vrátí "zda"
```

```
String[] pole = {"Sbohem", "Nashle", "Čau"};  
int i = pole.length; //vrátí 3
```

Porovnání řetězců

Nelze použít operátor `==`, protože by se porovnály hodnoty referenčních proměnných a ne řetězců.

```
int i = ret1.compareTo(ret2); //porovná lexikograficky  
int i = ret1.compareToIgnoreCase(ret2); //nerozlišuje malé a velké  
boolean b = ret1.equals(ret2); //shoda řetězců  
boolean b = ret1.equalsToIgnoreCase(ret2); //nerozlišuje malé a velké
```

Další metody String (metody instance)

```
String ret1 = ret2.toLowerCase(); //na malé, obdobně toUpperCase
```

```
String ret3 = ret1 + ret2; //lze i funkcí concat
```

```
String s2, s1 = "cacao";  
s2 = s1.replace('c', 'k'); //s2 bude "kakao", s1 beze změny
```

```
String s3 = s1.substring(2); //od 2 do konce, tedy "cao"  
String s4 = s1.substring(1, 4); //od 1 do 4-1, tedy "aca"
```

```
char[] znaky = new char[10];  
String s = "Koloseum";  
s.getChars(2, 5, znaky, 0); //do znaky od indexu 0 dá "los"
```

```
boolean b = s.startsWith("Ko"); //test začátku, obdobně endsWith
```

```
char znak = s.charAt(6); //vrátí 'u'
```

```
int i = s.indexOf('u'); //hledání znaku nebo řetězce  
int i = s.indexOf("Ko", 3); //hledání od třetího znaku  
int i = s.lastIndexOf("e"); //hledání od konce směrem na začátek
```

```
String s3 = s1.trim(); //ořez „bílých“ znaků na začátku a na konci
```

JAVA

Konverze z řetězce a na řetězec

Pro konverzi na řetězec se používají metody třídy *valueOf()* a *format()*. Postačí však i součet s prázdným řetězcem.

```
String s = "" + i; //obdobně: String s = String.valueOf(i);
```

Pro opačnou konverzi se používají metody *parse*. Parametrem je možné určit i soustavu.

```
double cislo1 = Double.parseDouble("3.14");
long cislo2 = Long.parseLong("1AC3", 16); //převod z hexa
boolean b = Boolean.valueOf("true").booleanValue();
```

Řetězení metod

Protože metody často vrací instanci třídy, je možné opět použít metodu instance.

```
String s = "\r\n\t cacao\t \r\n";
int i = s.trim().toUpperCase().substring(2).indexOf('O'); //i bude 2
```

Metoda split()

```
String radka = "1;2;458;89;AA"; //například csv
String[] podretezce = radka.split(";"); //vytvoří pole pěti řetězců
```

Pokud je oddělovačů více, uzavírají se do hranatých závorek.

```
String radka = "1,2 458;89.AA";
String[] podret = radka.split("[, ;.<]"); //znak < se nepoužije
```

Výjimkou je tečka. I když je jediná, přesto se musí uzavřít do hranatých závorek.

```
String nazev = "soubor.ext";
String[] podret = radka.split("[.]");
```

V dokumentaci k `java.util.regex.Pattern` je uvedena oblast výrazů POSIX character classes. Například pro všechny interpunkční znaky je pak zápis tento:

```
String radka = "1,2 458[89.AA";
String[] podret = radka.split("\\p{Punct}"); //takže i znaky [ a ]
```

Metoda toString()

Metoda *toString()* je v každém objektu a slouží pro jeho identifikaci. Je doporučeno její překrytí vlastní metodou, například za použití metod *getClass()* a *getName()*

```
public String toString() {
    String jmenoTridy = new String(getClass().getName());
    return (jmenoTridy + ": "); //následované třeba výpisem atributů
}
```

JAVA

Třída StringBuffer

```
StringBuffer b1, b2, b3;  
b1 = new StringBufer();           //vytvoří řetězec o kapacitě 16 znaků  
b2 = new StringBufer(100);        //vytvoří řetězec o kapacitě 100 znaků  
b3 = new StringBufer("Ahoj");     //vytvoří řetězec o kapacitě 20 znaků
```

V posledním případě prostě Java přidala 16 znaků jako rezervu. Kapacitu a délku lze měnit pomocí metod *ensureCapacity()* a *setLength()*. Pokud zvětšíme délku, neinicializované znaky se nastaví na `\u0000`. Aktuální délku a kapacitu řetězce vrací metody *length()* a *capacity()*. Další metody:

```
b3.reverse();                     //obrátil řetězec, tedy bude "johA"  
b3.append(true);                 //přidá „primitiva“ na konec jako text  
b3.delete(1,3);                  //smaže znak od 1 do 3-1, tedy zbude "jAtrue"  
b3.deleteCharAt(0);              //smaže znak z dané pozice, tedy zbude "Atrue"  
b3.insert(0,3.14);               //přidá cokoli na danou pozici, tedy "3.14Atrue"  
b3.replace(0,5,"XX");            //záměna na dané pozici, tedy "XXtrue"  
b3.setCharAt(1,'M');             //záměna znaku na dané pozici, tedy "XMtrue"  
  
String s1, s2, s3;  
s1 = b3.toString();              //převede řetězec na typ String  
s2 = b3.substring(1);            //převede jen znaky od indexu, tedy "Mtrue"  
s2 = b3.substring(1,3);         //převede "Mt"
```

Třída Character

```
boolean b = Character.isDigit('\u0045');    //je číslo?  
boolean b = Character.isLetter('A');  
boolean b = Character.isLetterOrDigit('?');  
boolean b = Character.isLowerCase('č');  
boolean b = Character.isUpperCase('\n');  
boolean b = Character.isWhitespace(' ');  
  
char c, d = 'A';  
c = Character.toLowerCase(d);    //obdobně toUpperCase
```

Modifikátory deklarace třídy

public	... přístupná i mimo balík, kde je deklarována
abstract class	... z této třídy nelze vytvořit instanci, je jen pro dědění
final class	... tuto třídu již nelze dědit

Pokud není uveden modifikátor, je třída neveřejná, děditelná a je možné vytváření jejich instancí. Třída veřejná (*public*) musí být v souboru, který má stejný název.

Předávání parametrů „odkazem“

Jako parametr metodě můžeme předat referenční proměnnou odkazující na nějaký objekt. Změna proměnné instance tohoto objektu pak samozřejmě přetrvá i po ukončení metody. Neměnný je parametr, ne instance na kterou ukazuje.

JAVA

Dědičnost

```
public class Kvadr extends Obdelnik {    //extends je to dědění
    public int hloubka;

    public Kvadr(int sirka, int vyska, int hloubka) {
        super(sirka, vyska);            //volání konstrukturu rodiče
        this.hloubka = hloubka;
    }

    public int obsah() {                  //překrývání
        return (sirka * vyska * hloubka);
    }

    ...    //zde třeba nové metody, další změněné metody, atd.
}
```

Pokud překrýváme metodu z rodiče (tj. vytváříme metodu se stejným názvem) a použijeme jiné parametry, nedojde k překrytí (overriding, hiding), ale jen k přetížení (overloading).

Překrytá metoda však není ztracena. Lze ji vyvolat pomocí klíčového slova *super*.

```
Kvadr kva = new Kvadr(6, 8, 10);
int i = super.obsah();    //volá metody z rodiče, tedy i bude 48
```

Překrývání metody lze také zakázat. K tomu se používá klíčové slovo *final* před deklarací metody.

Pravidla pro konstruktory potomka

1. Pokud existuje v rodiči konstruktory bez parametrů
 - konstruktory potomka může být implicitní
 - konstruktory potomka nemusí mít vazbu na konstruktory rodiče
2. Pokud všechny konstruktory rodiče mají alespoň jeden parametr
 - alespoň jeden konstruktory potomka musí existovat
 - konstruktory potomka jako svůj první příkaz musí volat některý konstruktory rodiče

Abstraktní metoda

Jedná se o metodu rodiče (abstraktní třídy), kterou potomek musí překrýt. Abstraktní metoda má před deklarací klíčové slovo *abstract* a nemá ani žádné tělo.

```
abstract int getI();    //tot' vše
```

Třída object

Jedná se o základní třídu, všechny třídy jsou její potomky. Má těchto 9 metod:

clone(), equals(), hashCode(), finalize(), toString()	...metody, které lze překrýt
getClass(), notify(), notifyAll(), wait()	...finální metody

JAVA

Balíky

Jedná se o skupinu tříd. Jsou-li třídy soubory, pak jsou balíky adresáře.

Cesta k balíkům se zadává v systémové proměnné CLASSPATH. Standardně je tam cesta na Core API.

```
import java.lang.*;           //nemusíme psát, provádí se automaticky
import java.io.FileReader;     //načtení jen jedné třídy z balíku

public class Balik {
    public static void main(String[] args) {
        FileReader f = new FileReader("a.txt"); //třída z balíku java.io
        java.io.LineNumberReader lf = new java.io.LineNumberReader(f);
        System.out.println(lf.readLine());
        f.Close();
    }
}
```

Pokud neprovedeme import, musíme použít plně kvalifikované jméno (*fully qualified name*).

Statický import balíků

Načítání jen metod nebo atributů třídy.

```
import static java.lang.Math.*; //nemusím pak psát Math.PI, stačí PI
import static java.lang.Math.sin; //import jen metody sin()
```

Vytváření balíků

```
package editor; //název vytvářeného balíku
public class Balik {
    public static void main(String[] args) {
        System.out.println("Halo, tady jsem.");
    }
}
```

Při překladu pomocí `javac -d . Balik.java` se vytvoří složka **editor**. K oddělování složky a třídy se používá tečka, ne zpětné lomítko. Spuštění programu tedy probíhá voláním `java editor.Balik`

Přístupová práva

Před jakoukoliv proměnnou či metodou lze uvést některý ze tří specifikátorů přístupu. Pokud není uveden, jedná se o tzv. přátelský přístup. V potomkovi (podtřídě) nelze práva atributů a metod omezit.

Přístup	v téže třídě	v podtřídě téhož balíku	v jiné třídě téhož balíku	v podtřídě jiného balíku	v jiné třídě jiného balíku
private	Ano	Ne	Ne	Ne	Ne
neuvedeno (<i>friend</i>)	Ano	Ano	Ano	Ne	Ne
protected	Ano	Ano	Ano	Ano	Ne
public	Ano	Ano	Ano	Ano	Ano

JAVA

Rozhraní (interface)

Rozhraní definuje metody a konstanty k implementaci. Velmi se tak podobá abstraktní třídě. Ale na rozdíl od tříd nelze deklarovat proměnné, ale zase lze implementovat více než jedno rozhraní. Metody rozhraní jsou implicitně *public*.

```
public interface Info {
    public void kdoJsem();
}

public interface Vlast {
    public void vlastnosti();
}

public class Usecka implements Info {
    int delka;
    Usecka (int delka) { this.delka = delka; }
    public void kdoJsem() { System.out.println("úsečka"); }
}

public class Koule implements Info, Vlast {
    int polomer;
    Koule (int polomer) { this.polomer = polomer; }
    public void kdoJsem() { System.out.println("gula"); }
    public void vlastnosti() { System.out.println("r = " + polomer); }
    public int getR() { return polomer; } //mimo rozhraní
}
```

Poznámka: Jedině třída označená jako *abstract* nemusí implementovat všechny metody rozhraní.

Instance rozhraní

```
Info info = new Koule(3);           //instance rozhraní
Info.kdoJsem();                      //OK
Info.vlastnosti();                  //nelze! Není to metoda rozhraní
int i = ((Koule)info).getR();       //OK
```

Dědičnost v rozhraní a konstanty

```
public interface ObaDva extends Info, Vlast {
    public static final int POCET = 3; //stačilo by int POCET = 3
}
```

Třída implementující toto rozhraní musí mít metody z obou rodičovských rozhraní. Konstanty jsou v rozhraní vždy *public static final*, proto to ani není nutné psát. Odkaz je klasický:

```
int i = ObaDva.POCET; //odkukoliv z programu
int i = POCET;        //pouze ve třídě implementující rozhraní ObaDva
```

Poznámka: Dědičnosti tříd nemají s dědičností rozhraní nic společného. Potomek třídy nemusí implementovat rozhraní rodiče.

JAVA

Operátor instanceof

Operátor *instanceof* slouží ke zjištění zda je objekt instance daného rozhraní či třídy.

```
Usecka u = new Usecka(5);
if (u instanceof Info)      //bude true
    System.out.println("úsečka u je podle pravidel rozhraní Info");
if (u instanceof Usecka)    //samozřejmě taky bude true
    System.out.println("úsečka u je instance třídy Usecka");
```

Polymorfismus

Využití instance předka k volání metod potomků. Tak je možné různě definované metody volat jednotně.

```
abstract class Zivocich {
    public void vypisInfo() {
        System.out.print(getClass().getName() + ". "); vypisDelku();
    }
    public abstract void vypisDelku();
}

class Ptak extends Zivocich {
    int delkaKridel;
    Ptak(int delka) { delkaKridel = delka; }
    public void vypisDelku() {
        System.out.println("Delka kridel: " + delkaKridel);
    }
}

class Slon extends Zivocich {
    int delkaChobotu;
    Slon(int delka) { delkaChobotu = delka; }
    public void vypisDelku() {
        System.out.println("Delka chobotu: " + delkaChobotu);
    }
}

public class PolymAbstr {
    public static void main(String[] args) {
        Zivocich[] z = new Zivocich[2];
        z[0] = new Ptak(10); z[1] = new Slon(12); //nebo třeba i náhodně
        z[0].vypisInfo(); z[1].vypisDelku();      //o tohle šlo
    }
}
```

Samozřejmě, že předek nemusí být vždy abstraktní třída jako v tomto případě. Můžeme mít klidně jako rodiče neabstraktní třídu, překrývat jednu její funkci v potomcích a tu pak volat přes instanci předka.

Polymorfismus za použití rozhraní

Místo kořenové třídy je také možné použít rozhraní.

JAVA

```
interface Zivocich {
    public abstract void vypis();
}

class Ptak implements Zivocich {
    int kridla;
    Ptak(int pocet) { kridla = pocet; }
    public void vypis() { System.out.print("Křídél: " + kridla); }
}

class Slon implements Zivocich {
    int delka;
    Slon(int delka) { this.delka = delka; }
    public void vypis() { System.out.print("Chobot: " + delka + " m"); }
}

public class PolymAbstr {
    public static void main(String[] args) {
        Zivocich z = new Slon(12); z.vypis();
    }
}
```

Vnořené třídy

Vnořená třída (*nested class*, *top-less class*) má neomezená přístupová práva k atributům a metodám vnější třídy (*top-level class*). K atributům a metodám vnitřní třídy se ale nedostane nikdo. Nelze deklarovat ani referenční proměnnou na vnořenou třídu.

```
public interface Info {    //příklady ukazují zapouzdření rozhraní
    void kdoJsem();
}

class Usecka {
    int delka;
    Usecka(int delka) { this.delka = delka; }

    public Info informace() { return new UseckaInfo(); }

    class UseckaInfo implements Info {    //vnitřní třída (inner class)
        public void kdoJsem() { System.out.print("Usecka: " + delka); }
    }
}

public class Test {
    public static void main(String[] args) {
        Usecka u = new Usecka(5);
        Info i = u.informace();    //vrátí referenci na vnitřní třídu
        i.kdoJsem();    //tuto metodu lze zavolat jen přes instanci rozhraní
    }
}
```

Při překladu vznikne soubor *Test.class*, *Usecka.class* a navíc také soubor *Usecka\$UseckaInfo.class*

JAVA

Anonymní vnitřní třída

```
class Usecka {
    int delka;
    Usecka(int delka) { this.delka = delka; }

    public Info informace() {
        return new Info() {           //vrátí instanci rozhraní
            public void kdoJsem() {    //a ta je tvořena vnitřní třídou
                System.out.print("Usecka: " + delka);
            }
        }; //konec příkazu return, musí být středník
    }
}
```

Metoda *main()* i třída *Test* zůstane beze změny. Z vnějšku je tedy vše stejně jako v předešlém příkladě. Při překladu vznikne soubor *Test.class*, *Usecka.class* a navíc také soubor *Usecka\$1.class*

Je možné vytvořit i proměnnou typu rozhraní, např.

```
public Info i = new Info() {
    public void kdoJsem() { //tady následuje definice vnitřní třídy
        System.out.print("Usecka: " + delka);
    }
}; //konec deklarace proměnné, musí být středník
```

Vícenásobná dědičnost pomocí vnitřní třídy

```
class Jmeno {
    public void kdojeto(Object o) {
        System.out.print(o.getClass().getName());
    }
}

class Usecka {
    protected int delka;
    Usecka(int delka) { this.delka = delka; }
}

class Obdelnik extends Usecka implements Info {
    private int sirka;
    Obdelnik(int delka, int sirka) { super(delka); this.sirka = sirka; }
    public void kdoJsem() {
        new VnitрниJmeno().kdoJsem(); } //přesměrování na vnitřní třídu

    class VnitрниJmeno extends Jmeno { //vnitřní třída ma jiného rodiče
        void kdoJsem() {
            kdoJeTo(Obdelnik.this); //využití ukazatele this
            System.out.println(" " + delka + "x" + sirka);
        }
    }
}
```

JAVA

```
public class Test {  
    public static void main(String[] args) {  
        Info i = new Obdelnik(3, 6);  
        i.kdoJsem(); //vypíše 3x6  
    }  
}
```

Tímto postupem jsme vlastně třídě *Obdelnik* dovolili zdědit i metodu třídy *Jmeno*.
Počet vnitřních tříd není omezen, takže pro dědění další třídy se dá použít další vnitřní třída...

Výjimky (exception)

Předkem výjimek je abstraktní třída *Throwable*. Její potomci jsou třídy *Error*, *Exception* a její potomek *RuntimeException* (asynchronní výjimky). Své výjimky odvozujeme od třídy *Exception*.

Předání výjimky výše (throws) a ošetření výjimky (try-catch)

```
import java.util.Scanner;  
import java.io.File;  
  
public class VyjimkaDeklarovana {  
    public static int[] VytvorANactiPole() throws IOException {  
        Scanner sc = new Scanner(new File("data.txt"));  
        int n = sc.nextInt(); //úvodní číslo v souboru - označuje počet  
        int[] pole = new int[n];  
        for (int i = 0; i < n; i++) pole[i] = sc.nextInt(); //prvky  
        return pole;  
    }  
  
    public static void main(String[] args) throws IOException {  
        int[] cisla = VytvorANactiPole();  
        System.out.println(Arrays.toString(cisla));  
    }  
}
```

Metoda *VytvorANactiPole()* pouze předává výjimku výše a zrovna tak metoda *main()*.
V přepracované metodě níže je výjimka ošetřena. Metoda *main()* tedy už nemusí mít v deklaraci *throws*.

```
public static int[] VytvorANactiPole() {  
    int[] pole = null;  
    try {  
        Scanner sc = new Scanner(new File("data.txt"));  
        int n = sc.nextInt(); pole = new int[n];  
        for (int i = 0; i < n; i++) pole[i] = sc.nextInt();  
    }  
    catch (Exception e) {  
        System.out.println("Soubor nenalezen");  
        System.exit(1); //ukončení programu  
    }  
    return pole;  
}
```

JAVA

Předání ošetřené výjimky výše

```
public static int[] VytvorANactiPole() throws IOException { //změna
    int[] pole = null;
    try {
        Scanner sc = new Scanner(new File("data.txt"));
        int n = sc.nextInt(); pole = new int[n];
        for (int i = 0; i < n; i++) pole[i] = sc.nextInt();
        return pole;
    }
    catch (Exception e) {
        System.out.println("Soubor nenalezen");
        throw e; //zde je druhá změna
    }
}
```

V tomto příkladě je výjimka ošetřena a předána výše. Metoda *main()* tedy musí nějak výjimku řešit.

Pokud chceme získat stejný popis výjimky jako vypisuje Java, použijeme metodu *printStackTrace()*

```
catch (Exception e) {
    e.printStackTrace(); //pozor! Výpis jde na konzolu
}
```

Ošetření více výjimek

Počet bloků *catch* není v Javě nijak omezen. Na pořadí bloků *catch* však záleží. Jako první tedy uvádíme specifitější výjimky, protože bloky dále se již nevykonají.

```
public static int[] VytvorANactiPole() throws Exception { //jakákoliv
    int[] pole = null;
    try {
        Scanner sc = new Scanner(new File("data.txt"));
        int n = sc.nextInt(); pole = new int[n];
        for (int i = 0; i < n; i++) pole[i] = sc.nextInt();
        return pole;
    }
    catch (IOException e) {
        System.out.println("Soubor nenalezen");
        throw e;
    }
    catch (InputMismatchException e) {
        e.printStackTrace();
        System.out.println("Chyba v načítaných datech!");
        throw e;
    }
    catch (Exception e) {
        System.out.println("Jiná výjimka:");
        e.printStackTrace();
        throw e;
    }
}
```

JAVA

Vyvolání výjimky

```
throw new InputMismatchException(); //vyvolání výjimky např. pro test

class BankomatException { //vlastní výjimka
    public BankomatException() {
        super("Bankomat mimo provoz"); //text vracený getMessage()
    }
}
```

Pro identifikaci výjimky můžeme použít metodu *getMessage()*

```
catch (BankomatException be) {
    System.out.println(be.getMessage());
}
```

Blok finally

Konstrukci try-catch je možné rozšířit o blok *finally*. Tento blok bude proveden vždy, tj. bez ohledu na to, zda nějaká výjimka nastala nebo ne. Koncový blok *finally* se dokonce provede i když hlídáný blok *try* obsahuje příkaz *return*.

```
File frJmeno = new File("jmeno.txt"); //tohle ještě nevyvolá výjimku
FileReader fr = null;
try {
    fr = new FileReader(frJmeno);
    for (int i = 0; i < frJmeno.length(); i++)
        System.out.print((char)fr.read());
    return frJmeno.length();
}
catch (FileNotFoundException e) {
    ...
}
catch (IOException e) {
    ...
}
finally {
    if (fr != null) fr.close();
}
```

Konstrukce try-finally

Vznikne vynecháním bloků *catch*. Tato konstrukce nemusí mít ale nic společného s výjimkami. Využívá se zde hlavně vlastnosti, že blok *finally* se provede i poté, co někde v hlídáném bloku *try* byl příkaz *return* (který by jinak provádění metody ukončil).

Práce s adresáři a soubory

Pro práci se soubory slouží třída *File*. Pro rozlišení souboru od složky se používají metody *isDirectory()* a *isFile()*, jinak se s nimi pracuje prakticky stejně.

JAVA

```
File.separatorChar    ... statická proměnná, obsahuje oddělovač složek (typu Char) – '\\'
File.separator        ... statická proměnná, obsahuje oddělovač složek (typu String) – "\"
File.pathSeparatorChar ... statická proměnná obsahující oddělovač cest (typu Char) – ';'
File.pathSeparator    ... statická proměnná obsahující oddělovač cest (typu String) – ";"
```

```
String s = System.getProperty("user.dir"); //vrací aktuální adresář
```

```
File soubAbs = new File(s, "a.txt"); //konstruktor s dvěma parametry
File soubRel = new File("TMP" + File.separator + "a.txt"); //s jedním
```

```
String s = soubAbs.getName(); //vrací jméno souboru – "a.txt"
String s = soubAbs.getAbsolutePath(); //vrací celou cestu + jméno
```

```
if (soubAbs.exists() == false) soubAbs.createNewFile();
```

```
File adr = new File("TMP"); //složka
if (adr.exists() == false) adr.mkdir(); //vytvoření v akt. adresáři
```

Pro vytvoření více vnořených adresářů se používá metoda *mkdirs()*

```
long i = soubRel.length(); //délka souboru
System.out.print(new Date(soubRel.lastModified())); //čas posl. změny
```

```
File soub = new File("a.txt");
File jiny = new File("c.txt");
soub.renameTo(jiny); //přejmenování, ale pouze na disku
adr.renameTo(new File("TMP-OLD")); //přejmenování, ale pouze na disku
```

Po metodě přejmenování souboru se jméno změní na disku, ne v instanci. Instance tak už bude k ničemu.

```
soub.delete(); //smazání souboru, ale neproběhne – viz výše
jiny.delete(); //smazání souboru c.txt
```

```
String[] jmena = adr.list(); //vrací názvy souborů ve složce
File[] soubory = adr.listFiles(); //vrací soubory typu File
```

Filtrovaný výpis adresáře

Provádí se pomocí implementace rozhraní *java.io.FileNameFilter*, který má jedinou metodu *accept()*, a pomocí přetížených metod *list()* a *listFiles()*, které jako parametr mohou mít referenci na objekt typu *FileNameFilter*

```
class FiltrPripony implements FileNameFilter {
    String maska;
    FiltrPripony(String maska) { this.maska = maska; }
    public boolean accept(File dir, String name) {
        if (name.lastIndexOf(maska) > 0) //hledání v řetězci odzadu
            return true;
        else
            return false;
    }
}
```

JAVA

```
class FiltrVelikosti implements FilenameFilter {
    long velikost;
    FiltrVelikosti(long vel) { velikost = vel; }

    public boolean accept(File dir, String name) {
        File f = new File(dir, name);
        if (f.length() > velikost)
            return true;
        else
            return false;
    }
}

public class Test {
    public static void main(String[] args) {
        String jmenoAktDir = System.getProperty("user.dir");
        File aktDir = new File(jmenoAktDir);

        String[] jmena;
        FiltrPripony filtrPr = new FiltrPripony(".java");
        jmena = aktDir.list(filtrPr);
        for (int i = 0; i < jmena.length; i++)
            System.out.println(jmena[i]);

        File[] soubory;
        FiltrVelikosti filtrVel = new FiltrVelikosti(1000);
        soubory = aktDir.listFiles(filtrVel);
        for (int i = 0; i < soubory.length; i++) {
            String s = soubory[i].getName() + "\t" + soubory[i].length();
            System.out.println(s);
        }
    }
}
```

Proudy (stream)

Proudy jsou zobecněním čtení ze vstupů a zápisem do výstupů. Java má pro tento účel v balíku *java.io* připraveny tyto čtyři základní abstraktní třídy:

1. Znakově orientovaný proud **Reader** (základní jednotka je 16-bitová pro znak Unicode).

Základní metody:

```
int read();
int read(char[] pole);
int read(char[] pole, int index, int pocet);
```

2. Znakově orientovaný proud **Writer** (základní jednotka je 16-bitová pro znak Unicode).

Základní metody:

```
int write();
int write(char[] pole);
int write(char[] pole, int index, int pocet);
void write(String retez);
void write(String retez, int index, int pocet);
```

JAVA

3. Bajtově orientovaný proud ***InputStream*** (základní jednotka je 8-bitová).

Základní metody:

```
int read();  
int read(byte[] pole);  
int read(byte[] pole, int index, int pocet);
```

4. Bajtově orientovaný proud ***OutputStream*** (základní jednotka je 8-bitová).

Základní metody:

```
int write();  
int write(byte[] pole);  
int write(byte[] pole, int index, int pocet);
```

Všechny metody *read()* vracejí hodnotu -1, když bylo dosaženo konce proudu.

Všechny třídy mají také metodu *close()* k uzavření proudu a mohou vyvolat výjimku typu *IOException*.

Z těchto čtyř tříd jsou odvozeny třídy, které fyzicky pracují se zařízeními. Viz následující tabulka:

	znakový přesun	bajtový přesun
paměť	<i>CharArrayReader, CharArrayWriter</i> <i>StringReader, StringWriter</i>	<i>ByteArrayInputStream, ByteArrayOutputStream</i> <i>StringBufferInputStream</i>
Soubor	<i>FileReader, FileWriter</i>	<i>FileInputStream, FileOutputStream</i>
Roura	<i>PipedReader, PipedWriter</i>	<i>PipedInputStream, PipedOutputStream</i>

Kromě nich existují ještě odvozené třídy, sloužící k nějaké úpravě dat. Například pro konverze (*InputStreamReader, OutputStreamReader*), filtrování (*FilterReader, FilterWriter, FilterInputStream, FilterOutputStream*), spojování (*SequenceInputStream*) a další.

```
public static void main(String[] args) throws IOException {  
    File frJmeno = new File("a.txt");  
    FileReader fr = new FileReader(frJmeno); //ukázka iniciace přes File  
  
    FileWriter fw = new FileWriter("b.txt");  
  
    int c;  
    while ((c = fr.read()) != -1) fw.write(c);  
  
    fr.close(); fw.close();  
}
```

Třída *FileReader* sice pracuje s 16-bitovými znaky, ale stejně bude číst soubor v Unicode po bajtech. Koná tak podle svého přednastavení, které se dá zjistit metodou *getEncoding()*. Vrací „Cp1250“.

```
public static void main(String[] args) throws IOException {  
    File frJmeno = new File("a.txt");  
    if (frJmeno.exists()) {  
        FileInputStream fr = new FileInputStream(frJmeno);  
        FileOutputStream fw = new FileOutputStream("b.txt");  
        int c;  
        for (long i = 0; i < frJmeno.length(); i++) {  
            c = fr.read(); fw.write(c);  
        }  
        fr.close(); fw.close();  
    }  
}
```


JAVA

Třída *FileReader* má navíc metody:

<code>skip(long pocet)</code>	... přeskočí při čtení daný počet znaků
<code>reset()</code>	... skok na začátek nebo na značku
<code>mark(long platnost)</code>	... vytvoření značky, platnost udává kolik znaků můžeme od označené pozice načíst, aniž by značka ztratila platnost
<code>boolean markSupported()</code>	... testuje, zda jsou metody <i>reset()</i> a <i>mark()</i> vůbec podporovány

Třída *FileWriter* má navíc metody:

<code>flush()</code>	... okamžité zapsání bufferovaných dat na disk
<code>FileWriter(String jmeno, boolean append)</code>	... další konstruktor, kde <i>append</i> udává, zda bude přepisovat na konec nebo ne

Bufferování a čtení po řádcích

K tomuto se používají třídy začínající slovem *Buffered* (místo *File*). Jedná se také o potomky základní čtyřky, ale slouží jen k úpravě dat. Proto jsou jen jakýmsi obalem fyzických proudů.

```
FileReader fr = new FileReader("a.txt");
BufferedReader in = new BufferedReader(fr);
FileWriter fw = new FileWriter("b.txt");
BufferedWriter out = new BufferedWriter(fw);
String radka;

while((radka = in.readLine()) != null) {
    out.write(radka);
    out.newLine(); //metoda readLine() vynechává znak konce řádků
}
fr.close();
out.close(); //kvůli zásobníku je potřeba volat toto místo fw.close()
```

Vrácení přečteného znaku

Když přečteme něco navíc, metoda *unread()* obalových tříd *PushbackReader* a *PushbackInputStream* to zase vrátí zpět do vstupního proudu:

```
FileReader fr = new FileReader("a.txt");
PushBackReader in = new PushBackReader(fr);
int c;
c = in.read(); //načten znak
in.unread(c); //zde je vrácen zpět do čtení
c = in.read(); //a zde je znovu načten stejný znak
```

Formátovaný výstup textu do souboru

Použijeme obalové třídy *PrintWriter* nebo *PrintStream* a jejich metody *println()*, *print()* nebo *format()*. Mají stejné vlastnosti jako jejich jmenovkyně ze třídy *System.out*

JAVA

Řádkové bufferování

Třída *PrintWriter* má navíc tento konstruktor:

```
PrintWriter(Writer out, boolean autoFlush);
```

Pokud v parametru *autoFlush* zadáme hodnotu *false*, bude výpis do souboru pozdržen až do výpisu znaku konce řádku. Pokud nezvolíme tento konstruktor nebo zadáme *autoFlush* jako *true*, bude se zapisovat do výstupu každý znak okamžitě.

Neformátovaný výstup základních datových typů

Bajtové proudy lze obalit třídami *DataInputStream* a *DataOutputStream*. Tyto třídy mají metody určené pro zápis základních datových typů – *readBoolean()*, *writeBoolean()*, *readDouble()*, *writeDouble()*, *readInt()*, *writeInt()*, atd. – a také pro čtení a zápis UTF-8 sloužící k uložení znaků Unicode:

```
void writeUTF(String s);  
String readUTF();
```

Zápis objektu do souboru - serializace

Bajtové proudy lze obalit také třídami *ObjectInputStream* a *ObjectOutputStream*. Tyto třídy mají mimo jiné i metody *readObject()* a *writeObject()*. Tím umožňují ukládat na disk instance tříd, které implementují rozhraní *java.io.Serializable*

```
import java.io.*;  
import java.util.*; //kvůli třídě Date, kterou jsem si vypůjčil  
  
public class UlozeniObjektu {  
    public static void main(String[] args) throws Exception {  
  
        ObjectOutputStream fw =  
            new ObjectOutputStream(new FileOutputStream("d.bin"));  
        Date d = new Date();  
        System.out.println("Před uložením: " + d);  
        fw.writeObject(d); //uložení instance datumu na disk  
        fw.close();  
  
        d = null;  
        System.out.println("Po nulování: " + d); //vypíše null  
  
        ObjectInputStream fr =  
            new ObjectInputStream(new FileInputStream("d.bin"));  
        Date e = (Date)fr.readObject();  
        System.out.println("Načteno: " + e);  
        fr.close();  
    }  
}
```

Místo třídy *Date* jsem samozřejmě mohl použít svou třídu implementující rozhraní *Serializable*.

JAVA

Seskupování obalů (vlastností)

Protože parametr konstruktorů obalových tříd je některého typu ze základní čtyřky (např. u třídy *PrintWriter* nebo *BufferedWriter* je samozřejmě typu *Writer*), je možné psát:

```
PrintWriter pBuf = new PrintWriter(  
    new BufferedWriter(  
        new FileWriter("soub.txt")));
```

Vstup a výstup do paměti

Podobná pravidla jako pro soubory platí i pro paměťové proudy.

```
public static void main(String[] args) throws IOException {  
    StringWriter sProud = new StringWriter();  
    PrintWriter pw = new PrintWriter(sProud);  
  
    for (int i = 1; i <= 3; i++) {  
        pw.print("Ahoj " + (4 - i) + "\n");  
        System.out.println(sProud); //bude stále připisovat Ahoj s čísly  
    }  
    pw.close();  
}
```

Přímý přístup do souboru

Alternativou k proudům je třída ***RandomAccessFile***. Na rozdíl od proudu umožňuje například současné čtení i zápis do souboru. Má tyto konstruktory:

```
RandomAccessFile(File soubor, String jakOtevrit);  
RandomAccessFile(String jmenoSouboru, String jakOtevrit);
```

První parametr je jasný, druhý může být buď hodnota "r" nebo "rw", tedy informace zda se má soubor otevřít jen pro čtení nebo i pro zápis. Použití "rw" soubor nemaže ani nenastavuje na nulovou délku. Dále má třída tyto metody:

<code>length()</code>	... délka souboru
<code>skipBytes(int n)</code>	... přeskočí následujících n bajtů
<code>getFilePointer()</code>	... vrací aktuální pozici v souboru
<code>seek(long pozice)</code>	... nastaví aktuální pozici v souboru
<code>setLength(long velikost)</code>	... nastaví novou velikost souboru (zvětší jej nebo ořízne)
<code>close()</code>	... uzavření souboru
<code>writeBytes(String s)</code>	... obdoba zápisu <code>writeLine()</code> , zapisuje řetězec
<code>writeChars(String s)</code>	... zapisuje řetězec 16-bitově, tedy zdvojnásobuje délku

Třída má deklarovány i metody `read()`, `write()`, `readBoolean()`, `writeBoolean()`, `readDouble()`, `writeDouble()`, `readInt()`, `writeInt()`, atd. Dokonce i `writeUTF()`, `readUTF()` a `readLine()`.

```
RandomAccessFile soub = new RandomAccessFile("jmeno.bin", "rw");  
soub.seek(0L); //přesun na začátek souboru
```

JAVA

Parametry příkazové řádky

Jsou v metodě *main()* předávány v parametru pole řetězců *args*. Pokud chceme mít v parametrech řetězec s mezerou, uzavřeme jej do uvozovek. Tyto uvozovky jsou při čtení parametru vynechány.

Standardní systémový výstup

Privátní třída *System* má všechny metody a proměnné statické. Pro konzolu má tyto proměnné:

<code>InputStream in</code>	... pro vstup z konzole
<code>PrintStream out</code>	... pro výstup na konzolu
<code>PrintStream err</code>	... pro chybové hlášení na konzolu

Přesměrovat výstup na konzolu nebo chybové hlášení můžeme pomocí metod *SetIn()*, *SetOut()* a *SetErr()*. Tyto metody potřebují jako parametr jiný otevřený proud.

```
FileOutputStream fw = new FileOutputStream("chyby.log");
PrintStream log = new PrintStream(fw);
System.setErr(log);
```

Systémové vlastnosti

Jejich přehled získáme příkazem

```
System.getProperties().list(System.out);
```

Jednotlivou vlastnost pak získáme metodou *getProperty()*

<code>file.encoding</code>	<code>Cp1250</code>	... kódování textových souborů
<code>file.separator</code>	<code>\</code>	... oddělovač adresářů v cestě
<code>line.separator</code>	<code>\r\n</code>	... oddělovač řádků
<code>path.separator</code>	<code>;</code>	... oddělovač cest
<code>os.arch</code>	<code>x86</code>	... typ procesoru
<code>os.name</code>	<code>Windows 7</code>	... název operačního systému
<code>os.version</code>	<code>6.1</code>	... verze operačního systému
<code>java.class.path</code>	<code>.</code>	... adresáře, kde jsou hledány soubory .class
<code>java.class.version</code>	<code>50.0</code>	... verze souborů .class
<code>java.version</code>	<code>1.6.0_21</code>	... verze Javy
<code>java.home</code>	<code>C:\Program Files (x86)\Java\jre6</code>	
<code>java.vendor</code>	<code>Sun Microsystems Inc.</code>	
<code>java.vendor.url</code>	<code>http://java.sun.com/</code>	
<code>user.name</code>	<code>jerabekp</code>	
<code>user.home</code>	<code>C:\Users\jerabekp</code>	
<code>user.dir</code>		... aktuální adresář

```
user.country=CZ
user.language=cs
user.timezone=
```

JAVA

```
awt.toolkit=sun.awt.windows.WToolkit
file.encoding.pkg=sun.io
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.awt.printerjob=sun.awt.windows.WPrinterJob
java.endorsed.dirs=C:\Program Files (x86)\Java\jre6\lib\...
java.ext.dirs=C:\Program Files (x86)\Java\jre6\lib\...
java.io.tmpdir=C:\Users\jerabekp\AppData\Local\Temp\
java.library.path=C:\Windows\SysWOW64;. ;C:\Windows\Sun\...
java.runtime.name=Java(TM) SE Runtime Environment
java.runtime.version=1.6.0_21-b07
java.specification.name=Java Platform API Specification
java.specification.vendor=Sun Microsystems Inc.
java.specification.version=1.6
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport...
java.vm.info=mixed mode, sharing
java.vm.name=Java HotSpot(TM) Client VM
java.vm.specification.name=Java Virtual Machine Specification
java.vm.specification.vendor=Sun Microsystems Inc.
java.vm.specification.version=1.0
java.vm.vendor=Sun Microsystems Inc.
java.vm.version=17.0-b17
sun.arch.data.model=32
sun.boot.class.path=C:\Program Files (x86)\Java\jre6\lib\...
sun.boot.library.path=C:\Program Files (x86)\Java\jre6\bin
sun.cpu.endian=little
sun.cpu.isalist=pentium_pro+mmx pentium_pro pentium+m...
sun.desktop=windows
sun.io.unicode.encoding=UnicodeLittle
sun.java.launcher=SUN_STANDARD
sun.jnu.encoding=Cp1250
sun.management.compiler=HotSpot Client Compiler
sun.os.patch.level=
user.variant=
```

Další metody třídy System

```
long z = System.currentTimeMillis(); //vrací počet ms od 1.1.1970
System.exit(-1); //násilné ukončení programu, -1 je návratová hodnota
System.gc(); //volání garbage collectoru
```

Informace o paměti

Ve třídě *Runtime* v balíku *java.lang* jsou metody *totalMemory()* a *freeMemory()*

```
Runtime r = Runtime.getRuntime(); //inicializace referenční proměnné
long i = r.freeMemory();
```

Alokovat paměť můžeme i při spouštění Javy pomocí přepínačů *Xms* (kolik alokovat na počátku programu) a *Xmx* (maximální možná alokace paměti).

```
D:\java>java -Xms500M -Xmx500M Program
```

JAVA

Vlákna (threads)

Každé vlákno je instancí třídy *java.lang.Thread* nebo jejího potomka. Klíčová je metoda *run()*, která popisuje, co vlákno dělá. Metoda *run()* se nespouští přímo, ale pomocí metody *start()*

```
public class Vlakno1 extends Thread {
    public Vlakno1(String jmeno) {
        super(jmeno);    //konstruktor threadu, chce jméno vlákna
    }

    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(i + ". " + getName());    //vrací jméno vlákna
            try {
                Thread.sleep(1000);    //čekání procesu v ms
            }
            catch (InterruptedException e) {
                System.out.println("Probudili jste mne predcasne.");
            }
        }
    }

    public static void main(String[] args) {
        Vlakno1 vl = new Vlakno1("ahoj");
        vl.start();

        new Vlakno1("nazdar").start();
    }
}
```

Střídání vláken

Vlákna se mohou také v běhu střídat, tj. neběžet současně, ale předávat si řízení. To se děje pomocí statické metody *yield()*, která říká, že se vlákno vzdává řízení.

Druhé vlákno si však řízení nemusí převzít, protože bude uspáno pomocí statické metody *sleep()*.

V takovém případě pokračuje první vlákno normálně dál za příkazem *Thread.yield()* ;

Stavy vláken

1. **Nové vlákno** – vlákno vytvořeno, ale neodstartováno
2. **Běhuschopné (runnable)** – vlákno odstartováno; těchto vláken může být více, ale jen jedno je skutečně běžící, ostatní čekají na předání řízení
3. **Neběhuschopné** – vlákno, které bylo uspáno (sleep), čeká na *wait()* nebo čeká na I/O
4. **Mrtvé vlákno** – vlákno, jehož metoda *run()* skončila

Každé vlákno má nastavenou prioritu. Budou-li běhuschopné dvě vlákna, bude řízení předáno vždy tomu s vyšší prioritou. K nastavení priorit slouží metody *getPriority()* a *setPriority()* a konstanty *MIN_PRIORITY*, *NORM_PRIORITY* (přidělována automaticky) a *MAX_PRIORITY*, mající hodnoty 1, 5 a 10.

Pokud se do běhuschopného stavu dostane vlákno s vyšší prioritou než má právě běžící, je běžící vlákno přinuceno předat řízení – jedná se o tzv. preemptivní plánování.

JAVA

Sdílení času (time-slicing)

Vlákna se stejnou prioritou se pravidelně střídají po operačním systémem pevně přiděleném čase. Není tedy nutný žádný mechanismus předávání pomocí metod *yield()* či *sleep()*.

Rozhraní Runnable

Rozhraní *Runnable* nutí programátora deklarovat pouze metodu *run()*. Aby se třída chovala jako vlákno, je potřeba ale naprogramovat i *start()*. Platí také, že vlákno musí být vždy instance třídy *Thread*. Proto se v metodě *start()* naší třídy vytváří instance třídy *Thread*, které v konstruktoru předáme referenci na naší třídu, např. pomocí *this*, a tato se odstartuje.

```
public class Vlakno2 implements Runnable {
    private Thread vl = null;

    public void start() {
        vl = new Thread(this);    //vytvoření vlákna s předáním naší třídy
        vl.start();                //odstartování tohoto vlákna
    }

    public void run() {
        while (interrupted() == false) {
            ...                    //nějaká činnost vlákna
        }
    }
}
```

Další metody třídy Thread

<code>Thread.currentThread()</code>	...vrací instanci právě běžícího vlákna
<code>Thread()</code>	...konstruktor bez parametrů, nemusíme jej tedy v naší třídě řešit
<code>boolean isAlive()</code>	...dotaz, zda vlákno ještě probíhá, tj. že ještě není ukončeno
<code>join()</code>	...čekání na ukončení vlákna
<code>join(long ms)</code>	...čekání na ukončení vlákna, ale maximálně <i>ms</i> milisekund (timeout)
<code>join(long ms, long ns)</code>	...čekání, ale maximálně zadaný čas v ms a nanosekundách
<code>interrupt()</code>	...příkaz k přerušení vlákna
<code>boolean interrupted()</code>	...dotaz v metodě <i>run()</i> , zda bylo vlákno přerušeno

Pozor! I metoda *main()* běží jako vlákno.

```
public static void main(String[] args) {
    Vlakno1 vl = new Vlakno1("moje");
    vl.start();
    vl.join();    //čekání na ukončení vlákna vl typu Vlakno1
    System.out.println("Ukončení main");
}
```

Metoda *interrupt()* vlákno neukončí, to umí metody *stop()* a *suspend()*, které se ale nemají používat (jsou označeny jako *deprecated*). Metoda *interrupt()* navíc probouzí uspané vlákno a vyvolává v něm výjimku *InterruptedException*. Proto je potřeba ji ve vláknech ošetřovat.

JAVA

Kritické sekce – synchronizované metody a bloky

Klíčovým slovem *synchronized* se označují metody, uprostřed kterých nemůže dojít k předání řízení.

```
synchronized public void nastav(int x, int y) {  
    xx = x;    //po tomto příkazu nechceme možnost, že se druhý neprovede  
    yy = y;  
}
```

Klíčovým slovem *synchronized* se také označují bloky kódu, které chceme svázat s určitým objektem. Například když tím objektem bude soubor, nebude se na jeho instanci přistupovat zároveň.

```
public class BlokSynchr {  
    RandomAccessFile file;  
    BlokSynchr(RandomAccessFile f) { file = f; }  
  
    public void presun(long kam) throws IOException {  
        synchronized (file) {  
            file.seek(kam);           //nebude možné zároveň číst i skákat  
        }  
    }  
    public int ctiInt() throws IOException {  
        synchronized (file) {  
            return file.readInt();    //nebude možné zároveň číst i skákat  
        }  
    }  
}
```

Synchronizace vláken

Slouží k tomu aby jeden proces čekal na výsledky druhého. K tomu máme metody základní třídy *Object*:

<code>wait()</code>	...zastavení vlákna až do jeho probuzení pomocí <code>notify()</code> či <code>notifyAll()</code>
<code>wait(long ms)</code>	...zastavení vlákna, ale maximálně <i>ms</i> milisekund
<code>wait(long ms, long ns)</code>	...zastavení vlákna, ale maximálně zadaný čas v ms a nanosekundách

<code>notifyAll()</code>	...probudí všechna vlákna v objektu, která byla zastavena pomocí <code>wait()</code>
<code>notify()</code>	...probudí jen jedno vlákno

Démon (daemon)

Program nemůže skončit dříve než jsou ukončena všechna vlákna. Výjimkou jsou však vlákna označená jako démon, na která při svém ukončení program nebere ohled, prostě jsou utnuta.

```
public static void main(String[] args) {  
    Vlakno1 vl = new Vlakno1("přízrak");  
    vl.setDaemon(true);    //označení, že se na něj na konci nebude čekat  
    vl.start();  
    if (vl.isDaemon()) System.out.println("Smůla, nedoběhne");  
}
```


JAVA

Grafické prostředí

V Javě jsou dvě – starší AWT a novější JFC Swing.

Základní AWT komponenty jsou odvozeny od třídy *java.awt.Component*.

Základní okno pak musí být instance třídy *Frame*

```
import java.awt.*;
import java.awt.event.*;

public class Tlacitkol extends Frame {
    Button ahojBT;
    Label lab;
    Tlacitkol() {
        super.setTitle(getClass().getName()); //konstruktor naší třídy
        this.setLayout(new FlowLayout()); //nastavení titulku okna
        ahojBT = new Button("Ahoj"); //tlačítko
        this.add(ahojBT);
        lab = new Label("Nazdar"); //label
        this.add(lab);
        this.setSize(170, 65); //nastavení šířky a výšky

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(1);
            }
        }); //možnost zavření okna
    }
    public static void main(String[] args) {
        Tlacitkol t = new Tlacitkol();
        t.setVisible(true); //nastavení viditelnosti
    }
}
```

Události

Zdroj události (zde například tlačítko) si pomoci metody *addXYZListener()* zaregistruje komu bude posílat zprávy. Zaregistrovaný objekt pak musí implementovat rozhraní *XYZListener* s metodami, které odpovídají možným zprávám.

```
public class Tlacitko2 extends Tlacitkol implements ActionListener {
    Tlacitko2() {
        ahojBT.addActionListener(this); //pro zprávy registruje sebe
        this.setSize(170, 65);
    }
    public void actionPerformed(ActionEvent e) {
        //jediná metoda rozhraní
        lab.setText("Ahoj");
    }
    public static void main(String[] args) {
        new Tlacitko2().setVisible(true);
    }
}
```

JAVA

Pro rozlišení, který objekt poslal zprávu má třída *ActionEvent* několik metod:

```
String s = e.getActionCommand()    ... identifikátor uvedený na komponentě ("Ahoj")
Object o = e.getSource()           ... instance původce zprávy (objekt ahojBT)
```

Metoda *getSource()* pochází od třídy *java.util.EventObject*, je tedy nutné importovat *java.util.**

Použití vnitřních tříd pro zprávy

Pro rozlišení zasílání zpráv je dobré použít vnitřní třídy:

```
public class Tlacitko3 extends Tlacitko2 {
    Button nazdarBT;

    class Udalost implements ActionListener { //vnitřní třída
        String vypis;

        public Udalost(String vypis) { //konstruktor vnitřní třídy
            this.vypis = vypis;
        }

        public void actionPerformed(ActionEvent e) {
            //implementace rozhraní
            lab.setText(vypis);
        }
    }

    Udalost ahojUD, nazdarUD;

    public Tlacitko3() {
        nazdarBT = new Button("Nazdar"); //přidáno druhé tlačítko
        this.add(nazdarBT);

        ahojUD = new Udalost("AHOJ");
        ahojBT.addActionListener(ahojUD);
        //přidána také instance vnitřní třídy
        nazdarUD = new Udalost("NAZDAR");
        nazdarBT.addActionListener(nazdarUD);
    }

    public static void main(String[] args) {
        new Tlacitko6().setVisible(true);
    }
}
```

Doporučovaný způsob je ale pro každý zdroj zpráv udělat vlastní vnitřní třídu:

```
public class Nejprehlednejsi extends Frame {
    Label lab;
    public Nejprehlednejsi() {
        super.setTitle(getClass().getName()); //volání konstruktoru Frame
        this.setLayout(new FlowLayout());
    }
}
```

JAVA

```
Button ahojBT = new Button("Ahoj");
this.add(ahojBT);
ahojBT.addActionListener(new AhojBTAL());

lab = new Label("Nazdar");
this.add(lab);

Button nazdarBT = new Button("Nazdar");
this.add(nazdarBT);
nazdarBT.addActionListener(new NazdarBTAL());

this.setSize(170, 65); //možno použít i this.pack()
this.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(1);
    }
});
}

class AhojBTAL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        lab.setText("AHOJ");
    }
}

class NazdarBTAL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        lab.setText("NAZDAR");
    }
}

public static void main(String[] args) {
    new Nejprehlednejsi().setVisible(true);
}
}
```

Dá se také navíc využít anonymních vnitřních tříd:

```
Button ahojBT = new Button("Ahoj");
this.add(ahojBT);
ahojBT.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //lab.setText("AHOJ");
        ahojBT_actionPerformed(e); //a nebo tento RAD styl
    }
}); //a nemusíme definovat vnitřní třídu AhojBTAL

void ahojBT_actionPerformed(ActionEvent e) {
    lab.setText("AHOJ");
}
```

Pokud je obsluha zprávy stejná pro více zdrojů (například uložení souboru stiskem ikony nebo výběr položky Uložit z menu), je dobré použít vnější třídu.

JAVA

Odregistrovat posluchače může zdroj zprávy pomocí metody *removeXYZListener()*. Seznam posluchačů je možné získat pomocí metody *getListeners()*

```
ActionListener[] a = (ActionListener[])
    (this.getListeners(ActionListener.class));
for (int i = 0; i < a.length; i++)
    this.removeActionListener(a[i]);
```

V příkladě je použit speciální statický atribut *class* (zde *ActionListener.class*). Skutečným parametrem funkce *getListeners()* je instance třídy *Class*

Vyvolání události

Událost můžeme vyvolat i mimořádně, pomocí metody *processEvent()*, která je ve třídě *java.awt.Component* definována takto:

```
protected void processEvent(AWTEvent e)
```

Slovo *protected* zde znamená, že ji lze volat pouze ze zděděných tříd, *AWTEvent* je abstraktní třída, rodič všech událostí. Jako parametr tedy můžeme použít *ActionEvent* s tímto konstruktorem:

```
ActionEvent(Object source, int id, String command)
```

Parametr *source* udává objekt, *id* identifikační číslo (zde nepoužíváno, vkládáme tedy vždy 1) a *command* je řetězec identifikující komponentu.

```
class UdalostBT extends Button {
    UdalostBT(String jmeno) {
        super(jmeno);
    }

    public void processEvent(AWTEvent e) {
        //překrytí metody => zpřístupnění
        super.processEvent(e);
    }

    void provedUdalost() {
        //vhodnější než překrytí je vytvoření nové metody
        String s = this.getLabel(); //pro tlačítka vrací text na něm
        super.processEvent(new ActionEvent(this, 1, s));
    }
}
```

```
... //a pak někde v kódu
UdalostBT ahojBT = new UdalostBT("Ahoj");
this.add(ahojBT);
ahojBT.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        lab.setText("AHOJ");
    }
});
ahojBT.provedUdalost(); //a je to
```

JAVA

Adaptéry

Pokud rozhraní *XYZListener* má více metod, musíme je implementovat všechny. Abychom to nedělali, jsou připraveny třídy *XYZAdapter*, které mají všechny metody implementovány prázdné a stačí je tedy pouze zdědit. Např. pro sedmimetodové rozhraní *WindowListener* tak existuje třída *WindowAdapter*

```
this.addWindowListener(new WindowAdapter() {  
    public void windowClosing(WindowEvent e) {  
        System.exit(1);  
    }  
}); //tohle už tu bylo použito výše, definována je pouze 1 metoda
```

Jiný příklad s použitím vnitřní třídy:

```
public class Okno1 extends Frame {  
    Label lab;  
  
    Okno1() {  
        super.setTitle(getClass().getName());  
        lab = new Label("Udalosti okna");  
        this.add(lab);  
        this.setSize(130, 65);  
    }  
  
    public static void main(String[] args) {  
        new Okno1().setVisible(true);  
    }  
}  
  
public class Okno2 extends Okno1 {  
  
    class Zavirac extends WindowAdapter { //vnitřní třída  
        public void windowClosing(WindowEvent e) {  
            lab.setText("Pockej chvíli");  
            for (int i = 0; i < 1000000; i++)  
                ;  
            System.exit(1);  
        }  
    }  
  
    Okno2() {  
        this.addWindowListener(new Zavirac()); //registrace anonyma  
    }  
  
    public static void main(String[] args) {  
        new Okno2().setVisible(true);  
    }  
}
```

Poznámka ke zdrojákům

Všechny příklady je možné najít na <http://www.kiv.zcu.cz/~herout/java/ujj2/>

JAVA

Barvy

Barvy jsou instancí třídy *java.awt.Color*, např.

```
Color.blue           //pozor! Není to konstanta Color.BLUE
Color.lightgray
Color.magenta        //purpurová
Color.cyan           //azurová
Color.orange         //oranžová
Color.pink           //růžová
Color mojebarva = new Color(cervena, zelena, modra);

Button ahoyBt = new Button("Ahoj");
ahoyBt.setForeground(Color.yellow);           //nastavení písma
this.add(ahoyBt);
this.setBackground(new Color(120, 255, 0));   //nastavení pozadí
```

Fonty

Fonty jsou instancí třídy *java.awt.Font*. Pro každou komponentu není potřeba nová instance fontu, stačí jedna pro jeden druh písma. Pro písma jsou připraveny tři konstanty:

Font.PLAIN, Font.BOLD, Font.ITALIC ... druhy písma (tzv. řezy, font face)

Kvůli přenositelnosti mezi platformami jsou definovány v Javě symbolické fonty:

<i>Symbolický (logický) font Javy</i>	<i>Font v MS Windows</i>	<i>Font v Linux</i>
Serif	Times New Roman	Times Roman
SansSerif	Arial	Helvetica
MonoSpaced	Courier New	Courier
Dialog	Arial	Helvetica
DialogInput	Courier New	Courier

Pokud žádný nenastavíme, použije se font Dialog v základním řezu (plain) ve velikosti 12 bodů. Pro práci s písmy má třída *Component* metody *setFont()* a *getFont()* a třída *Font* metody *getName()*, která vrací logické jméno fontu, *getFamily()*, která vrací fyzické jméno fontu, *isBold()*, *isItalic()* a *isPlain()* pro zjištění typu písma a *deriveFont()* pro odvození fontu.

```
Label ahoyLb = new Label("Ahoj");
this.add(ahoyLb);
Font f = new Font("SansSerif", Font.BOLD + Font.ITALIC, 16);
ahoyLb.setFont(f);

Font f1 = new Font("SansSerif", Font.PLAIN, 40);
Font f2 = f1.deriveFont(Font.BOLD, (float) 40.1); //odvození fontu
Font f3 = ahoyLb.getFont();
```

Znepřístupnění komponenty

K tomuto slouží metody komponenty *setEnabled(boolean b)* a *boolean isEnabled()*

JAVA

Viditelnost komponenty

K tomuto slouží metody komponenty `setVisible(boolean b)` a `boolean isVisible()`

Poloha a velikost komponenty

Poloha je udávána v pixelech a to od horního levého rohu okna programu (včetně titulového pruhu).

Levý horní roh komponenty získáme pomocí metody `getLocation()`, která vrací instanci třídy *Point* s proměnnými *x* a *y*.

Velikost komponenty získáme metodou `getSize()`, která vrací instanci třídy *Dimension* s proměnnými *height* a *width*.

Alternativou je metoda `getBounds()`, která vrací instanci třídy *java.awt.Rectangle* s proměnnými *x*, *y*, *height* a *width*.

Je také možné použít metody komponenty `getX()`, `getY()`, `getHeight()` a `getWidth()`, které vracejí hodnoty typu *int*.

Pro nastavení polohy lze použít některou z těchto metod:

```
void setBounds(int x, int y, int width, int height)
void setBounds(Rectangle r)
void setSize(Dimension d)
void setSize(int width, int height)
void setLocation(int x, int y)
```

Kurzory

V třídě *Cursor* je jako parametr konstruktoru možno použít některou z těchto 14 konstant typu *int*:

```
Cursor.DEFAULT_CURSOR, Cursor.CROSSHAIR_CURSOR, Cursor.TEXT_CURSOR,
Cursor.WAIT_CURSOR, Cursor.SW_RESIZE_CURSOR, Cursor.SE_RESIZE_CURSOR,
Cursor.NW_RESIZE_CURSOR, Cursor.NE_RESIZE_CURSOR,
Cursor.N_RESIZE_CURSOR, Cursor.S_RESIZE_CURSOR,
Cursor.W_RESIZE_CURSOR, Cursor.E_RESIZE_CURSOR, Cursor.HAND_CURSOR,
Cursor.MOVE_CURSOR
```

Samožřejmostí jsou metody komponenty `getCursor()` a `setCursor()`

```
Cursor c = new Cursor(Cursor.WAIT_CURSOR);
ahojBt.setCursor(c);
String s = c.getName() + ", typ = " + Integer.toString(c.getType());
```

Výběr speciálnějších metod některých komponent

```
void setAlignment(int alignment)    ... u návěští (labelu) nastavuje zarovnání
Label.CENTER, Label.LEFT, Label.RIGHT ... konstanty zarovnání pro návěští (labely)
```

```
int getSelectedIndex()             ... metoda třídy Choice (Combobox – DropDownList)
String getItem(int index)          ... metoda třídy Choice, vrací text položky na zadané pozici
```

JAVA

Radiobuttony - CheckboxGroup

`setState(boolean), boolean getState()` ... nastavení/zjištění zaškrtnutí checkboxu

Checkboxy lze sdružit do skupiny, kde vybraný může být pouze jeden (ten poslední zaškrtnutý). Proto má třída `Checkbox` ještě rozšířené konstruktory:

```
Checkbox(String label, boolean stav, CheckboxGroup group)
Checkbox(String label, CheckboxGroup group, boolean stav)
```

Zaškrtnutý checkbox získáme metodou `getSelectedCheckbox()` z třídy `java.awt.CheckboxGroup`

```
radio = new CheckboxGroup();
prep1 = new Checkbox("Ahoj", radio, true); this.add(prep1);
prep2 = new Checkbox("Nazdar", radio, false); this.add(prep2);
Checkbox vyber = radio.getSelectedCheckbox();
```

TextField – vstupní textové pole

```
public class MujEditBox extends Okno1 {
    TextField vstupTF, echoTF;
    Label vysledekLB;
    MujTextField() {
        super.setTitle(getClass().getName());
        vstupTF = new TextField(10); //max. délka 10 znaků
        echoTF = new TextField();
        Label popisVstupLB = new Label("Vstup", Label.RIGHT);
        Label popisEchoLB = new Label("Echo", Label.RIGHT);
        vysledekLB = new Label("Zde bude opsan Vstup");
        this.add(popisVstupLB);
        this.add(vstupTF); //zde se bude zadávat text
        this.add(popisEchoLB);
        this.add(echoTF); //zde se zadá zakrývací znak
        this.add(vysledekLB);
        this.setSize(250, 85);

        vstupTF.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { //reakce na Enter
                vysledekLB.setText(e.getActionCommand()); //zapsaný text
            }
        });

        echoTF.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                char echo = e.getActionCommand().charAt(0); //např. *
                if (echo == '0')
                    vstupTF.setEchoChar((char) 0); //odstranění zakrývání
                else
                    vstupTF.setEchoChar(echo); //nastavení zakrývání (pro hesla)
            }
        });
    }
}
```


JAVA

Další metody některých komponent

<code>void makeVisible(int index)</code>	... metoda třídy <code>List</code> , posune viditelnou část seznamu
<code>TextArea.SCROLLBARS_BOTH</code>	... jedna z konstant určující existenci posuvníků u <code>TextArea</code>
<code>String getSelectedText()</code>	... metoda <code>TextComponent</code> , předka tříd <code>TextArea</code> a <code>TextField</code>
<code>void setEditable(boolean b)</code>	... metoda <code>TextComponent</code> , komponenta bude jen ke čtení

Menu

Menu jsou potomci třídy `java.awt.MenuComponent`, která ale není potomkem třídy `java.awt.Component`. Pro menu jsou důležité čtyři třídy:

1. *MenuBar*

Základ menu, který je potřeba přiřadit do *Framu*. Není na obrazovce vidět, negeneruje události

2. *Menu*

Vlastní nabídka. Pomocí ní lze vytvářet i podmenu (submenu). Negeneruje ale žádné události

3. *MenuItem*

Položka menu. Generuje událost třídy `ActionEvent`

4. *CheckboxMenuItem*

Položka menu se zaškrtávkem. Negeneruje událost třídy `ActionEvent`, ale třídy `ItemEvent`

```
MenuBar lista = new MenuBar();
this.setMenuBar(lista); //platí, že okno může mít jen jedno MenuBar
Menu m1 = new Menu("Menu1", true);
lista.add(m1);
MenuItem m11 = new MenuItem("MenuItem11");
MenuItem m12 = new MenuItem("MenuItem12");
MenuItem m13 = new MenuItem("Exit", new MenuShortcut(KeyEvent.VK_E));
m1.add(m11);
m1.addSeparator(); //linka mezi položkami
m1.add(m12);
m1.add(m13); //položka přístupná i pomocí <Ctrl> + <E>
m12.setEnabled(false); //znepřístupnění volby
Menu m2 = new Menu("Menu2", true);
lista.add(m2);
CheckboxMenuItem m21 = new CheckboxMenuItem("CheckboxMenuItem");
m2.add(m21);
Menu m22 = new Menu("Menu21", true);
m2.add(m22); //podmenu
MenuItem m221 = new MenuItem("MenuItem211");
MenuItem m222 = new MenuItem("MenuItem212");
m22.add(m221);
m22.add(m222);
public void actionPerformed(ActionEvent e) {
    vystupTA.append(e.getActionCommand()); //vystupTA je zde TextArea
    vystupTA.append(e.getItem().toString()); //nebo i takto
}
```

JAVA

Metoda FocusLost()

```
vstupTF = new TextField("Přednastavený text");
this.add(vstupTF);
vstupTF.addActionListener(new VstupTF());
vstupTF.addFocusListener(new FocusVstupTF());

class FocusVstupTF extends FocusAdapter {
    public void focusLost(FocusEvent e) {
        vystupTA.append(vstupTF.getText());
    }
}
class VstupTF implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        vystupTA.append(vstupTF.getText());
    }
}
```

Oddělení výpočtů (business logic) od uživatelského prostředí (GUI)

Princip je podobný obsluze událostí. Výpočetní objekt pošle zobrazovacímu informaci o změně. Používá se k tomu rozhraní *Observer* a třída *java.util.Observable* s těmito základními metodami:

<code>void addObserver(Observer o)</code>	... zaregistrování posluchače
<code>void setChanged()</code>	... nastaví příznak změny stavu výpočtu
<code>void notifyObservers()</code>	... uvědomí posluchače, že došlo ke změně
<code>void notifyObservers(Object arg)</code>	... uvědomí posluchače, že došlo ke změně a přidá objekt změny

Zobrazovací třída musí použít rozhraní *Observer* s jedinou metodou *update()*

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;

class Citac extends Observable { //výpočtová třída oddělená od GUI
    protected int hodnota;
    public Citac(int hodnota) {
        setHodnota(hodnota);
    }
    public void setHodnota(int hodnota) {
        this.hodnota = hodnota;
        setChanged(); // došlo ke změně
        notifyObservers(new Integer(hodnota));
        // změna je reprezentovaná novým objektem
    }
    public void plusJedna() {
        setHodnota(++hodnota);
    }
    public void minusJedna() {
        setHodnota(--hodnota); //nejprve ji sníží a pak teprve předá
    }
}
```

JAVA

```
//první příjemce - textové needitovatelné pole
class CitacTextField extends TextField implements Observer {
    CitacTextField(int hodnota) {
        this.setColumns(6);
        this.setEditable(false);
        this.setText("" + hodnota);
    }
    public void update(Observable o, Object arg) { //kdo posílá a co
        this.setText(arg.toString());
    }
}

//druhý příjemce - vodorovný scrollbar
class CitacScrollbar extends Scrollbar implements Observer {
    CitacScrollbar(int hodnota, int min, int max) {
        super(Scrollbar.HORIZONTAL, hodnota, 0, min, max);
    }
    public void update(Observable o, Object arg) {
        int pozice = ((Integer) arg).intValue();
        this.setValue(pozice);
    }
}

//hlavní okno programu
public class MujObserver extends Frame {
    Citac citac;
    CitacTextField ctf; //první příjemce
    CitacScrollbar csb; //druhý příjemce
    Button plusBT, minusBT;

    MujObserver() {
        super.setTitle(getClass().getName());
        this.setLayout(new BorderLayout(10, 10));

        minusBT = new Button(" -1 ");
        this.add(minusBT, BorderLayout.WEST); //posun úplně nalevo
        minusBT.addActionListener(new MinusBT());

        ctf = new CitacTextField(0);
        this.add(ctf, BorderLayout.CENTER); //posun na střed

        plusBT = new Button(" +1 ");
        this.add(plusBT, BorderLayout.EAST); //posun úplně doprava
        plusBT.addActionListener(new PlusBT());

        csb = new CitacScrollbar(0, -10, 10) ;
        this.add(csb, BorderLayout.SOUTH); //posun dolů

        citac = new Citac(0); //instance výpočtové třídy
        citac.addObserver(ctf); //registrace posluchače
        citac.addObserver(csb);

        this.pack();
    }
}
```

JAVA

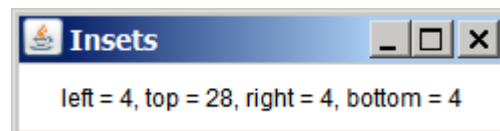
```
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});

class PlusBT implements ActionListener {    //reakce na tlačítko +1
    public void actionPerformed(ActionEvent e) {
        citac.plusJedna();
    }
}
class MinusBT implements ActionListener {    //reakce na tlačítko -1
    public void actionPerformed(ActionEvent e) {
        citac.minusJedna();
    }
}

public static void main(String[] args) {
    new MujObserver().setVisible(true);
}
```

Kontejnery

Potomci třídy *java.awt.Container* – *Frame*, *Window*, *Panel*, *ScrollPane*, *Dialog* a *FileDialog*. Zajímavá je metoda kontejneru *getInsets()*, která vrací velikost už obsazené části okna. Používá se i její překrytí pro lepší zobrazení komponent. Navracená instance třídy *Insets* má ve svých čtyřech proměnných zapsanou obsazenou část plochy:



Vybrané metody pro okna *Frame*:

void setState(int state)	... nastavení „ikonizace“
int getState()	... stav „ikonizace“
Frame.ICONIFIED, Frame.NORMAL	... konstanty používané pro „ikonizaci“
void toBack(), void toFront()	... přesun okna do pozadí nebo do popředí
boolean isShowing()	... vrací zda má okno příznak „viditelné“ (visible)
void setTitle(String title)	... nastavení titulku
String getTitle()	... vrací nastavení textu v titulku
void setResizable(boolean r)	... zakáže zvětšování/zmenšování okna
void setIconImage(Image image)	... nastavení obrázku jako ikonky titulku
Image getIconImage()	... vrací obrázek ikonky titulku
Toolkit getDefaultToolkit()	... metoda třídy <i>java.awt.Toolkit</i>
Image getImage(String filename)	... načtení obrázku ze souboru (metoda třídy <i>Toolkit</i>)
Dimension getScreenSize()	... velikost obrazovky v pixelech (metoda <i>Toolkit</i>)
int getScreenResolution()	... velikost bodů na palec (DPI, metoda <i>Toolkit</i>)

JAVA

Dialog

Používá se, pokud chceme v programu otevřít další okno. Je závislé na hlavním okně. Konstruktory:

```
Dialog(Frame owner)
Dialog(Frame owner, boolean modal)
Dialog(Frame owner, String title)
Dialog(Frame owner, String title, boolean modal)
Dialog(Dialog owner)
Dialog(Dialog owner, boolean modal)
Dialog(Dialog owner, String title)
Dialog(Dialog owner, String title, boolean modal)
```

```
void setModal(boolean b)    ... nastavení, zda okno bude modální (znemožněn přístup jinam)
boolean getModal()        ... zjištění, zda je okno modální
void show()               ... zobrazení dialogu
void hide()               ... skrytí dialogu
void dispose()            ... uzavření dialogu
```

FileDialog

Modální dialog s přednastavenou reakcí na uzavření pomocí ikonky. Konstruktory:

```
FileDialog(Frame parent)
FileDialog(Frame parent, String title)
FileDialog(Frame parent, String title, int mode)
```

```
FileDialog.LOAD, FileDialog.SAVE    ... konstanty pro mód dialogu (implicitní je LOAD)
void setMode(int mode)             ... dodatečné nastavení módu
int getMode()                     ... zjištění módu okna
void setFile(String file)          ... přednastavuje soubor ve výběru (třeba i *.java)
void setDirectory(String dir)      ... přednastavuje výchozí složku (implicitně aktuální)
String getFile()                   ... po uzavření dialogu vrací vybraný soubor (či null)
String getDirectory()              ... vrací vybranou složku
```

ScrollPane

Do instance třídy *ScrollPane* se obvykle vkládá pouze jedna komponenta, pro kterou je výhodné použití posuvníků (například objekt třídy *java.awt.Canvas* pro kreslení). Posuvníky mohou pracovat ve třech módech: *SCROLLBARS_ALWAYS*, *SCROLLBARS_AS_NEEDED* (implicitní) a *SCROLLBARS_NEVER*.

```
sp = new ScrollPane();
sp.setSize(100, 50); //nastavení velikosti komponenty
this.add(sp); //this je jako vždy nějaká instance třídy Frame
Dimension d = sp.getViewPortSize(); //velikost právě zobrazené plochy

Panel p = new Panel(new FlowLayout()); //příklad kontejneru Panel
Label velLB = new Label(); p.add(velLB);
Button nahoru = new Button("Nahoru"); p.add(nahoru);
sp.add(p); //panel s návěštím a tlačítkem jsme dali na ScrollPane
```

JAVA

Metoda validate()

Zajišťuje správné zobrazení komponenty (refresh). Je to metoda kontejneru.

```
zmenBT = new Button("Zmen font");
this.add(zmenBT);
f = new Font("SansSerif", Font.BOLD, 20);
zmenBT.setFont(f); //změna fontu na mnohem větší
this.validate();   //zajištění správného zobrazení
```

Layout manager

Objekt třídy *java.awt.LayoutManager*, mající na starosti rozmísťování komponent do kontejnerů. Nejjednodušší je *FlowLayout* (implicitní pro panely) – implicitně rozmísťuje komponenty centrovane, s pětipixelovou mezerou, jakoby do jednoho řádku. Konstruktory:

```
FlowLayout()
FlowLayout(int zarovnani)
FlowLayout(int zarovnani, int hmezera, int vmezera)
FlowLayout.CENTER, FlowLayout.LEFT, FlowLayout.RIGHT ... konstanty zarovnání
```

Dalšími Layout manažery jsou *GridLayout*, který dává komponenty do mřížky, *BorderLayout* (implicitní pro *Frame*, *Window* a *Dialog*) který umí umístit jen pět komponent a to do světových stran + do středu (příklad *MujObserver* výše), *CardLayout*, který umožňuje přepínání bloků, a *GridBagLayout*.

```
public class PriklCardLayout extends Okno2 implements ItemListener {
    Panel bloky;
    PriklCardLayout() {
        super.setTitle(getClass().getName());
        this.setLayout(new BorderLayout(10, 10));
        Choice volba = new Choice(); //volby k výběru zobrazené karty
        volba.addItem("1"); volba.addItem("2");
        volba.addItemListener(this);
        Panel p1 = new Panel(); //panel s dvěma tlačítky (pro kartu 1)
        p1.add(new Button("A")); p1.add(new Button("B"));
        Panel p2 = new Panel(); //panel s dvěma checkboxy (pro kartu 2)
        p2.add(new Checkbox("C")); p2.add(new Checkbox("D"));

        bloky = new Panel();
        bloky.setLayout(new CardLayout()); //CardLayout přidělen panelu
        bloky.add("1", p1);                //definování karet
        bloky.add("2", p2);

        this.add(bloky, BorderLayout.NORTH); //karty budou nahoře
        this.add(volba, BorderLayout.SOUTH); //výběr Choice bude dole
        this.setSize(180, 95); //this.pack();
    }
    public void itemStateChanged(ItemEvent e) { //reakce na výběr Choice
        CardLayout c = (CardLayout) bloky.getLayout();
        c.show(bloky, (String) e.getItem()); //zobrazení vybrané karty
    }
    ...//následuje už jen public static void main se setVisible(true)
```

JAVA

Třída GridBagConstraints

Je to jakýsi omezovač komponenty před vložením do nejuniverzálnějšího manažeru *GridBagLayout*.

```
GridBagLayout gbl = new GridBagLayout();  
this.setLayout(gbl);  
GridBagConstraints gbc = new GridBagConstraints(); //stačí jedno new  
gbc.anchor = GridBagConstraints.NORTH_EAST; //např., jedná se „kotvu“  
gbl.setConstraints(vkládaná_komponenta, gbc);  
this.add(vkládaná_komponenta);
```

Přehled událostí

<i>události</i>	<i>obvyklý zdroj</i>	<i>obsah zpráv</i>	<i>metody</i>
ActionEvent	tlačítko, menu	akce	actionPerformed()
AdjustmentEvent	scrollbar	změna stavu posuvníku	adjustmentValueChanged()
ComponentEvent	lib. komponenta	velikost, poloha, přístupnost	componentHidden(), componentShown(), componentMoved(), componentResized()
ContainerEvent	kontejner	přidání, ubrání prvku	componentAdded(), componentRemoved()
FocusEvent	lib. komponenta	vybrání komponenty (focus)	focusLost(), focusGained()
ItemEvent	výběry, checkbox	výběr položky	itemStateChanged()
KeyEvent	lib. komponenta	vstup z klávesnice	keyPressed(), keyTyped(), keyReleased()
MouseEvent	lib. komponenta	kurzor, kliknutí myši	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
MouseEvent (MouseMotion)	lib. komponenta	pohyb myši nad objektem	mouseDragged(), mouseMoved()
TextEvent	text. komponenta	změna textu	textValueChanged()
WindowEvent	okno	manipulace s oknem	windowOpened(), windowClosing(), windowClosed(), windowActivated(), windowDeactivated(), windowIconified(), windowDeiconified()

`String s = e paramString()` ... popis události

Přiřazení focus

K tomu mají komponenty tyto metody (ze třídy *Component*):

```
void requestFocus()      ... přiřazení focus  
boolean hasFocus()      ... dotaz, zda je přiřazen focus
```

JAVA

```
class FL extends FocusAdapter {
    public void focusGained(FocusEvent e) {
        Component c = e.getComponent(); //vrátí přímo komponentu
        focLB.setText(c.hasFocus() + " " + c.getName()
            + " " + c.isFocusTraversable()); //zjištění zda může mít focus
    }
}
```

Pro přiřazení focus další komponentě v pořadí mají kontejnery metodu **void** `transferFocus()`

Zjištění klávesy

Metodou `keyTyped(KeyEvent e)` rozhraní *KeyListener* zjistíme znak z klávesnice.

char <code>e.getKeyChar()</code>	... vrací stisknutý znak
int <code>e.getModifiers()</code>	... stisk speciálních kláves
<code>String getKeyModifiersText(int modifiers)</code>	... převod informace na text
boolean <code>e.isAltDown()</code>	... stisk <Alt>
boolean <code>e.isControlDown()</code>	... stisk <Ctrl>
boolean <code>e.isShiftDown()</code>	... stisk <Shift>

Metoda `getKeyModifiersText()` umí vrátit např. "Alt+Shift" nebo "Alt Graph"

Metodou `keyPressed()` nebo `keyReleased()` rozhraní *KeyListener* zjistíme klávesu.

char <code>e.getKeyCode()</code>	... vrací stisknutou klávesu
<code>String getKeyText(int keyCode)</code>	... převod informace na text (statická metoda)
boolean <code>e.isActionKey()</code>	... stisknuta akční klávesa

Za akční klávesy se nepovažují např. <Enter> nebo <Shift>, <Ctrl> a <Alt>.

Pro každou klávesu je definovaná konstanta začínající „VK_“, například `VK_CONTROL`, `VK_TAB`, `VK_CAPS_LOCK`, `VK_ESCAPE`, `VK_ENTER`, `VK_INSERT`, `VK_END`, `VK_PAGE_UP`, `VK_UP`, `VK_SPACE`, `VK_NUMPAD5`, atd.

Obsluha událostí myši

Metodami rozhraní *MouseListener*

<code>mouseClicked()</code> , <code>mousePressed()</code> , <code>mouseReleased()</code>	... kliknutí, stisk, uvolnění
<code>mouseEntered()</code> , <code>mouseExited()</code>	... kurzor myši se dostal/opustil komponentu

int <code>e.getModifiers()</code>	... stisk kláves myši
<code>InputEvent.BUTTON1_MASK</code>	... konstanta pro levé tlačítko
<code>InputEvent.BUTTON3_MASK</code>	... konstanta pro pravé tlačítko
<code>InputEvent.BUTTON2_MASK</code>	... konstanta pro prostřední tlačítko
<code>isAltDown()</code> , <code>isControlDown()</code> , <code>isShiftDown()</code> , <code>getComponent()</code>	... jako jinde

int <code>e.getClickCount()</code>	... vrací počet kliknutí
int <code>e.getWhen()</code>	... vrací čas vzniku události (v ms od 1.1.1970)

JAVA

```
public class PriklMouseEvent extends Frame {
    Choice vyslMysCH; //pro zobrazování výsledků
    Label mysiLB; //pro myší pokusy
    PriklMouseEvent() {
        super.setTitle(getClass().getName());
        this.setLayout(new BorderLayout(10, 10));

        vyslMysCH = new Choice();
        this.add(vyslMysCH, BorderLayout.NORTH);

        mysiLB = new Label("Myší zóna", Label.CENTER);
        mysiLB.setBackground(Color.lightGray);
        this.add(mysiLB, BorderLayout.CENTER);
        mysiLB.addMouseListener(new ML()); //zaregistrování posluchače

        this.setSize(250, 100);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

    class ML extends MouseAdapter {
        long minulyCas = 0;
        public void mouseClicked(MouseEvent e) {
            int m = e.getModifiers();
            boolean m1 = ((m & InputEvent.BUTTON1_MASK) != 0) ? true : false;
            boolean m2 = ((m & InputEvent.BUTTON2_MASK) != 0) ? true : false;
            boolean m3 = ((m & InputEvent.BUTTON3_MASK) != 0) ? true : false;
            String c = String.valueOf(vyslMysCH.getItemCount() + 1);

            if (e.isShiftDown()) {
                vyslMysCH.add(c + ". Shift; 1." + m1 + " 2." + m2 + " 3." + m3);
            }
            if (m3 && e.isAltDown()) {
                String s = e.getComponent().getName();
                vyslMysCH.add(c + ". Alt; " + s);
            }
            if (m1) {
                long tedCas = e.getWhen();
                long milisek = tedCas - minulyCas;
                if (milisek < 300 && e.getClickCount() > 1)
                    vyslMysCH.add(c + ". double-click [ms] " + milisek);
                else
                    vyslMysCH.add(c + ". click [ms] " + milisek);
                minulyCas = tedCas;
            }
            if (vyslMysCH.getItemCount() > 0)
                vyslMysCH.select(vyslMysCH.getItemCount() - 1);
        }
    }
}

... //následuje už jen public static void main se setVisible(true)
```

JAVA

Grafika

Ke kreslení se používá třída *java.awt.Graphics*. O tvorbu její instance se nemusíme starat. Kreslit se dá pouze nad třemi komponentami – *Canvas*, *Panel* a *Applet*. Vlastní činnost kreslení se provádí v metodě *paint()* třídy *Canvas*. Horní levý roh má souřadnice [0,0]

```
class Platno extends Canvas {           //naše plátno
    public void paint(Graphics g) {      //jediná užitečná metoda tř. Canvas
        int x2 = getWidth() - 25;       //getWidth - šířka komponenty Canvas
        int y2 = getHeight() - 10;      //getHeight - max. rozsah pro y
        g.drawLine(15, 20, x2, y2);    //čára
    }
}

public class PrimitivaLine extends Frame {
    PrimitivaLine() {
        super.setTitle(getClass().getName());
        Platno pl = new Platno();        //instance plátna
        this.add(pl, BorderLayout.CENTER); //přidání do okna

        this.setSize(160, 80);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) { System.exit(1); }
        });
    }
    public static void main(String[] args) {
        new PrimitivaLine().setVisible(true);
    }
}
```

Metody třídy *Graphics* (tzv. grafická primitiva):

void drawLine(int x1, int y1, int x2, int y2)	... čára z [x1,y1] do [x2,y2]
void drawRect(int x, int y, int sirka, int vyska)	... obdélník
void fillRect(int x, int y, int sirka, int vyska)	... výplň obdélníka
void clearRect(int x, int y, int sirka, int vyska)	... vyplnění obdélníka barvou pozadí
void drawOval(int x, int y, int sirka, int vyska)	... elipsa nebo kružnice
void fillOval(int x, int y, int sirka, int vyska)	... výplň elipsy nebo kružnice

Obdélník se zakulacenými rohy:

void drawRoundRect(int x, int y, int sirka, int vyska, int polomerSirky, int polomerVysky)
void fillRoundRect(int x, int y, int sirka, int vyska, int polomerSirky, int polomerVysky)

Kruhová výseč:

void drawArc(int x, int y, int sirka, int vyska, int startUhel, int vykreslovaciUhel)
void fillArc(int x, int y, int sirka, int vyska, int startUhel, int vykreslovaciUhel)

Lomená čára (posloupnost úseček):

void drawPolyLine(int[] xPoints, int[] yPoints, int nPoints)

Polygon (automaticky se poslední bod spojí s prvním):

void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)

JAVA

Metody `repaint()` a `update()`

Změníme-li kresbu na plátně, je potřeba zavolat metodu *repaint()* třídy *Canvas* pro její zobrazení. Vyvolá se tak vlastně metoda komponenty `update(Graphics g)`, která nejprve plochu překreslí barvou pozadí a pak zavolá metodu *paint()*.

Metoda *paint()* je Javou volána automaticky, když je komponenta překreslována.

Pokud nechceme vymazat předešlou kresbu, ale jen přidat další kresbu, můžeme metodu *update()* překrýt:

```
public void update(Graphics g) {
    if (prekresli == false)    //prekresli - nějaká „globální“ proměnná
        paint(g);             //provede jen naši kresbu
    else
        super.update(g);       //vymaže a nakreslí na čistou plochu
}
```

Je možné použití také přetížené metody *repaint()*, která občerství jen vybraný obdélník:

```
repaint(int x, int y, int sirka, int vyska)
```

Metoda `getGraphics()`

Instanci *g* typu *Graphics* můžeme získat i metodou *getGraphics()*, např.:

```
class MML extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent e) {
        Graphics grPl = Platno.this.getGraphics(); //jen this je MML!
        grPl.drawLine(x1, y1, e.getX(), e.getY());
    }
}
```

Barvy a orámování panelu

```
void g.setColor(Color c)    ... nastavení barvy, kterou malujeme
Color getColor()            ... zjištění barvy
```

Orámování panelu provedeme v jeho metodě *paint()*. Musíme však ještě překrýt metodu *getInsets()*, aby orámování nebylo překreslováno (toto by šlo provést i jinak, ale komplikovaněji).

```
class OramovanyPanel extends Panel {
    private final static int OKRAJ = 10;

    public Insets getInsets() {
        return new Insets(OKRAJ, OKRAJ, OKRAJ, OKRAJ);
    }

    public void paint(Graphics g) {
        Dimension d = getSize();
        g.drawRect(1, 1, d.width - 3, d.height - 3);
    }
}
```

JAVA

Ořezávání a XOR mód

```
void setClip(int x, int y, int sirka, int vyska)
```

Touto metodou zapneme omezení pro jedno následující volání *paint()*. Následující kresba se provede jen v zadaném obdélníku.

```
void setXORMode(Color c)
```

Touto metodou zapneme XOR mód pro jedno následující volání metody *paint()*. Bude se provádět nonekvivalence barev, čímž bude vždy zaručena viditelnost kreslené čáry (byť zobrazené jinou barvou).

Výpis textu

```
drawString(String str, int x, int y)
drawChars(char[] data, int offset, int length, int x, int y)
```

Tyto metody vypíší na pozici [x,y] zadaný text. Pozice však není levý horní roh textu, ale nejlevější bod základny textu (baseline), což je vlastně pomyslný řádek na němž je text napsán.

```
Font f = g.getFont();           //vrací aktuálně nastavený font
int vel = f.getSize();           //tato funkce vrátí velikost fontu
g.drawString(s, 0, 0 + vel);     //posune text, ale nepřesně!
```

Poznámka: Vracené velikosti metodami v třídě *Font* jsou udávány v DPI, ne v pixelech!

Fontová metrika

Pro práci s fonty v grafice je potřeba použít třídu *FontMetrics* s těmito metodami:

<code>getMaxAscent()</code>	... vrací vzdálenost baseline od úplného vrchu písmene (včetně diakritiky)
<code>getMaxDescent()</code>	... vrací vzdálenost baseline od úplného spodku písmene (pod „linkou“)
<code>getLeading()</code>	... vrací mezeru mezi řádky v daném písmu
<code>getHeight()</code>	... vrací výšku písma jako součet <i>getAscent()</i> , <i>getDescent()</i> a <i>getLeading()</i>
<code>stringWidth(String str)</code>	... vrací šířku zadaného textu
<code>charWidth(char ch)</code>	... vrací šířku zadaného znaku
<code>getMaxAdvance()</code>	... vrací šířku nejširšího znaku fontu

Fontovou metriku získáme metodou *getFontMetrics()* třídy *Graphics*, která má jako parametr font.

```
Font f = new Font(jméno, Font.PLAIN, 1);
Font fd = f.deriveFont(řez, (float) velikost);
FontMetrics fm = g.getFontMetrics(fd);
```

Metoda getAllFonts()

Za pomoci třídy *java.awt.GraphicsEnvironment* lze získat seznam všech fontů v systému. K tomu slouží metody *getAllFonts()* vracející pole objektů třídy *Font* a *getAvailableFontFamilyNames()*, která vrací v poli řetězců názvy rodin písem.

JAVA

```
GraphicsEnvironment ge =
    GraphicsEnvironment.getLocalGraphicsEnvironment();
Font[] fonty = ge.getAllFonts();
String[] jmena = ge.getAvailableFontFamilyNames();
jmenaCH = new Choice();
for (int i = 0; i < fonty.length; i++)
    jmenaCH.add(fonty[i].getName());
```

Obrázky

Java přímo pracuje s obrázky ve formátu GIF, JPEG a PNG. K tomu má základní třídu *java.awt.Image*. Pro načtení obrázku používáme metodu *getImage()* z třídy *java.awt.Toolkit*.

```
Image img = null;
void nactiAVykresliObrazek() {
    Toolkit t = this.getToolkit();
    img = t.getImage("kocka.jpg");
    repaint();
}
public void paint(Graphics g) {
    Dimension d = getSize();
    if (img != null)
        g.drawImage(img, 0, 0, d.width - 1, d.height - 1, this);
} //při vynechání 4 a 5 parametru bude obrázek v orig. velikosti
```

Program ale nenačte obrázek hned, ale až jej budeme chtít zobrazit. Abychom tomu zamezili, použijeme třídu *java.awt.MediaTracker*, která má tyto metody:

void addImage(Image img, int id)	... načtení obrázku okamžitě na pozadí (<i>id</i> bývá poř. číslo)
void waitAll()	... čekání na načtení všech obrázků
void waitAll(long ms)	... čekání na načtení všech obrázků, ale jen určený počet ms
void waitID(int id)	... čekání na načtení jen obrázku poř. č. <i>id</i>
void waitID(int id, long ms)	... čekání na načtení obrázku <i>id</i> , ale jen určený počet ms
boolean checkAll()	... vrací true v případě, že už jsou všechny obrázky načteny
boolean checkID(int id)	... vrací true v případě, že obrázek <i>id</i> je již načten
boolean isErrorAny()	... vrací true když nastala chyba při načítání obrázků
boolean isErrorID(int id)	... vrací true když nastala chyba při načtení obrázku <i>id</i>

```
this.setCursor(new Cursor(Cursor.WAIT_CURSOR));
Toolkit t = Toolkit.getDefaultToolkit();
MediaTracker mt = new MediaTracker(this); //získání MediaTracker
img1 = t.getImage("../kocka.jpg"); //dobré použít file.separator
mt.addImage(img1, 0);
img2 = t.getImage("a:\\kouzelnik.jpg");
mt.addImage(img2, 1);
try {
    mt.waitAll(); //anebo třeba jen mt.waitForID(0);
}
catch (InterruptedException e) {
    e.printStackTrace();
}
this.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
```

JAVA

Monitorování natažení obrázku

Metoda *drawImage()* má poslední parametr typu *ImageObserver*. Rozhraní *ImageObserver* má metodu

```
boolean imageUpdate(Image img, int infoflags, int x, int y, int width, int height)
```

kteřá je automaticky průběžně vyvolávaná metodou *drawImage()*. Obrázek pak začne vykreslovat až obdrží hodnotu false.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*; //balík s třídami pro obrázky

class Platno extends Canvas implements ImageObserver {
    Image img;
    PostupPrace pp;
    Platno(PostupPrace p) { pp = p; }
    public boolean imageUpdate(Image im, int info,
                               int x, int y, int w, int h) {
        double celkVyska = (double) im.getHeight(this);
        int proc = (int) ((y / celkVyska) * 100.0);
        pp.setProcenta(proc);
        boolean hotovo = ((info & (ERROR | ALLBITS)) != 0);
        if (hotovo) repaint();
        return !hotovo;
    }
    void nactiObrazek(String jmeno) {
        Toolkit t = this.getToolkit();
        img = t.getImage(jmeno);
        pp.nulujProcenta();
        repaint();
    }
    public void paint(Graphics g) {
        Dimension d = getSize();
        if (img != null)
            g.drawImage(img, 0, 0, d.width - 1, d.height - 1, this);
    } //existují i konstruktory s definovanou barvou pozadí obrázku
}

public class NacitaniObrazku extends Frame {
    Platno pl;
    PostupPrace pp;
    NacitaniObrazku() {
        super.setTitle(getClass().getName());
        pp = new PostupPrace(true);
        pl = new Platno(pp); this.add(pl, BorderLayout.CENTER);

        Button b1 = new Button("Načti obrázek");
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String fs = System.getProperty("file.separator");
                pl.nactiObrazek("../" + fs + "kouzelnik.jpg");
            }
        });
    }
}
```

JAVA

```
Panel p = new Panel();
p.add(bl); p.add(pp);
this.add(p, BorderLayout.SOUTH);

this.setSize(350, 250);
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(1);
    }
});
}
private final static int OKRAJ = 10;
public Insets getInsets() {
    return new Insets(OKRAJ * 3, OKRAJ, OKRAJ, OKRAJ);
}
public static void main(String[] args) {
    new NacitaniObrazku().setVisible(true);
}
}

class PostupPrace extends Canvas {
    private int procent = 0;
    private static final int OKRAJ = 3;
    private int vys = 10 + OKRAJ * 2;
    private boolean zobrazovatCislo;
    public PostupPrace(boolean zobr) { zobrazovatCislo = zobr; }
    public void nulujProcenta() { procent = 0; }
    public void setProcenta(int p) {
        if (procent < p) { //at se furt nepřekresluje
            procent = p; repaint();
        }
    }
    public Dimension getMinimumSize() { return new Dimension(102, vys); }
    public Dimension getPreferredSize() { return getMinimumSize(); }
    public void paint(Graphics g) {
        Dimension d = getSize();
        g.setColor(Color.blue);
        if (procent >= 99) procent = 100;
        if (zobrazovatCislo == true) {
            g.clearRect(1, 1, d.width - 2, d.height - 2);
            String s = "" + procent + "%";
            FontMetrics fm = g.getFontMetrics();
            int sir = fm.stringWidth(s);
            int xc = (d.width - sir) / 2;
            g.drawString(s, xc, d.height - 1 - OKRAJ);
            g.setXORMode(Color.white);
        }
        g.drawRect(0, 0, d.width - 1, d.height - 1);
        double p = procent / 100.0;
        int xp = (int) (p * d.width);
        g.fillRect(1, 1, xp - 2, d.height - 2);
    }
}
```

JAVA

Aplety (applets)

Aplety jsou programy používané na www stránkách. Nemají metodu *main()*, spouští se z html kódu.

```
import java.awt.*;
import java.applet.*;

public class PrvniAplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("První aplet", 10, 20);
    }
}
```

Třída *java.applet.Applet* je přímým potomkem *java.awt.Panel*, platí tedy pro ni obdobná pravidla.

Překlad apletů probíhá stejně jako u programů. Přeložený soubor se pak volá v kódu html:

```
<HTML>
<BODY>
Nepovinný text <b>před</b> apletem
<p>
<APPLET CODE="PrvniAplet.class" WIDTH=200 HEIGHT=30>
</APPLET>
<p>
Nepovinný text <b>za</b> apletem
</BODY>
</HTML>
```

Pro ladění apletů můžeme využít *appletviewer.exe*, kterému sa jako parametr dává název souboru html. Ignoruje v něm vše kromě konstrukce *<APPLET> </APPLET>*

Základní metody třídy *Applet*:

<i>init()</i>	... činnost prováděná při zavedení stránky www
<i>start()</i>	... činnost prováděná při náběhu apletu do zobrazované části stránky
<i>paint()</i>	... volána automaticky pro překreslení nebo vyvolána metodou <i>repaint()</i>
<i>stop()</i>	... činnost prováděná když se aplet dostane do nezobrazované části stránky
<i>destroy()</i>	... činnost prováděná při zavírání stránky www

Parametry můžeme apletu předávat pomocí html konstrukce *<PARAM>*

```
<HTML>
<BODY>
<APPLET CODE="MujAplet.class" WIDTH=200 HEIGHT=30>
    <PARAM NAME="obrazek" VALUE="kocka.jpg"
    <PARAM NAME="zvuk" VALUE="mnau.wav"
</APPLET>
</BODY>
</HTML>
```

K jejich načtení slouží metoda *getParameter()*

```
String jmObr = getParameter("obrazek");
```

Stejný aplet můžeme na jedné stránce www zavolat i vícekrát, například s jinými parametry.

JAVA

HTML pro aplety

CODE	... jméno spuštěné třídy
WIDTH, HEIGHT	... šířka a výška apletu; musí být vždy uvedeny
CODEBASE	... relativní nebo absolutní url cesta, kde se budou hledat soubory tříd
NAME	... jméno ¹ apletu, což lze využít pro komunikaci mezi aplety navzájem
ALT	... zástupný text, který se použije v případě nemožnosti spustit aplet
ALIGN	... zarovnání (<i>left, right, top, absmiddle, bottom</i>)
HSPACE	... vodorovné odsazení v pixelech
VSPACE	... svislé odsazení v pixelech
ARCHIVE	... jméno JAR souboru s třídami (a samozřejmě i s dalšími soubory)

¹Metoda `getAppletContext().getApplet(jméno)` vrací instanci třídy *Applet*

Další metody apletu

<code>URL getCodeBase()</code>	... vrací cestu zadanou v html v CODEBASE
<code>void showStatus(String msg)</code>	... zapíše text do stavového řádku
<code>Image getImage(URL url, String name)</code>	... načtení obrázku ze zadané cesty

Přehrávání zvuku

Třída *Applet* má pro přehrávání audioklipu metodu `play(URL url, String name)`
Zvuky je ale v apletu i aplikaci možné přehrávat také pomocí rozhraní *AudioClip*. Má tři metody:

<code>play()</code>	... jednorázové přehrávání audioklipu
<code>loop()</code>	... přehrávání audioklipu stále dokola
<code>stop()</code>	... zastavení přehrávání audioklipu

V apletu získáme instanci audioklipu voláním metody `getAudioClip(URL url, String name)`

```
public class HraciAplet extends Applet implements ActionListener {
    Button jedBT;
    AudioClip zvuk;
    boolean zastav = false;
    public void init() {
        zvuk = getAudioClip(getCodeBase(), "zvonek.wav");
        jedBT = new Button("Hraj"); this.add(jedBT);
        jedBT.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        if (zastav == false) { zvuk.loop(); jedBT.setLabel("Stop"); }
        else { zvuk.stop(); jedBT.setLabel("Hraj"); }
        zastav = !zastav;
    }
    public void stop() {
        zvuk.stop();
        if (zastav == true) { jedBT.setLabel("Hraj"); zastav = false; }
    }
}
```

JAVA

```
<HTML>
<BODY>
Hrající aplet
<APPLET CODE="HraciAplet.class" WIDTH=300 HEIGHT=30
        CODEBASE="."
>
<!-- toto je poznámka v HTML -->
</APPLET>
</BODY>
</HTML>
```

Použití jednoho souboru pro aplet i aplikaci

```
/*
<APPLET CODE="ApletAAplikace.class" WIDTH=100 HEIGHT=55>
</APPLET>
*/

public class ApletAAplikace extends Applet {
    Button zijuBT;
    int pocet = 0;

    public void init() {
        zijuBT = new Button("Žiju");
        add(zijuBT);
        zijuBT.addActionListener(new BAL());
    }

    class BAL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            pocet++;
            repaint();
        }
    }

    public void paint(Graphics g) {
        g.drawString("" + pocet + ". stisk", 30, 45);
    }

    public static void main(String[] args) { //tu aplet nepoužije
        Frame okno = new Frame("Aplikace");
        Applet apl = new ApletAAplikace();
        okno.add(apl); //vždyť je to vlastně panel
        apl.init();    //místo pomyslného konstruktoru
        okno.setSize(120, 85);
        okno.setVisible(true);
        okno.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

JAVA

Archiv JAR

Typicky používaný pro kompletní souborovou strukturu programu. Výhodou je i pojmenování dle libosti. Největší vymožeností však je, že pro spuštění programu není nutné archiv rozbalovat.

```
D:\java>jar.exe cf jméno_archivu jména_souborů
```

Tento příkaz je nejklassičtější, vytvoří archiv. Do archivu je přidán i tzv. manifest. V archivu mohou být i soubory s kterými pak program pracuje. Např.:

```
D:\java>jar cf moje.jar *.class *.wav
```

```
<HTML>
<BODY>
Hrající applet
<APPLET CODE="HraciApplet.class" WIDTH=300 HEIGHT=30
        CODEBASE="."
        ARCHIVE="moje.jar"
>
</APPLET>
</BODY>
</HTML>
```

Spuštění aplikace v archivu

Pomocí manifestu musíme určit hlavní třídu s metodou *main()*. K tomu se používá klíč *Main-Class*. Do textového souboru je tedy potřeba zapsat

Main-Class: Nejprehlednejsi

a tento soubor (například popis.txt) do manifestu přidat:

```
D:\java>jar cfm druhe.jar popis.txt *.class
```

Tím je řečeno, že hlavní třída je *Nejprehlednejsi.class*. Nyní soubor *druhe.jar* spustíme:

```
D:\java>java -jar druhe.jar
```

Poznámka: Pokud jsem v *popis.txt* nedal za klíč <Enter>, tj. nepřešel jsem na další řádek, klíč se do manifestu nepřenesl.

Čeština

Java podporuje tato 8-bitová kódování češtiny:

ISO8859_2	... norma pro Unix, někdy označovaná jako ISO LATIN2
Cp1250	... norma pro MS Windows
Cp852	... kódování používané MS-DOS, označované jako Latin 2

Konzolové okno používá kódování Cp852.

V mezinárodních systémech jsou standardy ISO8859_1 pro Unix a Cp1252 pro MS Windows.

Pro Unicode má Java zkratku `Unicode` a pro UTF-8 má zkratku `UTF8`

JAVA

Soubory Unicode

Text v Unicode může být kódovaný buď Little Endian (vyšší byte je zapsán až druhý) nebo Big Endian (vyšší byte je zapsán první). Aby bylo jasné kódování textu, je na začátku souboru umístěn předposlední znak kódování Unicode, tedy dvojice bajtů buď FF FE nebo FE FF.

FE FF 42 00 E9 00 0F 01 61 00
 B é ď a

UTF-8

Slouží k zápisu Unicode znaků do osmibitového systému.

Unicode	přepis do UTF-8	
0000-007F	0xxx xxxx	
0080-07FF	110x xxxx 10xx xxxx	← zde jsou české akcentované znaky
0800-FFFF	1110 xxxx 10xx xxxx 10xx xxxx	

Znak	Unicode	Unicode	UTF-8	UTF-8
B	0042	0000 0000 0100 0010	0100 0010	42
é	00E9	0000 0000 1110 1001	1100 0011 1010 1001	C3 A9
ď	010F	0000 0001 0000 1111	1100 0100 1000 1111	C4 8F
a	0061	0000 0000 0110 0001	0110 0001	61

Konvertování češtiny

Změny kódování se dějí pomocí tříd *OutputStreamWriter* a *InputStreamReader*.

```
public class NaKonzoli {
    public static void main(String[] args) throws Exception {
        OutputStreamWriter od = new OutputStreamWriter(System.out);
        System.out.println(od.getEncoding()); //napíše Cp1250

        OutputStreamWriter o = new OutputStreamWriter(System.out, "Cp852");
        System.out.println(o.getEncoding()); //nastavili jsme Cp852
        PrintWriter p = new PrintWriter(o);
        p.print("Příšerně žluťoučký kůň úpěl ďábelské ódy.\n");
        p.flush();
    }
}
```

Pro zápis do souboru by to bylo podobné:

```
FileOutputStream fw = new FileOutputStream(jméno_souboru);
OutputStreamWriter ofw = new OutputStreamWriter(fw, kódování);
PrintWriter pofw = new PrintWriter(ofw);
String akcenty = "áčďěěíňůřšťůůž";
pofw.println(akcenty);
pofw.println(akcenty.toUpperCase()); //převedou se vždy správně
pofw.close();
```

JAVA

Konvertor

```
import java.io.*;

public class Konvertor {
    public static void main(String[] args) throws Exception {
        if (args.length != 4) { //počet parametrů je jiný než 4
            System.out.println("Pouziti:\n" +
                " java Konvertor inkod outkod infile outfile\n" +
                " kody: Cp1250, Cp852, ISO8859_2, UTF8, Unicode");
            System.exit(1);
        }

        FileInputStream fr = new FileInputStream(args[2]); //infile
        InputStreamReader ifr = new InputStreamReader(fr, args[0]);

        FileOutputStream fw = new FileOutputStream(args[3]);
        OutputStreamWriter ofw = new OutputStreamWriter(fw, args[1]);

        int c;
        while ((c = ifr.read()) != -1)
            ofw.write(c);

        ofw.close();
        fr.close();
    }
}
```

Metoda getBytes()

Tato metoda třídy *String* vrací pole bajtů reprezentující text v zadaném kódování.

```
byte[] getBytes(String zkratka_kodovani)
```

Třída *String* má také jeden zajímavý konstruktor:

```
String(byte[] bajty, String zkratka_kodovani)
```

Těchto vlastností se dá využít pro napsání konvertoru i pomocí přístupů třídou *RandomAccessFile*:

```
RandomAccessFile fr = new RandomAccessFile("vstup.Cp1250", "r");
int delka = (int) fr.length();
byte[] pole = new byte[delka];
fr.read(pole); //kompletní načtení
fr.close();

String s = new String(pole, "Cp1250");

RandomAccessFile frw = new RandomAccessFile("vystup.Cp852", "rw");
frw.write(s.getBytes("Cp852"));
frw.close();
```

JAVA

Internacionalizace (i18n)

Třída *Locale* z balíku *java.util* poskytuje informace o lokalizaci.

```
OutputStreamWriter o = new OutputStreamWriter(System.out, "Cp852");
PrintWriter p = new PrintWriter(o);
Locale d = Locale.getDefault(); //takto lehce získáme instanci
p.println("Země : " + d.getCountry()); //CZ
p.println("Jazyk: " + d.getLanguage()); //cs
p.println("Země : " + d.getDisplayCountry()); //Česká republika
p.println("Jazyk: " + d.getDisplayLanguage()); //čeština
p.println("ISO země : " + d.getISO3Country()); //CZE
p.println("ISO jazyk: " + d.getISO3Language()); //ces
```

Třída *java.awt.Component* má metody *getLocale()* a *setLocale()*. Tak můžeme nastavit každé komponentě jinou lokalitu. Lokalitu můžeme i „vytvořit“:

```
usLocale = new Locale("en", "US"); //načte systémovou lokalitu
```

Každá třída (u které to má smysl) má statickou metodu *getAvailableLocales()*, která vrátí podporované lokality v poli, například:

```
Locale[] loks = DateFormat.getAvailableLocales();
```

Formátování čísel a měny

Používá se třída *java.text.NumberFormat* na kterou lze převést primitivní datové typy *long* a *double*. U třídy *NumberFormat* se pro vytvoření instance nevolá konstruktor (příkazem *new*).

```
OutputStreamWriter o = new OutputStreamWriter(System.out, "Cp852");
PrintWriter p = new PrintWriter(o);
NumberFormat nf;
double d = 1234567.89;
Locale[] lo = { new Locale("cs", "CZ"), new Locale("sk", "SK"),
               new Locale("en", "US"), new Locale("de", "DE") };
for (int i = 0; i < lo.length; i++) {
    nf = NumberFormat.getNumberInstance(lo[i]); //takto se získává instance
    String s = nf.format(d); //metoda třídy NumberFormat
    p.println(s + "\t" + lo[i].getDisplayName());
    p.flush();
}
```

Úplně stejně se formátuje i měna, jen s použitím jiné statické metody a to *getCurrencyInstance()*

```
nf = NumberFormat.getCurrencyInstance(lo[i]); //instance pro formátování měny
nf = NumberFormat.getPercentInstance(lo[i]); //instance pro formátování procent
```

Formátování čísel podle vzoru

Používá se třída *DecimalFormat*, která je potomkem *NumberFormat*. Instance se tvoří klasicky přes konstruktor příkazem *new*

JAVA

```
DecimalFormat df = new DecimalFormat("0.0#");
String s = nf.format(d);
df.applyPattern("#,##0"); //změna vzoru
String t = nf.format(d);
```

Formátování datumu a času

Používá se třída *java.text.DateFormat* podobně jako *NumberFormat*.

```
OutputStreamWriter o = new OutputStreamWriter(System.out, "Cp852");
PrintWriter p = new PrintWriter(o);
DateFormat df;
Date d = new Date();
Locale[] lo = DateFormat.getAvailableLocales();
for (int i = 0; i < lo.length; i++) {
    df = DateFormat.getDateInstance(DateFormat.DEFAULT, lo[i]);
    String s = df.format(d);
    p.println(s + "\t" + lo[i].getDisplayName());
    p.flush();
}
```

První parametr metody *getDateInstance()* definuje mód výpisu. Kromě *DEFAULT* může být ještě *SHORT*, *MEDIUM*, *LONG* a *FULL*. Identické konstanty lze použít i v metodě *getTimeInstance()*

```
df = DateFormat.getTimeInstance(DateFormat.DEFAULT, lo[i]);
df = DateFormat.getDateInstance(DateFormat.LONG, DateFormat.SHORT,
    lo[i]); //první je mód datumu, druhý času
```

Formátování datumu a času podle vzoru

Používá se třída *SimpleDateFormat*. Instance se tvoří obdobně jako u *DecimalFormat* – příkazem *new*

```
Date d = new Date();
Locale lo = Locale.getDefault();
SimpleDateFormat sdf;
sdf = new SimpleDateFormat("EEE, dd. MMMM yyyy [HH.mm.ss:SSS]", lo);
String s = sdf.format(d);
```

Porovnávání řetězce a abecední řazení

Používá se třída *java.text.Collator*, která spolupracuje s lokalizací

```
Collator defCol = Collator.getInstance(); //získání instance
for (int i = 0; i < ret.length; i++) {
    for (int j = i + 1; j < ret.length; j++) { //bublanka
        if (defCol.compare(ret[i], ret[j]) > 0) {
            String tmp = ret[i]; ret[i] = ret[j]; ret[j] = tmp;
        }
    }
}
```

JAVA

Řazení postupnou záměnou se můžeme vyhnout použitím rozhraní *Comparator*

```
import java.util.*;
import java.text.*;
import java.io.*;

public class Razeni {
    public static void main(String[] args) throws Exception {
        OutputStreamWriter o = new OutputStreamWriter(System.out, "Cp852");
        PrintWriter p = new PrintWriter(o);

        String[] ret = { "plátno", "plankton", "plachta", "plagiát", "plac",
            "nový věk", "Nový Svět", "Nový svět", "nový Svět", "nový svět",
            "Nový Svet", "Nový svet", "abc traktoristy", "ABC nástrojaře",
            "abc nástrojaře", "ABC kováře", "ABC klempíře", "abc frézaře",
            "ABC", "Abc", "abc", "A", "a" }; //ted' je seřazeno přesně opačně

        Arrays.sort(ret, new CeskyAbecedniComparator());

        for (int i = 0; i < ret.length; i++)
            p.print(ret[i] + "\n");
        p.flush();
    }
}

class CeskyAbecedniComparator implements Comparator {
    Collator defCol = Collator.getInstance();

    public int compare(Object o1, Object o2) {
        String s1 = (String) o1;
        String s2 = (String) o2;
        return defCol.compare(s1, s2);
    }
}
```

Detekce hranice slov, vět, řádků

Používá se třída *java.text.BreakIterator*, kde instanci získáme některou z těchto metod:

<code>getWordInstance(Locate l)</code>	... detekuje hranice slov
<code>getSentenceInstance(Locate l)</code>	... detekuje hranice vět
<code>getLineInstance(Locate l)</code>	... detekuje místa možného zlomu řádku
<code>getCharacterInstance(Locate l)</code>	... detekuje hranice spřežek (v češtině jen ch)

Další metody této třídy:

<code>void setText(String text)</code>	... nastavuje zkoumaný text
<code>int first(), next(), previous(), last()</code>	... posunování po textu
<code>int following(int offset)</code>	... vrací pravý hraniční index od zadané pozice
<code>int preceding(int offset)</code>	... vrací levý hraniční index od zadané pozice
<code>BreakIterator.DONE</code>	... tuto hodnotu vrací metoda <i>next()</i> když došla na konec