

Programovací jazyk Java

Kurz 2014-03-05
Praha, Internet Info

Pavel Tišnovský
tisnik@centrum.cz



Orientační obsah kurzu (1)

- Stručná charakteristika a historie programovacího jazyka Java
- Verze jazyka Java
- Použití Javy: desktop, server, applety, mobilní zařízení
- Formát zápisu algoritmů zapsaných v programovacím jazyku Java
- Proměnné, numerické a znakové hodnoty, operátory
- Podmíněné příkazy: if-else a switch
- Tvorba smyček: for, while, do-while
- Pole a řetězce
- Základy objektově orientovaného programování: zapouzdření, dědičnost, polymorfismus
- Třídy a objekty v Javě, rozhraní
- Standardní třídy dostupné v Javě (+ kolekce)

Orientační obsah kurzu (2)

- Využití Eclipse při programování v Javě
 - Automatické doplňování kódu
 - Poloautomatická oprava chyb
 - Refaktoring
- Lekce „na přání“
 - GUI (AWT, Swing, layout managers, listeners)
 - Java 2D
 - Networking

Obsah jednotlivých částí

1. Stručná charakteristika a historie Javy
2. Použití Javy: desktop, server, web, mobily
3. Porovnání rychlosti Javy s dalšími jazyky
4. Formát zápisu algoritmů zapsaných v Javě
5. Proměnné, numerické a znakové hodnoty
6. Operátory a výrazy
7. Podmíněné příkazy if-else a switch
8. Programové smyčky
9. Pole
10. Řetězce

Obsah jednotlivých částí (pokračování)

- 11. Základy OOP
- 12. Třídy a objekty v Javě
- 13. Rozhraní
- 14. Kolekce
- 15. Standardní třídy dostupné v Javě
- 16. Zachytávání výjimek
- 17. Vlákna
- 18. Grafické uživatelské rozhraní
- 19. JDBC
- 20. Práce s Eclipse

Část 1

Stručná charakteristika a historie programovacího jazyka Java



Co o Javě napsali její autoři?

The Java Programming Language is a general-purpose, concurrent, strongly typed, class-based object-oriented language. It is normally compiled to the bytecode instruction set and binary format defined in the Java Virtual Machine Specification.

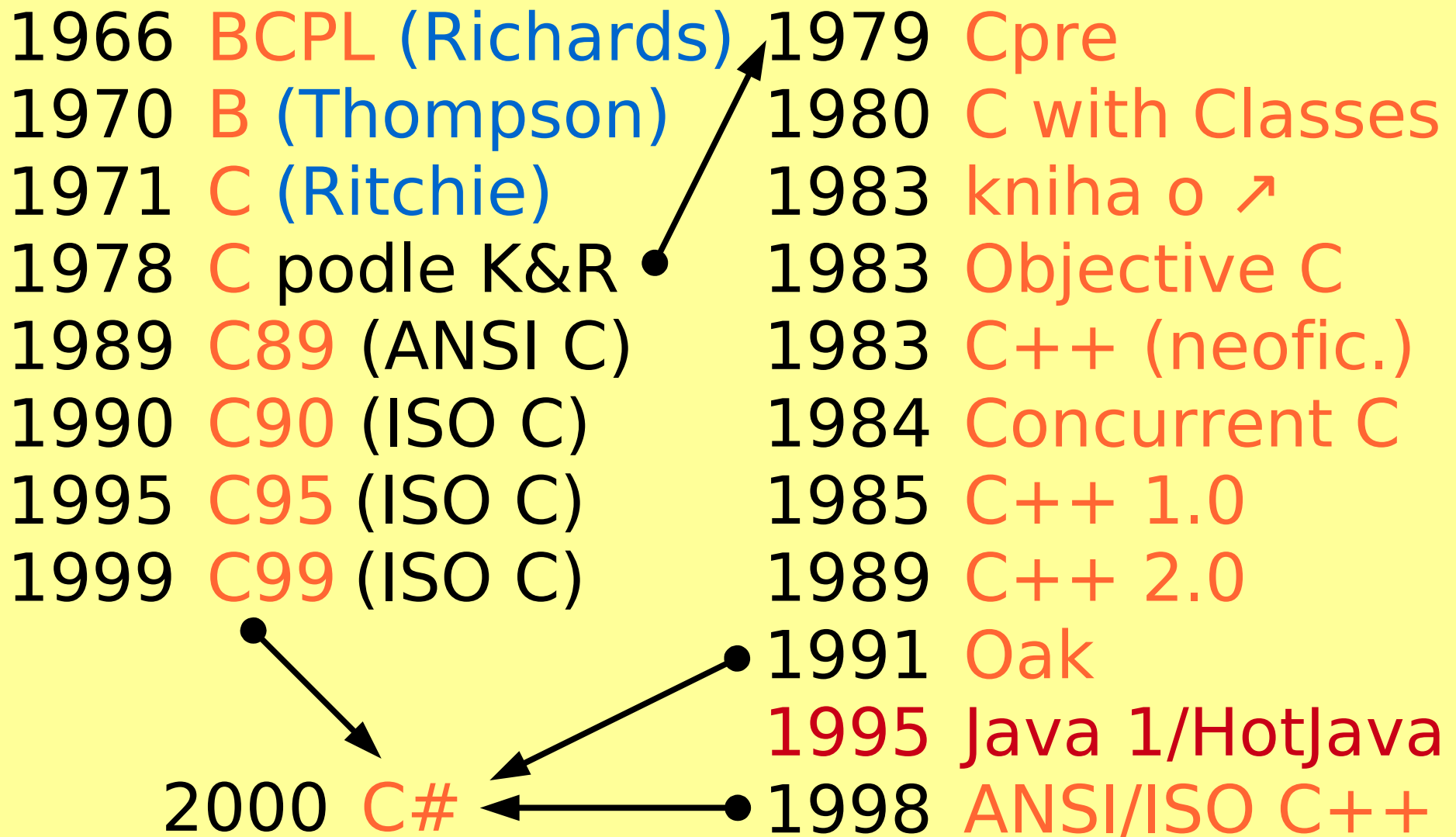
Stručná charakteristika Javy

- programovací jazyk určený pro řešení obecných problémů
 - aplikace ovládané z příkazového řádku
 - aplikace s plnohodnotným GUI
 - systémy klient/server a n-vrstvé systémy
 - applety
 - distribuované výpočty (+dnes populární cloud)
- objektově orientovaný (OOP) jazyk
- jazyk nezávislý na platformě (OS, CPU)
- podpora pro běh ve více vláknech přímo v programových konstrukcích jazyka

Java a další programovací jazyky

- Syntaxe založená především na C a C++
 - velké množství programátor
 - snadný přechod z C/C++ na Javu
- Garbage collection
 - použito u mnoha dynamických jazyků: LISP, Python, Smalltalk
- Kolekce (collection)
 - částečně inspirováno Smalltalkem, funkce podobná STL v C++
- Rozhraní (interface)
 - inspirováno zejména jazykem Objective C
- Originální Wirthův Pascal a další (VB)
 - virtuální stroj

Historie jazyka C a jeho „potomků“, včetně počátků Javy



Historie Javy

1991 Oak (Gosling)

1995 Java 1 a browser HotJava

1996 Java 1.02 (oprava nedostatků Javy 1)

1996 JDK 1.1

1998 JDK 1.2 \Rightarrow Java 2 (velký úspěch Javy)

2004 Java 1.5 \approx JDK 5.0 (rozšíření jazyka)

současnost Java 1.6 \approx JDK 6.0

přechod na Java 1.7 \approx JDK 7.0)



OpenJDK

2007

- firma Sun
- založeno na Java 1.7
- cca 4% uzavřeného kódu
- změny korespondují se Sun JDK



OpenJDK a IcedTea

2009-2010

- Úpravy a opravy chyb dalšími firmami a vývojáři (Red Hat,...)
- Větev OpenJDK6
- IcedTea6 a IcedTea7
 - založen na open source nástrojích a knihovnách
 - Používán v Linuxových distribucích
- Sun → Oracle
 - budoucnost Javy?



IcedTea

- Vývoj zahájen firmou Red Hat Inc.
- 7. června 2007
 - Oficiální představení projektu
- gcj/classpath
- Struktura nástrojů pro překlad
- Infrastruktura
 - Mercurial repo
 - Wiki
 - Mail lists
 - Bugzilla



Příklad – IcedTea v Ubuntu

```
Actions  Undo  Package Resolver  Search  Options  Views  Help
C-T: Menu  ?: Help  q: Quit  u: Update  g: Download/Install/Remove Pkgs
aptitude 0.4.11.11 Will free 7803kB of disk space

i openjdk-6-jdk 6b14-1.4.1 6b14-1.4.1
i pkg-config 0.22-1 0.22-1
i python-debian 0.1.12ubun 0.1.12ubun
--- universe - Unsupported Free Software. (1)
--- doc - Documentation and specialized programs for viewing documentation (11)
--\ editors - Text editors and word processors (24)
--\ main - Fully supported Free Software. (23)
i ed 0.7-3ubunt 0.7-3ubunt
i nano 2.0.9-2 2.0.9-2
i openoffice.org 1:3.0.1-9u 1:3.0.1-9u

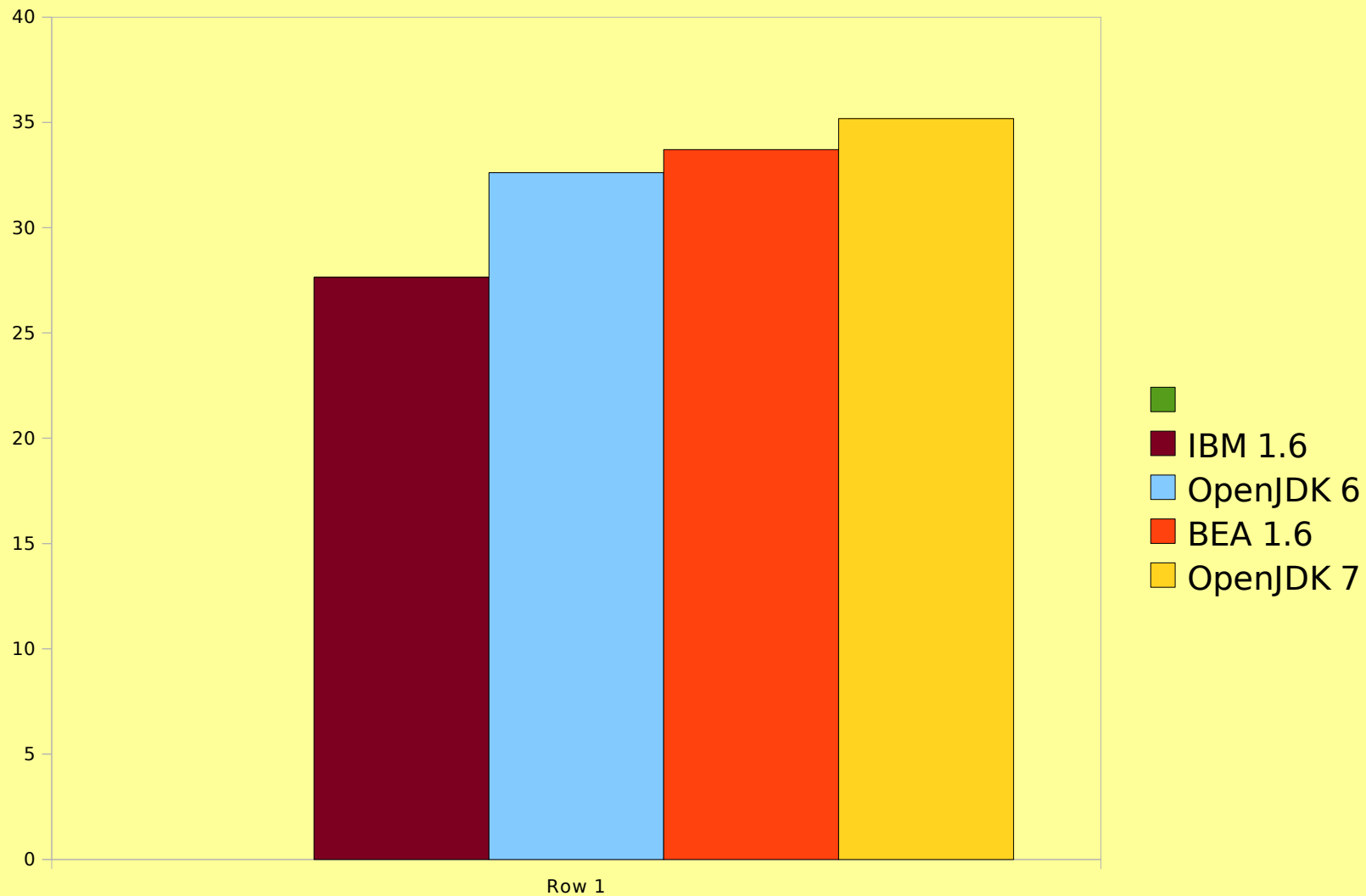
OpenJDK Development Kit (JDK)
OpenJDK is a development environment for building applications, applets, and
components using the Java programming language.

The packages are built using the IcedTea build support and patches from the
IcedTea project.
Homepage: http://openjdk.java.net/
```

Příklad – IcedTea v Ubuntu

```
tisnik@bender:~$ java -version
java version "1.6.0_0"
OpenJDK Runtime Environment (IcedTea6 1.4.1) (6b14-1.4.1-0ubuntu11)
OpenJDK Client VM (build 14.0-b08, mixed mode, sharing)
tisnik@bender:~$
```


OpenJDK - výkon



Část 2

Použití Javy: desktop, server,
applety a mobilní zařízení

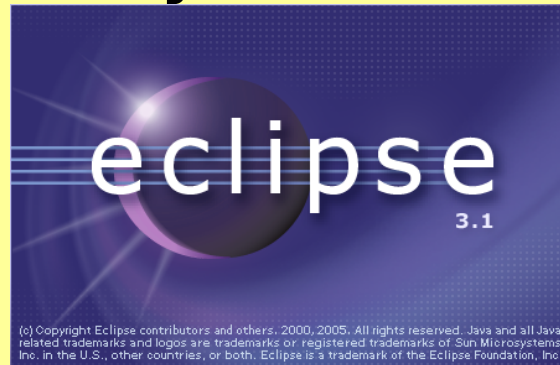


Java na desktopu

- Aplikace orientované na příkazový řádek
- Aplikace s grafickým uživatelským rozhraním:

- **AWT**
- **Swing**

- Applety

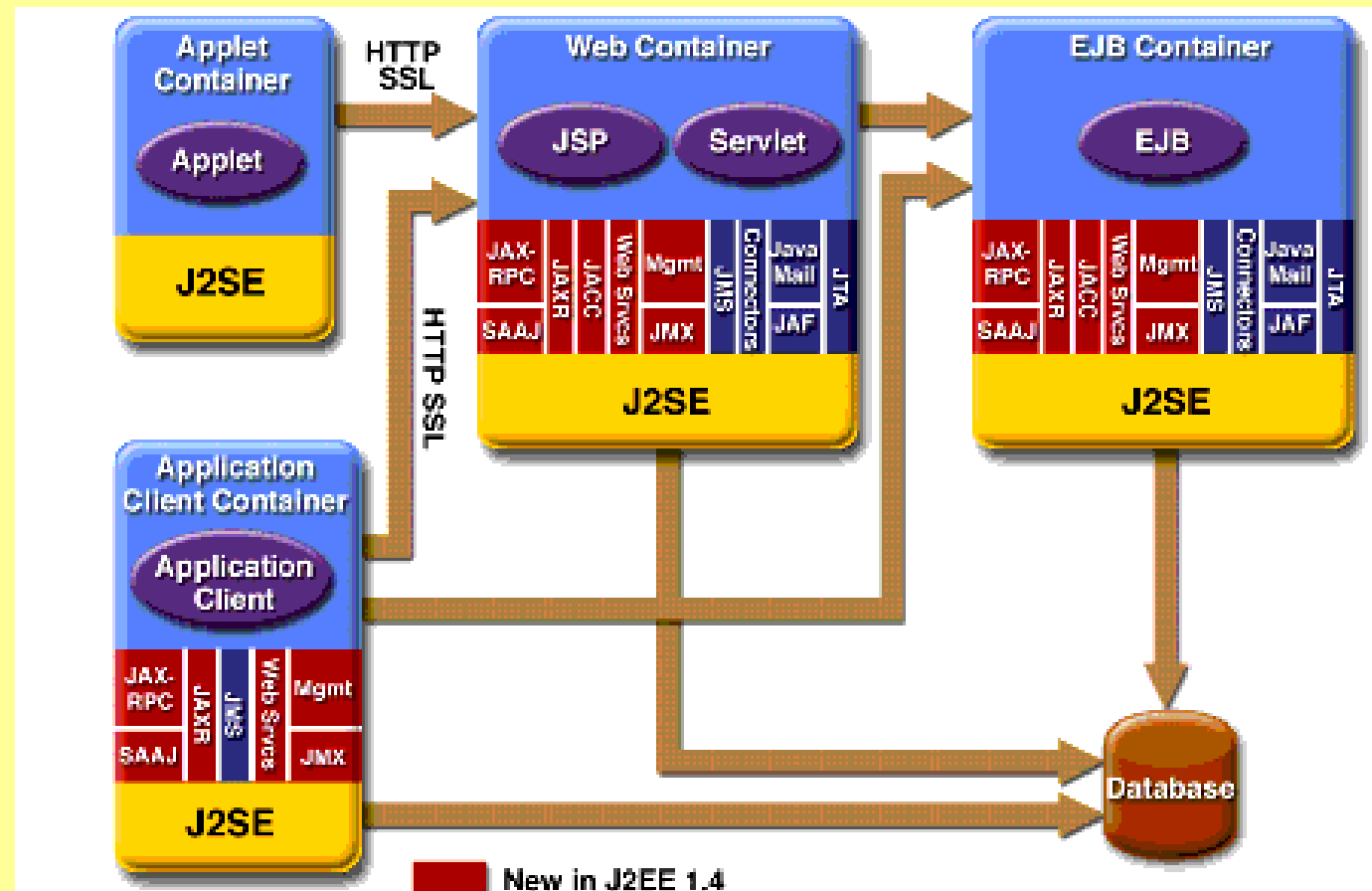


zobrazované ve webovém prohlížeči:

- stále podporovány, jejich obliba však klesá (existují i jiná řešení založená na Web 2.0)
- Java zabudovaná do dalších aplikací:
 - **CADy, OpenOffice.org a mnoho dalších**

Java na serverech

- JSP a servlety
- webové služby
 - web services
- J2EE:
 - JDBC
 - JSP
 - EJB
 - JNDI
 - JMS
 - ...



Java na mobilních zařízeních

- Speciální edice pro mobilní zařízení:
 - J2ME
- Některé knihovny v této edici chybí nebo mají zredukovanou funkcionalitu
- Další knihovny byly naopak přidány:
 - konfigurace zařízení
 - ovládání stylusem a numerickou klávesnicí
 - komunikace
- Výsledek:
 - menší nároky na výkon CPU
 - snížené množství alokované paměti

Část 3

Porovnání rychlosti Javy s vybranými programovacími jazyky



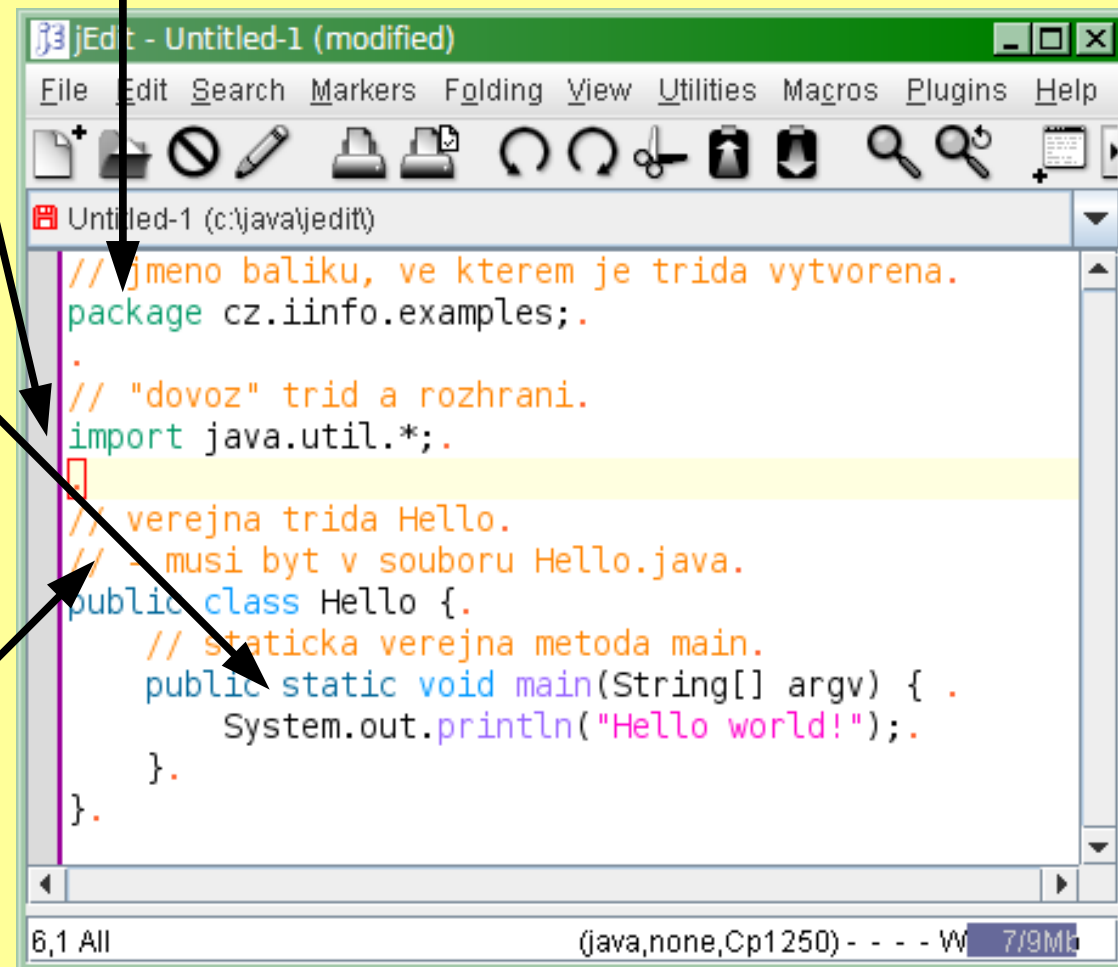
Část 4

Formát zápisu algoritmů zapsaných v programovacím jazyku Java



Formát zápisu algoritmů v Javě

- Specifikace balíku
- Příkaz „import“
- Deklarace
 - tříd
 - rozhraní
 - metod
- Příkazy
 - podmínky
 - smyčky
- Výrazy
- Poznámky



„Hello world!“

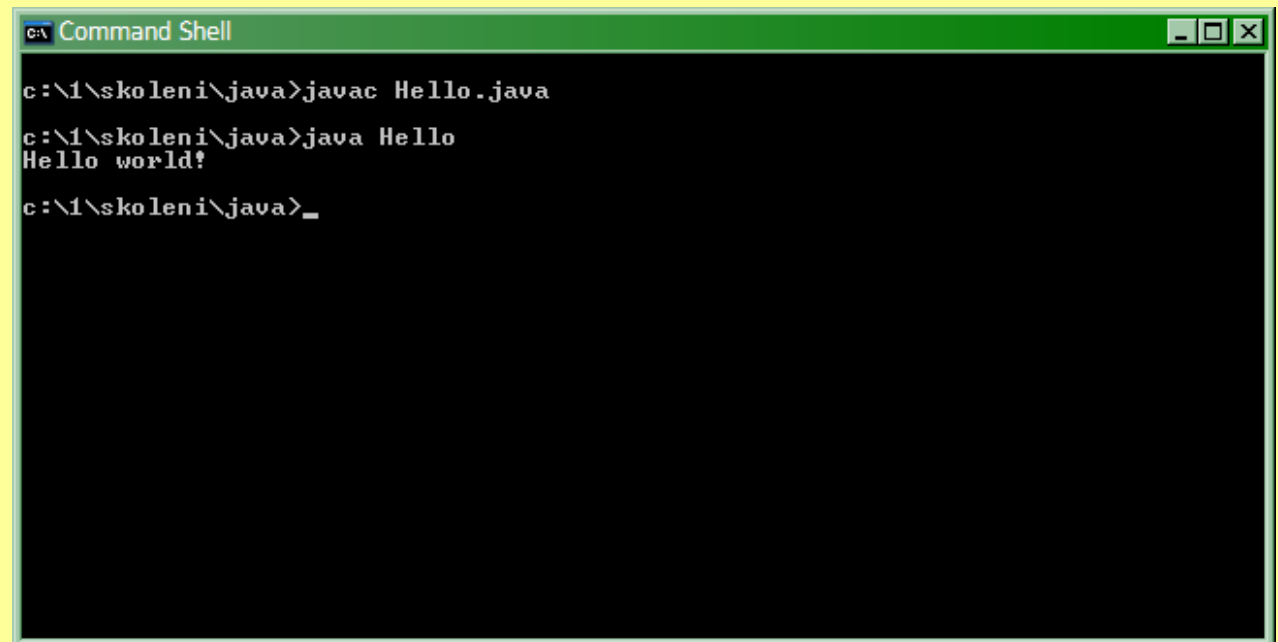
```
// jméno balíku, ve kterém je třída vytvořena
//package cz.iinfo.examples;

// "dovoz" tříd a rozhraní
import java.util.*;

// veřejná třída Hello
// - musí být umístěna v souboru Hello.java
public class Hello {
    // staticka verejna metoda main
    public static void main(String[] argv) {
        System.out.println("Hello world!");
    }
}
```

Překlad a spuštění programu

- `javac Hello.java`
- vytvoří se soubor `Hello.class` obsahující „bytekód“ určený pro JVM
- `java Hello`
- neuvádí se koncovka souboru
- Java nalezne statickou veřejnou metodu `main` a spustí ji



```
C:\ Command Shell

c:\1\skoleni\java>javac Hello.java
c:\1\skoleni\java>java Hello
Hello world!
c:\1\skoleni\java>_
```

Část 5

Proměnné, numerické a znakové hodnoty



Základní datové typy

- byte -128..127 8 bitů
- short -32768..32767 16 bitů
- int $-2^{31}..2^{31}-1$ 32 bitů
- minimum -2 147 483 648
- maximum 2 147 483 647
- long $-2^{63}..2^{63}-1$ 64 bitů
- minimum -9 223 372 036 854 775 808L
- maximum 9 223 372 036 854 775 807L
- char
- podpora Unicode
- boolean
- hodnoty true a false

Obalové třídy nad základními datovými typy (1)

- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Character` !!! `char`
- `java.lang.Boolean`

Obalové třídy nad základními datovými typy (2)

- Metody:
 - `***value()`
 - `parse***(text)`
 - `parse***(text, radix)`
 - Neexistuje pro všechny třídy
 - `toString(***)`
 - `valueOf(***)`
 - `valueOf(String)`
- Konstanty:
 - `MIN_VALUE`
 - `MAX_VALUE`
 - `SIZE`

Zjištění rozsahu celočíselných datových typů v runtime (1)

```
System.out.println(Byte.MIN_VALUE);  
-128
```

```
System.out.println(Byte.MAX_VALUE);  
127
```

```
System.out.println(Short.MIN_VALUE);  
-32768
```

```
System.out.println(Short.MAX_VALUE);  
32767
```

Zjištění rozsahu celočíselných datových typů v runtime (2)

```
System.out.println(Integer.MIN_VALUE);  
-2147483648
```

```
System.out.println(Integer.MAX_VALUE);  
2147483647
```

```
System.out.println(Long.MIN_VALUE);  
-9223372036854775808
```

```
System.out.println(Long.MAX_VALUE);  
9223372036854775807
```


Zjištění počtu bitů nutných pro reprezentaci hodnoty (1)

```
System.out.println(Byte.SIZE);
```

8

```
System.out.println(Short.SIZE);
```

16

```
System.out.println(Integer.SIZE);
```

32

```
System.out.println(Long.SIZE);
```

64

Zjištění počtu bitů nutných pro reprezentaci hodnoty (2)

```
System.out.println(Character.SIZE);  
16
```

Základní datové typy - FP

- float
 - podle normy IEEE 754
 - jednoduchá přesnost (32 bitů)
 - to odpovídá 6-7 číslicím
 - $\pm 3.40282347E+38$
- double
 - podle normy IEEE 754
 - dvojitá přesnost (64 bitů)
 - to odpovídá 15 číslicím
 - $\pm 1.79769313486231570E+308$
- poznámka:
 - při požadavku na vyšší přesnost/rozsah:
třídy BigInteger nebo BigDecimal

Speciální FP hodnoty

- `Float.POSITIVE_INFINITY`
 - Kladné nekonečno
- `Float.NEGATIVE_INFINITY`
 - Záporné nekonečno
- `Float.NaN`
 - Not a number (výsledek některých operací)
- `Double.POSITIVE_INFINITY`
 - -//-
- `Double.NEGATIVE_INFINITY`
 - -//-
- `Double.NaN`
 - -//-

Zjištění rozsahu FP datových typů v runtime

```
System.out.println(Float.MIN_VALUE);  
1.4E-45
```

```
System.out.println(Float.MAX_VALUE);  
3.4028235E38
```

```
System.out.println(Double.MIN_VALUE);  
4.9E-324
```

```
System.out.println(Double.MAX_VALUE);  
1.7976931348623157E308
```

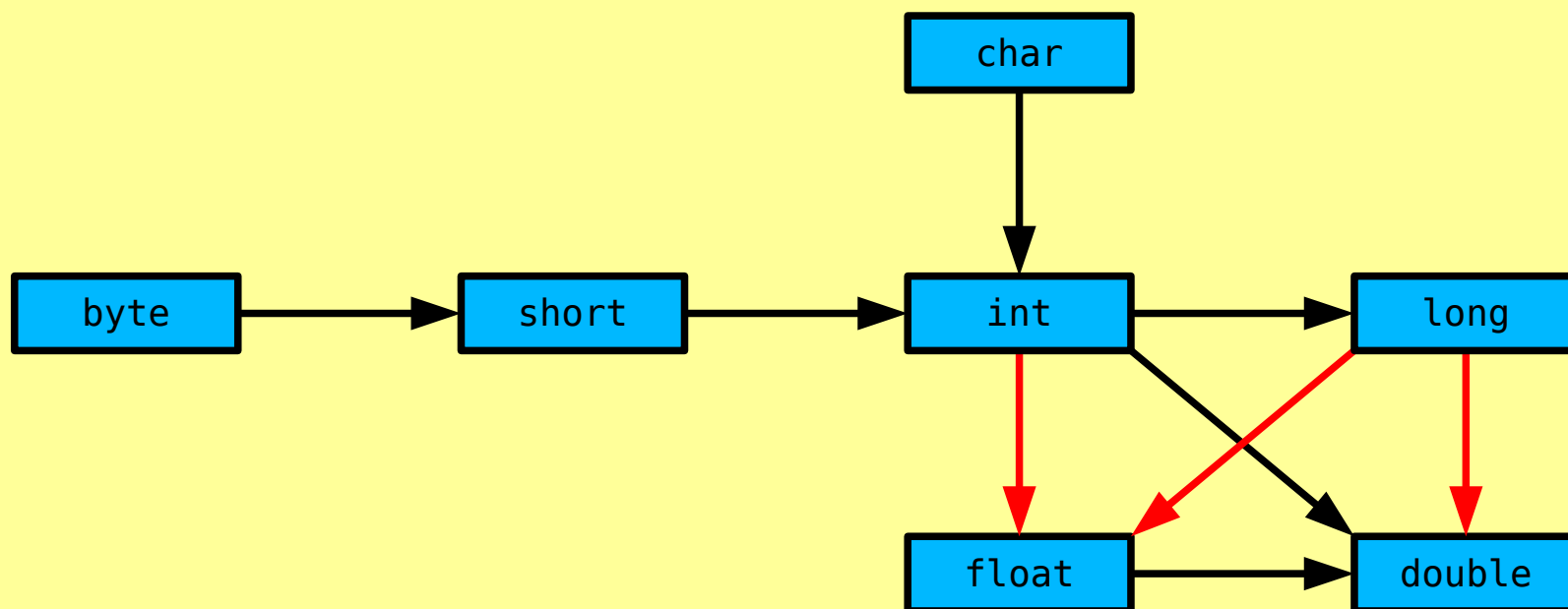
Číselné konstanty

- desítková soustava:
 - 123, 0, 5555, -100
- hexadecimální soustava:
 - 0xAA, 0xaa
- osmičková soustava:
 - 077, 0123
- "dlouhá" čísla (typ long):
 - 123456789L
- čísla s pohyblivou řádovou čárkou:
 - 123.456f (typ float)
 - .42
 - 123.456e-7

Znakové konstanty

- běžné znaky:
 - 'a', 'α' (16bitový Unicode)
- řídicí znaky:
 - '\n', '\t' ...
- znak s ASCII kódem 0:
 - '\0' odpovídá 0
- samotné zpětné lomítko:
 - '\\'
- samotný apostrof:
 - '\''
- zápis kódu znaku:
 - '\100'

Konverze mezi datovými typy



Řetězcové literály

- Běžný řetězec:
 - "Hello world!"
- Řetězec s řídicími znaky:
 - "Hello \t world!\n"
- Prázdný řetězec:
 - ""

Řetězce

- sekvence znaků (typ char)
- Java nemá primitivní datový typ řetězec
 - použití třídy String
 - přetížený operátor + (konkatenace)
 - ale už ne operátor ==
- v některých případech vhodnější instance jiných tříd
 - StringBuffer
 - StringBuilder

Část 6

Operátory a výrazy



Operátory a jejich priorita

Operátory

```
1  () [] . ++ -- (posfixové verze, za výrazem)
2  ++ -- (prefixové verze) + - (unární) ~ !
3  new (typ)
4  * / %
5  + -
6  << >> >>>
7  < <= > >= instanceof
8  == !=
9  &
10 ^
11 |
12 &&
13 ||
14 ? :
15 = += -= atd.
```

zprava doleva

Rozdělení operátorů do skupin

- Aritmetické operátory
 - + - * / %
 - ++ --
- Logické operátory
 - && || !
 - ? :
- Relační operátory
 - == != < <= > >=
- Bitové operátory
 - & | ^ ~
- Posuny na úrovni jednotlivých bitů
 - << >> >>>

Použití aritmetických operátorů

```
int x = 10;  
int y = 3;
```

System.out.println(x + y);	13
System.out.println(x - y);	7
System.out.println(x * y);	30
System.out.println(x / y);	3
System.out.println(x % y);	1
System.out.println(x >> y);	1
System.out.println(x << y);	80

Celočíselné dělení vs. FP dělení

```
System.out.println(10/3);  
3
```

```
System.out.println(10.0/3);  
3.3333333333333335
```

```
System.out.println(10/3f);  
3.3333333
```

```
System.out.println((double)10/3);  
3.3333333333333335
```

Bitové posuny (1)

```
int x = 10;  
int y = x << 1;  
int z = x >> 1;  
int w = x >>> 1;
```

```
System.out.println(Integer.toBinaryString(x));
```

1010

```
System.out.println(Integer.toBinaryString(y));
```

10100

```
System.out.println(Integer.toBinaryString(z));
```

101

```
System.out.println(Integer.toBinaryString(w));
```

101

Bitové posuny (2)

```
int x = -10;  
int y = x << 1;  
int z = x >> 1;  
int w = x >>> 1;
```

```
System.out.println(Integer.toBinaryString(x));  
1111111111111111111111111111111110110  
System.out.println(Integer.toBinaryString(y));  
11111111111111111111111111111111101100  
System.out.println(Integer.toBinaryString(z));  
1111111111111111111111111111111111111011  
System.out.println(Integer.toBinaryString(w));  
0111111111111111111111111111111111111011
```

Výrazy

- Výraz vs. příkaz:
 - `a=10*b;`
 - `10*b;`
 - `c=funkce()`
 - `funkce()`
 - `x++`
 - `x++;`
- Prefixové a postfixové operátory:
 - `++ a --`
 - `b=a++;`
 - `b=++a;`

Rozdíl mezi `a++` a `++a`

```
int x = 10;  
int y = 10;  
int z = x++;  
int w = ++y;
```

```
System.out.println("x=" + x); x=11  
System.out.println("y=" + y); y=11  
System.out.println("z=" + z); z=10  
System.out.println("w=" + w); w=11
```

Část 7

Podmíněné příkazy: if-else a switch



Podmíněný příkaz if-then

```
if (podmínka) příkaz;
```

```
if (podmínka) {  
    blok příkazů;  
    další příkazy;  
}
```

srovnej podmíněný výraz:

```
c=(a==0) ? 0 : b/a;
```

Podmíněný příkaz if-then-else

```
if (podmínka) příkaz;  
else příkaz2;
```

```
if (podmínka) {  
    blok příkazů;  
    další příkazy;  
}  
else {  
    blok příkazů;  
    další příkazy;  
}
```

Programová konstrukce typu switch

```
switch (celočíselný výraz) {  
    case konstanta1:  
        ....  
        break;  
    case konstanta2:  
    case konstanta3:  
        ....  
        break;  
    default:  
        break;  
}
```

Část 8

Tvorba smyček:
for, while, do-while a for-each



Smyčka typu `while`

```
while (podmínka) příkaz;
```

```
while (podmínka) {  
    blok příkazů;  
    další příkaz;  
}
```

```
while (podmínka)  
    ;
```

nevhodné (špatná čitelnost):

```
while (podmínka);
```

Smyčka typu do-while

```
do příkaz; while (podmínka);
```

```
do {  
    blok příkazů;  
    další příkaz;  
} while (podmínka);
```

Smyčka typu for

```
for (inic. výraz; podmínka; iter. příkaz)  
    příkaz;
```

```
for (inic. výraz; podmínka; iter. příkaz) {  
    tělo smyčky;  
    příkaz;  
}
```

```
for (inic. výraz; podmínka; iter. příkaz)  
    ;
```

```
for (i=0; i<100; i++)  
    System.out.println("radek"+i);
```

Smyčka typu **for** a datový typ float/double

```
for (double x = 0; x != 10; x += 0.1) . . .
```

nemusí nikdy skončit!

řešení:

- 1) použít operátor `<=`
- 2) použít konstrukci `Math.abs(x-0.1)<ε`
- 3) jiný datový typ (`BigDecimal`s)

Příkazy `break` a `continue`

```
while (true) {  
    a++;  
    if (a==100) break;  
    System.out.println("a="+a);  
}
```

```
while (true) {  
    a--;  
    if (a==10) continue;  
    System.out.println("a="+a);  
}
```

Příkaz **break** s návěstím

```
public class TestBreak {  
    public static void main(String[] args) {  
        konec:  
        for (int j=1; j<10; j++) {  
            for (int i=1; i<10; i++) {  
                System.out.print(i*j+"\t");  
                if (i==6 & j==7) break konec;  
            }  
            System.out.println();  
        }  
    }  
}
```

Smyčka typu for-each

- Konstrukce přidána až v Javě 1.5 (5.0)
- Použitelné pro kolekce a pole

```
int sum(int[] a) {  
    int result = 0;  
    for (int i : a)  
        result += i;  
    return result;  
}
```

Část 9

Pole



Pole

- Kontejner obsahující položky stejného typu (čísla, znaky, řetězce, objekty)
- Vytvoření reference na pole
 - `int[] a;`
 - `int b[];`
- Alokace položek
 - `int[] c=new int[10]`
- Přístup k položkám
 - `c[3]=10;`
 - `x=c[0]`
- Délka pole
 - `c.length`

Pole – jednoduchý příklad

```
public class TestPoli1 {  
    public static void main(String[] args) {  
        // vytvoření pole  
        int[] a=new int[10];  
  
        // naplnění položek hodnotami  
        for (int i=0; i<a.length; i++)  
            a[i]=i*i;  
  
        // tisk položek  
        for (int i=0; i<a.length; i++)  
            System.out.println(i+"\t"+a[i]);  
    }  
}
```

Inicializace polí

```
public class TestPoli2 {  
    public static void main(String[] args) {  
        // vytvoření pole  
        // s inicializací položek  
        int[] a={1, 2, 3, 5, 8, 13, 21};  
  
        // tisk položek  
        for (int i=0; i<a.length; i++)  
            System.out.println(i+"\t"+a[i]);  
    }  
}
```

Pole objektů

```
Color[] colors = new Color[10];
```

```
colors[0] = Color.RED;
```

```
colors[1] = Color.GREEN;
```

```
colors[5] = Color.BLUE;
```

```
colors[9] = new Color(1.0f, 0.5f, 0.0f);
```

```
for (Color color : colors) {  
    System.out.println(color);  
}
```

Pole objektů (pokračování)

```
java.awt.Color[r=255,g=0,b=0]  
java.awt.Color[r=0,g=255,b=0]  
null  
null  
null  
java.awt.Color[r=0,g=0,b=255]  
null  
null  
null  
java.awt.Color[r=255,g=128,b=0]
```

Pole objektů – neinicializované objekty

```
Color[] colors = new Color[10];

colors[0] = Color.RED;
colors[1] = Color.GREEN;
colors[5] = Color.BLUE;
colors[9] = new Color(1.0f, 0.5f, 0.0f);

for (Color color : colors) {
    System.out.println(color.toString());
}
```

Pole objektů – neinicializované objekty

```
java.awt.Color[r=255,g=0,b=0]  
java.awt.Color[r=0,g=255,b=0]  
Exception in thread "main"  
java.lang.NullPointerException  
    at Test.main(Test.java:14)
```

Vícerozměrná pole

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

for (int i = 0; i < pole.length; i++) {
    for (int j = 0; j < pole[i].length; j++) {
        pole[i][j] = (i+1) * (j+1);
    }
}

for (int i = 0; i < pole.length; i++) {
    for (int j = 0; j < pole[i].length; j++) {
        System.out.print(pole[i][j] + "\t");
    }
    System.out.println();
}
```


Vícerozměrná pole (Java 1.5)

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

for (int i = 0; i < pole.length; i++) {
    for (int j = 0; j < pole[i].length; j++) {
        pole[i][j] = (i+1) * (j+1);
    }
}

for (int[] vektor : pole) {
    for (int prvek : vektor) {
        System.out.print(prvek + "\t");
    }
    System.out.println();
}
```

Vícerozměrná pole - inicializace

```
int[][] pole = {  
    {1,2,3},  
    {4,5,6},  
    {7,8,9}  
};  
  
for (int[] vektor : pole) {  
    for (int prvek : vektor) {  
        System.out.print(prvek + "\t");  
    }  
    System.out.println();  
}
```

Část 10

Řetězce



Řetězce v Javě

- Nejedná se o primitivní datový typ – Java zná pouze řetězcový literál
- Z pohledu programátora se jedná o sekvenci znaků (typ **char** - Unicode)
- Třída String
 - konstantní (neměnné) řetězce
 - přetížený operátor **+** a **+=**
 - porovnávání řetězců pomocí **equals()** !!!
- v některých případech vhodnější instance jiných tříd (proměnné řetězce)
 - **StringBuffer**
 - **StringBuilder**

Vytvoření řetězce

```
public class StringTest1 {  
    public static void main(String[] args) {  
        // new je v tomto případě zbytečné  
        String s1=new String("Hello");  
        String s2=new String("world");  
        // použití přetíženého operátoru +  
        String s3=s1+" "+s2+'!';  
        System.out.println(s3);  
    }  
}
```

Vytvoření řetězce (2)

```
public class StringTest2 {  
    public static void main(String[] args) {  
        // bez new, pouze řetězcový literál  
        String s1="Hello";  
        String s2="world";  
        // použití přetíženého operátoru +  
        String s3=s1+" "+s2+'!';  
        System.out.println(s3);  
    }  
}
```

Metody třídy String (1)

- `charAt(index)`
 - vrátí znak na pozici „index“
- `compareTo(řetězec)`
 - lexikografické porovnání s dalším řetězcem
- `concat(řetězec)`
 - připojení dalšího řetězce na konec
- `contains(sekvence znaků)`
 - test, zda řetězec obsahuje sekvenci znaků
- `endsWith(podřetězec)`
 - test, zda řetězec končí zadaným podřetězcem
- `equals(řetězec či jiný objekt)`
 - test na rovnost dvou řetězců

Metody třídy String (2)

- `format(...)`
 - formátování dat a návrat výsledku
- `indexOf(znak)`
 - vrátí index prvního nalezeného znaku
- `indexOf(znak, index)`
 - vrátí index prvního nalezeného znaku od zadané pozice
- `indexOf(podřetězec)`
 - vrátí index prvního nalezeného podřetězce
- `indexOf(podřetězec, index)`
 - vrátí index prvního nalezeného podřetězce od zadané pozice

Metody třídy String (3)

- `lastIndexOf(znak)` a `lastIndexOf(řetězec)`
 - jako `indexOf()`, ale hledání od konce řetězce
- `length()`
 - vrátí délku řetězce
- `matches(regex)`
 - test, zda řetězec odpovídá regulárnímu výrazu
- `replace(znak1, znak2)`
 - náhrada znaku1 za znak2
- `startsWith(podřetězec)`
 - test, zda řetězec začíná zadaným podřetězcem
- `substring(index)`, `substring(index1, index2)`
 - vrácení podřetězce zadaného indexy

Metody třídy String (4)

- `toLowerCase()` a `toUpperCase()`
 - převod znaků na malá či velká písmena
- `trim()`
 - odstranění přebytečných mezer z řetězce
- `String valueOf(boolean)`
 - převod pravdivostní hodnoty na řetězec
- `String valueOf(char)`
 - převod znaku na řetězec
- `String valueOf(int)`
 - převod celého čísla na řetězec
 - + další převodní funkce pro ostatní primitivní datové typy

Konkatenace řetězců s využitím operátoru +

```
public class ConcatTest1 {  
    private static final int LOOP_COUNT = 10000;  
  
    private static String createString() {  
        String str = "";  
        for (int i = 0; i < LOOP_COUNT; i++) {  
            str += i + " ";  
        }  
        return str;  
    }  
  
    public static void main(String[] args) {  
        String str = createString();  
    }  
}
```

Konkatenace řetězců s využitím třídy StringBuffer

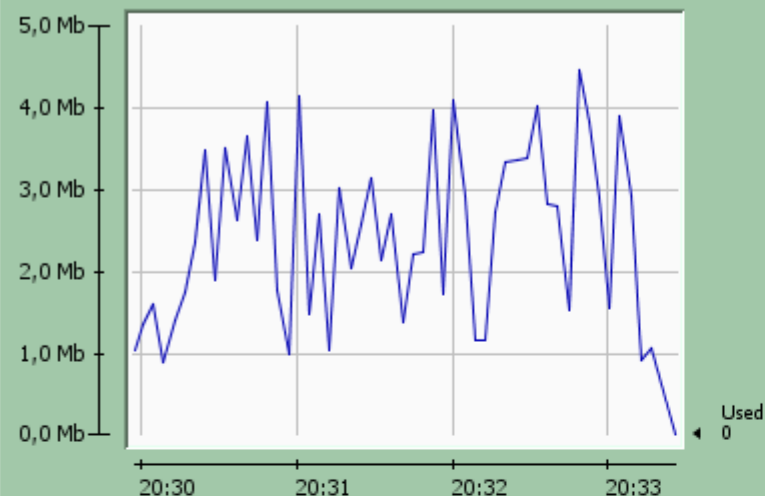
```
public class ConcatTest2 {  
    private static final int LOOP_COUNT = 10000;  
  
    private static String createString() {  
        StringBuffer str = new StringBuffer();  
        for (int i = 0; i < LOOP_COUNT; i++) {  
            str.append(i + " ");  
        }  
        return str.toString();  
    }  
  
    public static void main(String[] args) {  
        String str = createString();  
    }  
}
```

Konkatenace řetězců s využitím třídy StringBuffer*

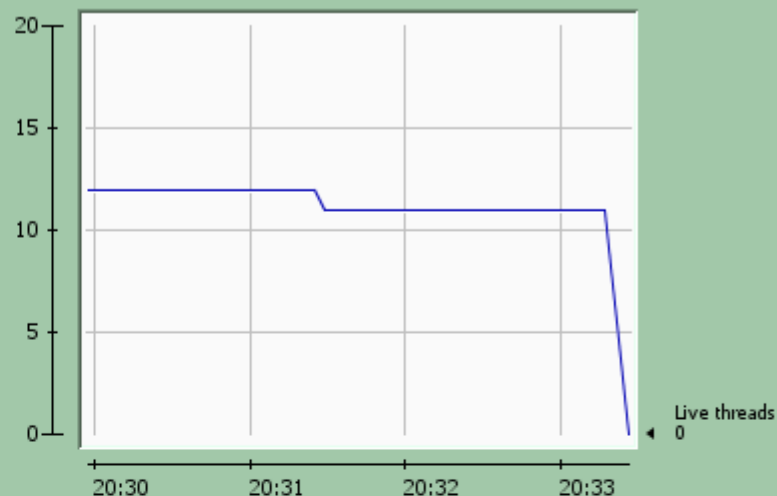
```
public class ConcatTest3 {  
    private static final int LOOP_COUNT = 10000;  
  
    private static String createString() {  
        StringBuffer str = new StringBuffer();  
        for (int i = 0; i < LOOP_COUNT; i++) {  
            str.append(i);  
            str.append(' ');  
        }  
        return str.toString();  
    }  
  
    public static void main(String[] args) {  
        String str = createString();  
    }  
}
```

Výsledek benchmarku: +

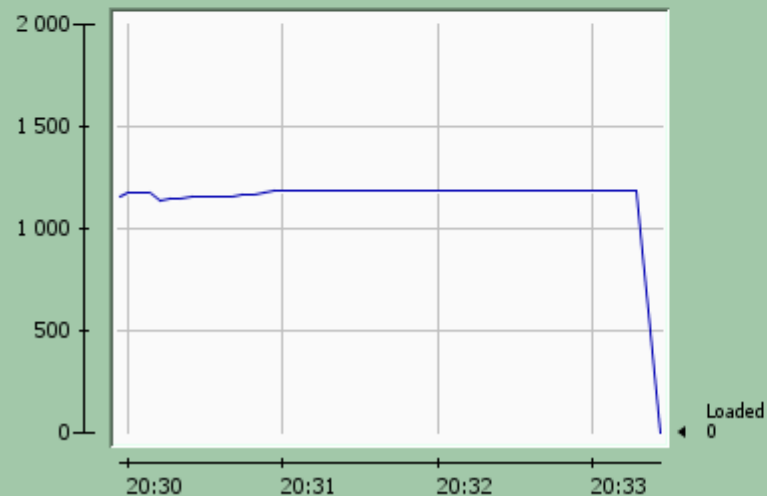
Heap Memory Usage



Threads



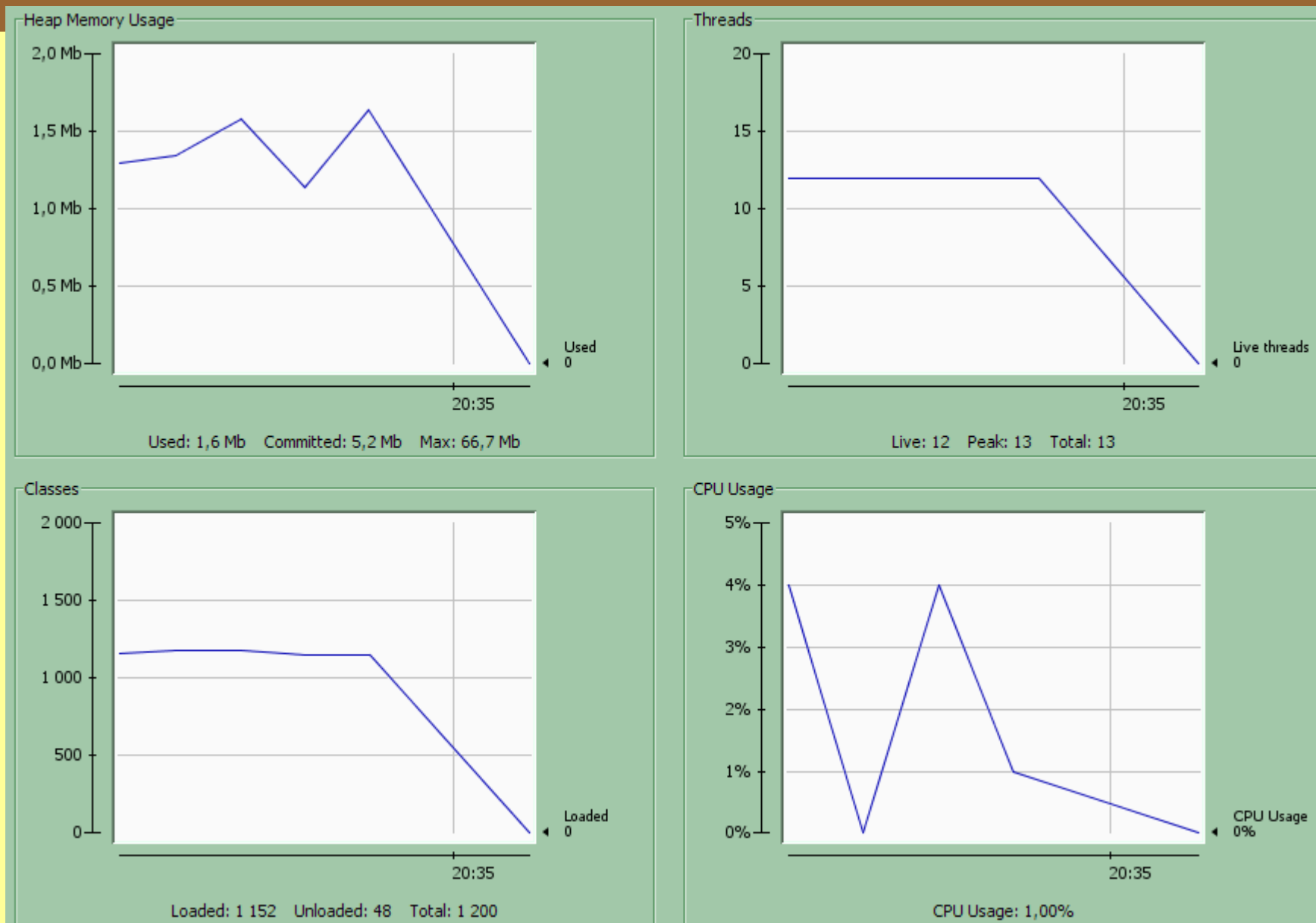
Classes



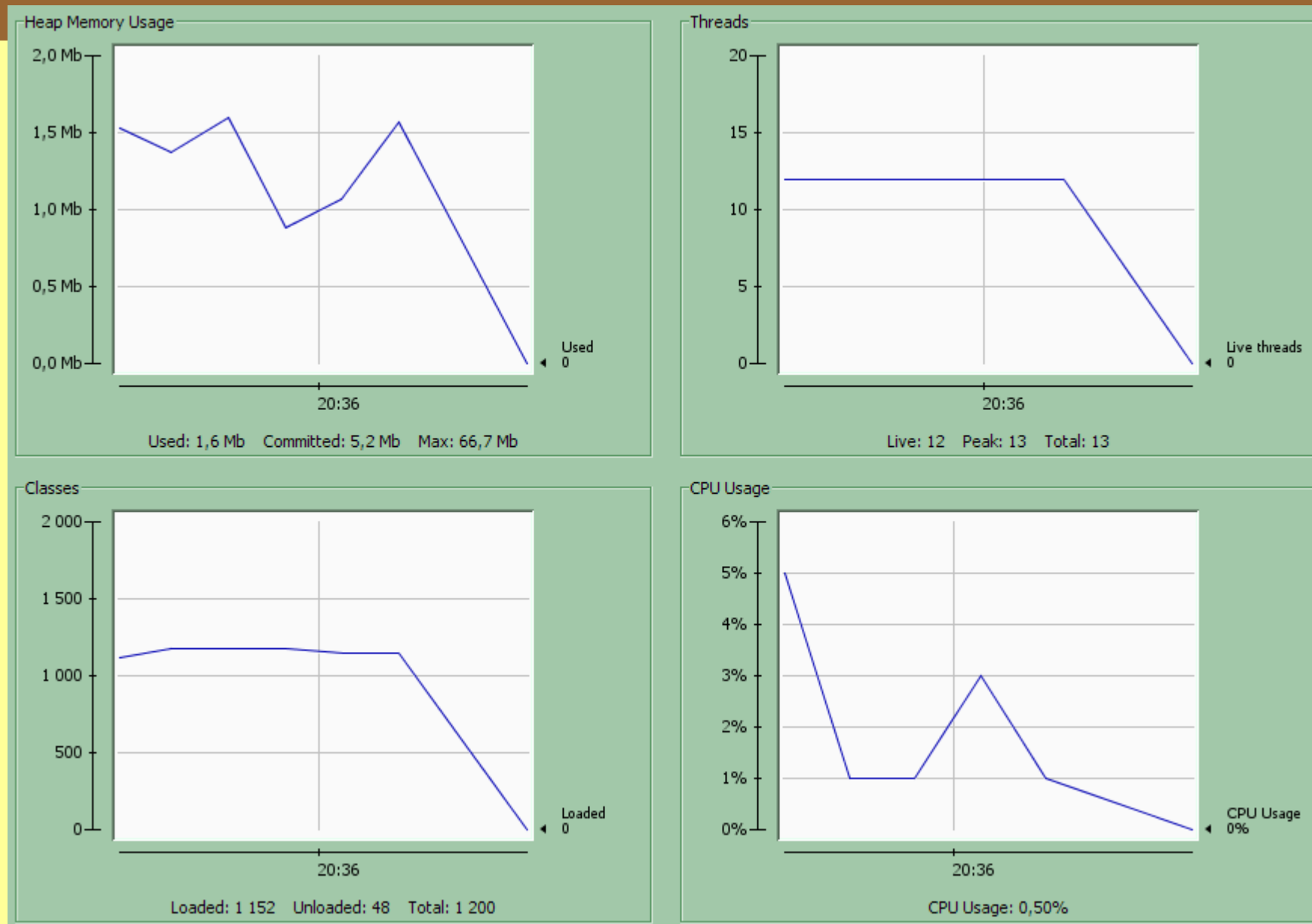
CPU Usage



Výsledek benchmarku: StringBuffer



Výsledek benchmarku: StringBuffer



Část 11

Základy objektově orientovaného
programování:
zapouzdření, dědičnost a polymorfismus



Objektově orientovaný přístup

- program=objekty+komunikace
- klasifikace
 - každý objekt patří do nějaké třídy
- skládání
 - hierarchie objektů
- dědičnost
 - hierarchie tříd

Vývoj programů

- OOA
 - objektově orientovaná analýza
- OOD
 - objektově orientovaný návrh
 - dekompozice
 - grafová notace (UML a spol.)
 - popis charakteristik systému
- OOP
 - objektově orientované programování
 - třídy v relaci dědičnosti
 - program je skupina spolupracujících objektů
 - objekt=instance třídy

Objektový model

- abstrakce
- zapouzdření
- modularita
- hierarchie

- dědičnost vs. skládání objektů
- polymorfismus
 - potomek může vždy nahradit předka
 - platné i pro abstraktní třídy

„has a“ versus „is a“

- Lod'
 - Plachetnice
 - Motorová lod'
 - Parník
 - Tanker
- Plachetnice **je** lod'
- Plachetnice **má** plachtu
- Parník **je** motorová lod'
- Parník **je** lod' (tranzitivita)
- Parník **má** motor

Dědičnost

- Lod'
 - Plachetnice
 - Motorová lod'
 - Parník
 - Tanker
- → společné vlastnosti (předek)
 - Plachetnice **má** barvu
 - Parník **má** barvu
- → specializace (potomci)
 - Počet plachet (integer)
 - Výkon motoru (kWh)
- → abstraktní „obecná“ lod' (abstraktní třída)

Abstraktní třídy

- Lod'
 - Plachetnice
 - Motorová lod'
 - Parník
 - Tanker
- → abstraktní „obecná“ lod' (abstraktní třída)
 - Žádná „obecná“ lod' ve skutečnosti neexistuje
 - Ovšem každá lod' má barvu, jméno, vlastníka
- Řešením je zavedení abstraktních tříd do jazyka
 - Nelze vytvořit jejich instance

Část 12

Třídy a objekty v Javě



Třídy

- třída=data+funkce+zapouzdření
- Typy tříd v Javě
 - veřejné/neveřejné
 - abstraktní
 - vnitřní (member)
 - lokální (v rámci metody)
 - anonymní

Třídy

- klíčové slovo class s udáním:
 - viditelnosti třídy
 - public nebo nic.
 - zda jde o třídu abstraktní či nikoli
 - abstract
 - modifikátor strictfp
 - modifikátor static – jen u vnitřních tříd
 - předka třídy
 - extends
 - pokud není uveden, tak je dosazeno Object
 - implementovaných rozhraní
 - implements

Datové položky - atributy

- Režim přístupu
 - `private`
 - `protected`
 - `public`
 - „`package protected`“
- Modifikátory
 - `final`
 - `strictfp`
 - `volatile`
 - `static`
- Název atributu
- (Inicializace atributu)

Režimy přístupu

Třída	stejná	jiná	extended	jiná
Balíček	shodný		odlišný	
=====				
private	ano	ne	ne	ne
„default“	ano	ano	ne	ne
protected	ano	ano	ano	ne
public	ano	ano	ano	ano

^

Modifikátor

„default“ ~ ~ ~ protected ale nelze používat ve třídě odvozené v jiném balíčku

Statické atributy a blok static

- Jedinečné pro třídu (a objekty = její instance)
- Blok static se vykoná po načtení třídy do virtuálního stroje
- `static { inicializace_atributů }`

Operátor new

- `ClassType var = new ClassType(parametry)`
- `AncestorClassType var = new ClassType(parametry)`
- `InterfaceType var = new ClassType(parametry)`
- + speciální syntaxe pro pole

Ukázka třídy s datovými položkami

```
class Complex {  
    public double re;  
    public double im;  
}  
  
public class test {  
    public static void main(String[] args) {  
        Complex c = new Complex();  
        c.re = 10.0;  
        c.im = 2.0;  
        System.out.println("c="+c.re+" "+c.im+"i");  
    }  
}
```

Metody a zapouzdření

```
public void setReal(double real) {  
    this.re = real;  
}
```

```
public double getReal() {  
    return this.re;  
}
```


Třída Complex se zapouzdřenými atributy

```
class Complex {  
    private double re;  
    private double im;  
    public void setReal(double real) {  
        this.re = real;  
    }  
    public void setImag(double imag) {  
        this.im = imag;  
    }  
    public double getReal() {  
        return this.re;  
    }  
    public double getImag() {  
        return this.im;  
    }  
}
```

Třída Complex se zapouzdřenými atributy

```
public class Test {  
    public static void main(String[] args) {  
        Complex c = new Complex();  
        c.setReal(10.0);  
        c.setImag(2.0);  
        System.out.println("c="+c.getReal()+" "+c.getImag()+"i");  
    }  
}
```

Další vylepšení třídy Complex

– metoda toString()

```
class Complex ...
```

```
...
```

```
    @Override
```

```
    public String toString() {
```

```
        return "c="+getReal()+" "+getImag()+"i";
```

```
    }
```

```
...
```

```
public class Test {
```

```
    public static void main(String[] args) {
```

```
        Complex c = new Complex();
```

```
        c.setReal(10.0);
```

```
        c.setImag(2.0);
```

```
        System.out.println(c);
```

```
    }
```

```
}
```

Konstruktory

```
class Complex {  
    ...  
    public Complex(double real, double imag) {  
        setReal(real);  
        setImag(imag);  
    }  
  
    ...  
  
public class Test {  
    public static void main(String[] args) {  
        Complex c = new Complex(10.0, 2.0);  
        System.out.println(c);  
    }  
}
```

Přetěžování konstruktorů

```
class Complex {  
    private double re;  
    private double im;  
  
    public Complex() {  
        this(0.0);  
    }  
  
    public Complex(double real) {  
        this(real, 0.0);  
    }  
  
    public Complex(double real, double imag) {  
        setReal(real);  
        setImag(imag);  
    }  
}
```

Hierarchie lodí – základní verze

```
class Lod {  
}
```

```
class Plachetnice extends Lod {  
}
```

```
class MotorovaLod extends Lod {  
}
```

```
class Parnik extends MotorovaLod {  
}
```

Hierarchie lodí – operátor instanceof

```
public class Test {  
    public static void main(String[] args) {  
        Lod          o1 = new Lod();  
        Plachetnice  o2 = new Plachetnice();  
        Parnik       o3 = new Parnik();  
        System.out.println(o1 instanceof Lod);  
        System.out.println(o1 instanceof MotorovaLod);  
        System.out.println(o1 instanceof Plachetnice);  
        System.out.println(o2 instanceof Lod);  
        System.out.println(o2 instanceof Plachetnice);  
        System.out.println(o3 instanceof Lod);  
        System.out.println(o3 instanceof MotorovaLod);  
    }  
}
```

Hierarchie lodí – společné vlastnosti (atributy+metody)

```
class Lod {  
    public Color color;  
    public String name;  
}  
  
class Plachetnice extends Lod {  
    public int pocetPlachet;  
}  
  
class MotorovaLod extends Lod {  
    public int vykonMotoru;  
}  
  
class Parnik extends MotorovaLod {  
}
```


Volání metod/přístup k atributům potomka

```
public class Test {  
    public static void main(String[] args) {  
        Lod o1 = new Plachetnice();  
        Lod o2 = new Parnik();  
        o1.color = Color.RED;  
        o2.color = Color.BLUE;  
        o1.pocetPlachet = 2;  
        o2.vykonMotoru = 3000;  
    }  
}
```

Volání metod/přístup k atributům potomka - překlad

```
Test.java:28: cannot find symbol  
symbol   : variable pocetPlachet  
location: class Lod
```

```
    o1.pocetPlachet = 2;  
      ^
```

```
Test.java:29: cannot find symbol  
symbol   : variable vykonMotoru  
location: class Lod
```

```
    o2.vykonMotoru = 3000;  
      ^
```

2 errors

Jedno z možných řešení

```
public class Test {  
    public static void main(String[] args) {  
        Lod o1 = new Plachetnice();  
        Lod o2 = new Parnik();  
        o1.color = Color.RED;  
        o2.color = Color.BLUE;  
        if (o1 instanceof Plachetnice)  
            ((Plachetnice)o1).pocetPlachet = 2;  
        if (o2 instanceof MotorovaLod)  
            ((MotorovaLod)o2).vykonMotoru = 3000;  
    }  
}
```

Základ OOP – problém v současné hierarchii

```
public class Test {  
    public static void main(String[] args) {  
        Lod o1 = new Lod();  
        Lod o2 = new MotorovaLod();  
        System.out.println(o1.getClass().getName());  
        System.out.println(o2.getClass().getName());  
    }  
}
```

Lod
MotorovaLod

Hierarchie lodí – abstraktní třídy

```
abstract class Lod {  
    public Color color;  
    public String name;  
}
```

```
class Plachetnice extends Lod {  
    public int pocetPlachet;  
}
```

```
abstract class MotorovaLod extends Lod {  
    public int vykonMotoru;  
}
```

```
class Parnik extends MotorovaLod {  
}
```

Hierarchie lodí – abstraktní třídy

```
pavel@bender:~/temp$ javac Test.java
```

```
Test.java:24: Lod is abstract; cannot be instantiated
```

```
    Lod o1 = new Lod();
```

^

```
Test.java:25: MotorovaLod is abstract; cannot be  
instantiated
```

```
    Lod o2 = new MotorovaLod();
```

^

2 errors

Část 13

Rozhraní



Rozhraní

- Definice chování tříd
 - Hlavičky metod
 - Konstanty
- Třída může **implementovat** dané rozhraní
- Třída **implementující** rozhraní se zavazuje, že bude implementovat i všechny **metody** v rozhraní definované
- V Java API jsou rozhraní využívána mnohem více, než abstraktní třídy
 - Důvod – neexistence vícenásobné dědičnosti

Rozhraní

```
interface Drawable {  
    public void draw();  
}
```

```
class Lod implements Drawable {  
    public Color color;  
    public String name;  
}
```

Rozhraní

```
Test.java:7: Lod is not abstract  
and does not override abstract  
method draw() in Drawable  
class Lod implements Drawable {  
^
```

1 error

Rozhraní se „propaguje“ na všechny potomky

```
interface Drawable {  
    public void draw();  
}
```

```
abstract class Lod implements Drawable  
{  
    public Color color;  
    public String name;  
}
```

Potomci musí rozhraní skutečně implementovat

Test.java:12: Plachetnice is not abstract and does not override abstract method draw() in Drawable

```
class Plachetnice extends Lod {
```

^

Test.java:20: Parnik is not abstract and does not override abstract method draw() in Drawable

```
class Parnik extends MotorovaLod {
```

^

Test.java:23: Tanker is not abstract and does not override abstract method draw() in Drawable

```
class Tanker extends MotorovaLod {
```

^

3 errors

Řešení (1.část)

```
interface Drawable {  
    public void draw();  
}  
  
abstract class Lod implements Drawable {  
    public Color color;  
    public String name;  
}  
  
class Plachetnice extends Lod {  
    public int pocetPlachet;  
    public void draw() {  
        System.out.println("Plachetnice s " +  
            pocetPlachet + " plachtami");  
    }  
}
```

Řešení (2.část)

```
abstract class MotorovaLod extends Lod {  
    public int vykonMotoru;  
    public void draw() {  
        System.out.println("Motorova lod s vykonem " +  
            vykonMotoru + " kW");  
    }  
}  
  
class Parnik extends MotorovaLod {  
}  
  
class Tanker extends MotorovaLod {  
}
```

Využití tříd implementujících rozhraní (1)

```
public class Test {  
    public static void main(String[] args) {  
        Lod o1 = new Plachetnice();  
        Lod o2 = new Parnik();  
        Lod o3 = new Tanker();  
        o1.draw();  
        o2.draw();  
        o3.draw();  
    }  
}
```

Plachetnice s 0 plachtami
Motorova lod s vykonem 0 kW
Motorova lod s vykonem 0 kW

Využití tříd implementujících rozhraní (2)

```
public class Test {  
    public static void main(String[] args) {  
        Lod[] konvoj = {  
            new Plachetnice(),  
            new Parnik(),  
            new Tanker(),  
        };  
        for (Lod lod : konvoj) {  
            lod.draw();  
        }  
    }  
}
```

Plachetnice s 0 plachtami
Motorova lod s vykonem 0 kW
Motorova lod s vykonem 0 kW

Část 14

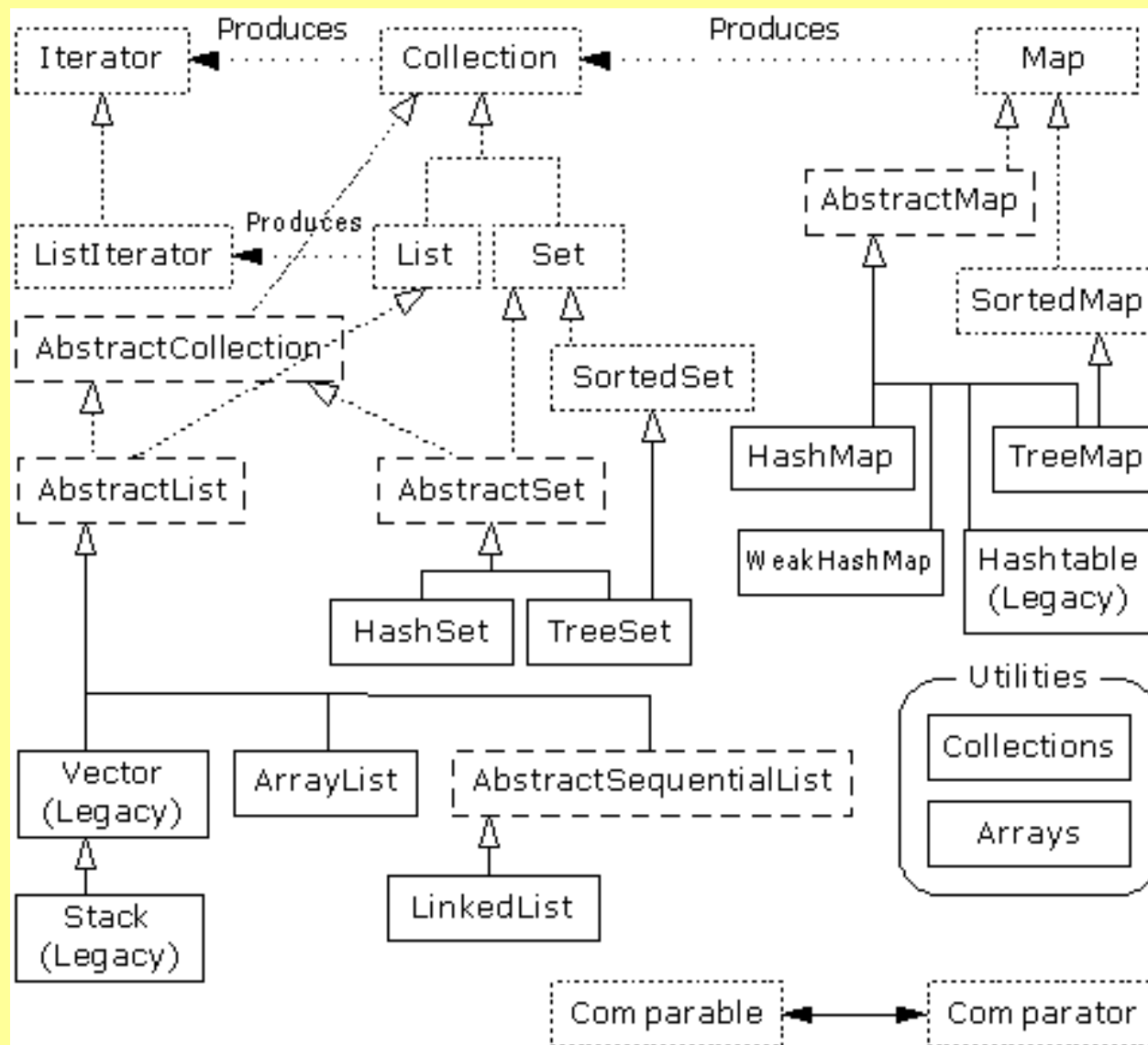
Kolekce



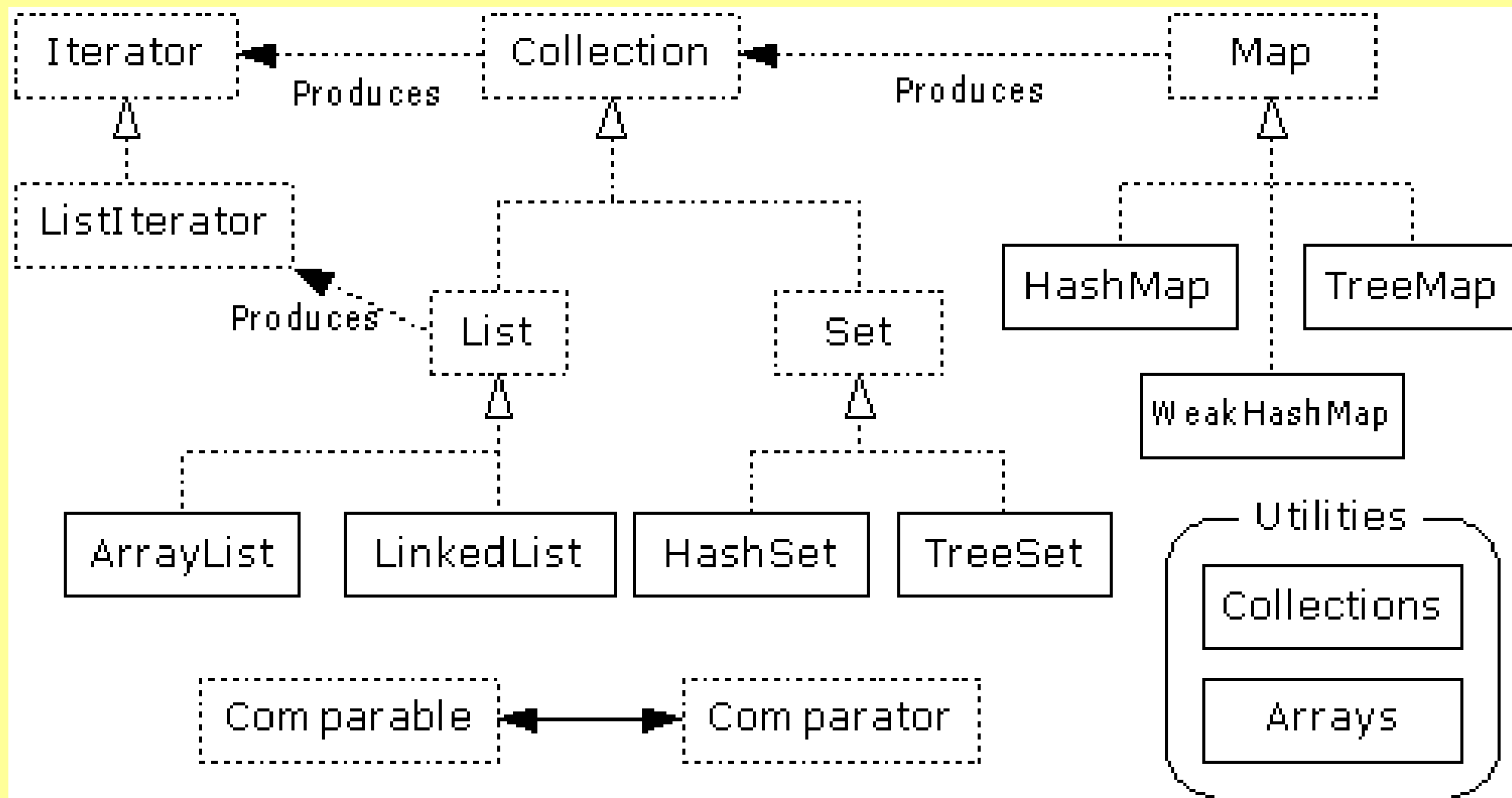
Kolekce

- Součást Java Collection Framework (JCF)
- Kontejnery pro ukládání objektový datových typů
 - Množiny
 - Seznamy
 - Mapy (neimplementuje Collection)
- Pomocné třídy a rozhraní
 - Iterátory
 - Komparátory
 - Převody pole ↔ kolekce
 - Řazení prvků v kolekcích

Diagram kolekcí v Javě 2



Co je skutečně třeba znát?



Rozhraní Collection

- Práce s prvky v kolekci
 - add(Object)
 - addAll(Collection)
 - remove(Object)
 - removeAll(Collection)
 - clear()
- Stav kolekce
 - contains(Object)
 - isEmpty()
 - size()
- Konverze
 - toArray()

Rozhraní Set

- Neobsahuje stejné elementy
 - → každý element je unikátní
- Pozor na změnu stavu objektů uložených v množině!
- Implementace rozhraní
 - HashSet
 - TreeSet
 - LinkedHashSet

Rozhraní List

- Ukládá prvky se zachováním jejich pořadí
- Možnost úschovy duplicitních prvků
- Nové metody v rozhraní:
 - `add(index, Object)`
 - `remove(index)`
 - `subList(from, to)`
- Implementace rozhraní
 - `ArrayList`
 - `LinkedList`

Rozhraní Map

- Asociativní pole, hashe
- Uchovává dvojici klíč-hodnota
- Klíč – unikátní
- Hodnota – libovolná
 - nemusí být unikátní
- Implementace rozhraní
 - HashMap
 - TreeMap

Metody rozhraní Map (1)

- Základní metody
 - `get(klíč)`
 - `put(klíč, hodnota)`
 - `containsKey(klíč)`
- Práce s prvky v mapě
 - `clear()`
 - `putAll(mapa)`
 - `remove(klíč)`

Metody rozhraní Map (2)

- Stav mapy
 - isEmpty()
 - size()
 - containsValue(hodnota)
- Konverze
 - entrySet()
 - keySet()
 - values()

Seznamy – praktický příklad (1)

```
public static void main(String[] args)
{
    // stejné rozhraní, rozdílné implementace
    List<String> seznam1 = new ArrayList<String>();
    List<String> seznam2 = new LinkedList<String>();

    testSeznamu(seznam1);
    testSeznamu(seznam2);
}
```

Seznamy – praktický příklad (2)

```
public static void testSeznamu(List<String> seznam)
{
    seznam.add("prvni");
    seznam.add("druhy");
    seznam.add("treti");

    for (String prvek : seznam) {
        System.out.println(prvek);
    }
}
```

Seznamy – praktický příklad (3)

```
public static void testSeznamu(List<String> seznam)
{
    System.out.println("Delka seznamu: " +
        seznam.size());

    System.out.println("Obsahuje 'prvni'?: " +
        seznam.contains("prvni"));

    System.out.println("Obsahuje 'xyzy'?: " +
        seznam.contains("xyzy"));

    seznam.remove(1);
}
```

Množiny – praktický příklad (1)

```
public static void main(String[] args)
{
    // stejné rozhraní, rozdílné implementace
    Set<Character> mnozina1 = new HashSet<Character>();
    Set<Character> mnozina2 = new TreeSet<Character>();

    testMnozin(mnozina1);
    testMnozin(mnozina2);
}
```

Množiny – praktický příklad (2)

```
public static void testMnozin(Set<Character> mnozina
) {
    mnozina.add('a');
    mnozina.add('z');
    mnozina.add('w');
    mnozina.add('x');
    mnozina.add('y');
    mnozina.add('j');
    mnozina.add('i');
    mnozina.add('b');
    mnozina.add('c');
    mnozina.add('d');
}
```

Množiny – praktický příklad (3)

```
public static void testMnozin(Set<Character> mnozina
) {
    // rozdíl v pořadí mezi HashSet a TreeSet!
    for (Character prvek : mnozina) {
        System.out.print(prvek);
        System.out.print(' ');
    }
}
```


Množiny – praktický příklad (4)

```
public static void testMnozin(Set<Character> mnozina
) {

    for (char ch = 'a'; ch <= 'z'; ch++) {
        System.out.print(ch);
        System.out.print(' ');
    }

    for (char ch = 'a'; ch <= 'z'; ch++) {
        System.out.print(mnozina.contains(ch)
            ? '^' : ' ');
        System.out.print(' ');
    }

}
```

Mapy – praktický příklad (1)

```
public static void main(String[] args)
{
    // stejné rozhraní, rozdílné implementace
    Map<String, Integer> mapa1 =
        new HashMap<String, Integer>();
    Map<String, Integer> mapa2 =
        new TreeMap<String, Integer>();

    testMap(mapa1);
    testMap(mapa2);
}
```

Mapy – praktický příklad (2)

```
public static void testMap(Map<String, Integer> mapa)
{
    mapa.put("one", 1);
    mapa.put("two", 2);
    mapa.put("three", 3);
    mapa.put("four", 4);
}
```

Mapy – praktický příklad (3)

```
public static void testMap(Map<String, Integer> mapa)
{
    Set<String> klice = mapa.keySet();
    for (String klic : klice) {
        System.out.println(
            klic + "\t -> " + mapa.get(klic));
    }

    for (String klic : mapa.keySet()) {
        System.out.println(
            klic + "\t -> " + mapa.get(klic));
    }
}
```

Mapy – praktický příklad (4)

```
public static void testMap(Map<String, Integer> mapa)
{
    for (Integer hodnota : mapa.values()) {
        System.out.println(hodnota);
    }
}
```

Mapy – praktický příklad (5)

```
public static void testMap(Map<String, Integer> mapa)
{
    for (Map.Entry<String, Integer> prvek :
        mapa.entrySet()) {
        System.out.println(
            prvek.getKey() + "\t -> " +
            prvek.getValue());
    }
}
```

Část 15

Standardní třídy dostupné v Javě



Java API

- Součást každé implementace J2SE, nezávisle na výrobci
- API může být kontrolováno testy kompatibility
- Java API obsahuje
 - Základní knihovny
 - Knihovny pro integraci Javy s okolním systémem
 - GUI toolkit

Základní knihovny

- java.lang
- java.util
- java.math
- Kolekce
- Logging
- Regulární výrazy
- Reflexe
- Beans
- Internationalization
- Networking

Integrace Javy s okolním systémem

- Scripting
 - JSR 223: Scripting for the Java Platform API
- RMI
 - Remote Method Invocation
- JNDI
 - Java Naming and Directory Interface
- JDBC
 - Java DataBase Connectivity
- IDL
 - Podpora CORBA (Common Object Request Broker Architecture)

GUI Toolkity

- AWT
 - Abstract Window Toolkit
- Swing
 - Součást Java Foundation Classes (JFC)
- Další balíčky používané v GUI
 - Java2D
 - Image I/O
 - Print service(s)
 - Sound
 - Accessibility
- (SWT)
 - Není součástí standardního Java API

java.lang (1)

- java.lang
 - V podstatě se jedná o nedílnou součást jazyka Java (i díky autoboxingu v Javě 1.5)
- java.lang.annotation
 - Podpora pro tvorbu a využití anotací
- java.lang.management
 - Monitorování a řízení vlastní JVM
- java.lang.reflect
 - Podpora reflexe - informací o třídách a objektech v čase běhu aplikace (runtime)

java.lang (2)

- Obalové datové typy
 - Number
 - Byte, Short, Integer, Long
 - Float, Double
 - BigDecimal, BigInteger
 - Boolean
 - Character
 - Void (nejde vytvořit instanci)
- Práce s řetězcí
 - String
 - StringBuffer
 - StringBuilder

java.lang (3)

- Object
 - Třída stojící na vrcholu třídní hierarchie
- System
 - Standardní vstupy a výstupy
 - Konzole
 - Properties...
- Thread
 - Zastavování a spouštění vláken, řízení priority, základní synchronizace
- Throwable
 - Třída (ne rozhraní!) představující předka pro chyby a výjimky

java.lang.Object

- Metody, které lze přepsat:
 - clone()
 - klonování objektu (možné použít i pro pole)
 - equals()
 - true/false, většinou nutné přepsat
 - finalize()
 - voláno GC při odstraňování objektu z paměti
 - hashCode()
 - dtto jako pro equals()
 - toString()
 - převod reprezentace objektu na řetězec

java.lang.System (1)

- Standardní vstupní a výstupní streamy
 - `PrintStream out`
 - `PrintStream err`
 - `InputStream in`
- Rozhraní pro konzoli
 - `Console console()`
- Systémové konfigurační parametry
 - `getenv()`
- Properties
 - `setProperties()`
 - `setProperty()`
 - `getProperty()`

java.lang.System (2)

- Různé metody související s JVM a systémem
 - gc()
 - exit(int status)
 - currentTimeMillis()
 - nanoTime()
- Podpora pro dynamické knihovny
 - load(filename)
 - loadLibrary(libraryName)

java.lang.System (3)

- Security manager
 - `getSecurityManager()`
 - `setSecurityManager()`
- Pomocí sec. manageru lze například zakázat čtení z určitých systémových vlastností, zakázat čtení či zápis do souborů atd.
 - viz též soubor
`/usr/lib/jvm/jre-1.6.0/lib/security/java.policy`

java.lang.reflect (1)

- Zajišťuje přístup k reflection API
- Každý prvek v Javě spadá do jedné z kategorií:
 - primitivní datový typ (boolean, int, ...)
 - referenční datový typ
- Referenční datový typ
 - třída (class)
 - pole (array)
 - výčet (enum)
 - rozhraní (interface)
- Reflection API je použitelné pro každý referenční datový typ

java.lang.reflect (2)

- Pro každý typ objektu je přímo v JVM vytvořena neměnná instance třídy `java.lang.Class`
- Přes tuto instanci je možné zkoumat všechny důležité vlastnosti třídy nebo objektu
- Základ:
 - `Class c1 = mujObjekt.getClass();`
 - `Class c2 = "foo".getClass();`
 - `Class c3 = System.out.getClass();`

java.lang.reflect (3)

- Metody třídy Class

- String getName()
- String getCaconicalName()
- Package getPackage()
- Annotation[] getAnnotations()
- Constructor[] getConstructors()
- Field[] getFields()
- Class[] getInterfaces()
- Method[] getMethods()
- Method getMethod(name, paramtypes)
- int getModifiers()

java.lang.reflect (4)

```
private void runAllTests(TestConfiguration config)
{
    Class c = this.getClass();
    Method[] methods = c.getDeclaredMethods();
    for (Method method : methods)
    {
        tryToInvokeTestMethod(config, method);
    }
}
```

java.lang.reflect (5)

```
private void
tryToInvokeTestMethod(TestConfiguration config,
Method method) {
    String methodName = method.getName();
    if (method.getName().startsWith("test")) {
        try {
            result = (TestResult)
                method.invoke(this, new Object[]
                    { param1, param2 });
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

java.lang.Math (1)

- Matematické funkce nejsou součástí jazyka Java, ale jsou definovány jako statické metody (== funkce) v této třídě
 - Konstanty
 - E
 - PI

java.lang.Math (2)

- java.lang.Math - funkce
 - abs, max, min
 - sin, cos, tan
 - sinh, cosh, tanh
 - asin, acos, atan, atan2(x,y)
 - pow, exp, log, log10
 - hypot = $\sqrt{x^2 + y^2}$
 - round, ceil, floor
 - toDegrees, toRadians

java.math.BigInteger (1)

- Celá čísla s libovolným rozsahem
- Obsahuje metody:
 - implementující základní aritmetické a logické operace
 - téměř všechny funkce z java.lang.Math
- Konstruktory
 - BigInteger(byte[] value)
 - BigInteger(String value)
 - BigInteger(String value, int radix)

java.math.BigInteger (2)

- Metody
 - Aritmetické operace
 - add, subtract, multiply, divide, divideAndRemainder, negate
 - Logické operace
 - and, andNot, or, not(), xor
 - Bitové operace
 - clearBit, flipBit, setBit, testBit
 - Bitové posuny
 - shiftLeft, shiftRight
 - Konverze
 - floatValue, doubleValue, intValue, longValue...

java.math.BigInteger (3)

```
import java.math.BigInteger;

public class BigIntegerTest {
    public static String factorial(int n) {
        if (n <= 1)
            return "1";
        BigInteger result = BigInteger.ONE;
        for (int i = 2; i <= n; i++)
            result = result.multiply(
                BigInteger.valueOf(i));
        return result.toString();
    }
    public static void main(String[] args) {
        for (int n = 1; n < 100; n++)
            System.out.println(n + "\t" + factorial(n));
    }
}
```

java.math.BigInteger (4)

1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

99

93326215443944152681699238856266700490715968264381621468
59296389521759999322991560894146397615651828625369792082
7223758251185210916864000000000000000000000000

java.math.BigDecimal (1)

- Reálná čísla s libovolným rozsahem (range) i přesností (accuracy)
- Konstruktory
 - `BigDecimal(BigInteger value)`
 - `BigDecimal(BigInteger value, int scale)`
 - `BigDecimal(int)`
 - `BigDecimal(double)`
 - `BigDecimal(String value)`

java.math.BigDecimal (2)

```
import java.math.BigDecimal;

public class BigDecimalTest {
    public static void main(String[] args) {
        double d = 0.1;
        BigDecimal bigDec = new BigDecimal(d);
        System.out.println(bigDec.toString());
    }
}

???
```

java.math.BigDecimal (3)

[illegible]

java.io (1)

- Všechny vstupně/výstupní funkce:
 - soubory
 - roury (pipes)
 - síťové spojení
 - paměťové buffery
 - buffering
 - parsing
 - ...

java.io (2)

- Rozdělení tříd v balíčku java.io
 - Podle typu I/O
 - soubory
 - paměťové buffery
 - pole
 - ...
 - Podle základní informační jednotky
 - bajtově orientované
 - znakově orientované
 - Podle směru přenosu
 - vstupní
 - výstupní

java.io (3)

- Abstraktní I/O třídy
 - Bajtově orientované
 - InputStream
 - OutputStream
 - Znakově orientované
 - Reader
 - Writer
 - InputStreamReader
 - OutputStreamWriter

java.io (4)

- Souborové I/O třídy
 - Bajtově orientované
 - FileInputStream
 - FileOutputStream
 - RandomAccessFile
 - Znakově orientované
 - FileReader
 - FileWriter

java.io (5)

- I/O třídy pro práci s objekty
 - Bajtově orientované
 - `ObjectInputStream`
 - `ObjectOutputStream`
 - Znakově orientované
 - neexistují

java.io (6)

- I/O třídy pro práci s řetězcí
 - Bajtově orientované
 - neexistují
 - Znakově orientované
 - StringReader
 - StringWriter

java.io (7)

- I/O třídy pro práci s poli
 - Bajtově orientované
 - ByteArrayInputStream
 - ByteArrayOutputStream
 - Znakově orientované
 - CharArrayReader
 - CharArrayWriter

Demonstrační příklad #1

```
import java.io.*;

public class IOTest1
{
    public static void main(String[] args) {
        File file = new File("src/IOTest1.java");
        System.out.println(file.getAbsolutePath());
        System.out.println(file.getPath());
        System.out.println(file.getAbsolutePath());
        System.out.println("exists:    " + file.exists());
        System.out.println("can read:  " + file.canRead());
        System.out.println("can write:" + file.canWrite());
        System.out.println("is file:   " + file.isFile());
        System.out.println("is dir:   " + file.isDirectory());
    }
}
```


Regulární výrazy

- Balíček
 - `java.util.regex`
- Třídy v balíčku
 - `Pattern`
 - zkompilovaná podoba regulárního výrazu
 - `Matcher`
 - provádí porovnání řetězce oproti regulárnímu výrazu
 - `PatternSyntaxException`
 - chybná syntaxe regulárního výrazu

Test na použití regulárních výrazů

```
import java.util.regex.*;

public class RegexTest {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("[a-z]+");
        Matcher matcher = pattern.matcher(
            "foo 123 XYZ 999 aaaBBBccc");
        while (matcher.find()) {
            System.out.format(
                "start: %3d end: %3d text: %s\n",
                matcher.start(),
                matcher.end(),
                matcher.group());
        }
    }
}
```

Část 16

Zachytávání výjimek



Důvody vedoucí k zavedení výjimek (1)

- Při běhu metody může dojít k různým chybovým (nebo jen neobvyklým) stavům, které je zapotřebí zpracovat
- Řešení v procedurálních jazycích
 - návratové kódy s číslem chyby
 - C standard library
 - nastavení chybové proměnné
 - v C globální proměnná `errno`
 - nověji se jedná o makro kvůli možnosti běhu vícevláknových aplikací

Důvody vedoucí k zavedení výjimek (2)

- Nevýhody využití návratových kódů
 - vlastní algoritmus je doplněn o množství nových podmínek
 - některé funkce musí vracet skutečné návratové hodnoty nepřímo - referencí
- Nevýhody využití proměnné **errno**
 - změna hodnoty proměnné při každém volání systémové funkce
 - taktéž vede k nutnosti použití množství nových podmínek v programu

Demonstrační příklad #2

```
import java.io.*;

public class IOTest2
{
    public static void main(String[] args) {
        BufferedReader bufferedReader = null;
        try {
            File file = new File("src/IOTest2.java");
            InputStream input = new FileInputStream(file);
            Reader reader = new InputStreamReader(input);
            bufferedReader = new BufferedReader(reader);
            String line;
            while ((line=bufferedReader.readLine()))!=null) {
                System.out.println(line);
            }
        }
        ...
    }
}
```

Demonstrační příklad #2

```
...  
    catch (FileNotFoundException e) {  
        e.printStackTrace();  
    }  
    catch (IOException e)  
    {  
        e.printStackTrace();  
    }  
...
```

Demonstrační příklad #2

...

```
finally {  
    if (bufferedReader != null) {  
        try {  
            bufferedReader.close();  
        }  
        catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}  
  
}  
  
}
```


Demonstrační příklad #3

```
try {  
    BufferedReader bufferedReader =  
        new BufferedReader(  
            new InputStreamReader(  
                new FileInputStream("src/IOTest3.java")  
            )  
        );  
    String line;  
    while ((line = bufferedReader.readLine()) != null) {  
        System.out.println(line);  
    }  
}
```

Demonstrační příklad #4

```
import java.io.*;

public class Hello
{
    public static void main(String[] args) {
        OutputStream out = null;
        try {
            out = new FileOutputStream("data.bin");
            for (int i = 'a'; i <= 'z'; i++) {
                out.write(i);
            }
            out.write(0);
            out.write(255);
            out.flush();
        }
    }
}
```

...

Demonstrační příklad #4

...

```
catch (FileNotFoundException e) {  
    e.printStackTrace();  
}  
catch (IOException e) {  
    e.printStackTrace();  
}
```

...

Demonstrační příklad #4

...

```
finally {  
    try {  
        if (out != null) {  
            out.close();  
        }  
    }  
    catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

```
}
```

```
}
```

Čtení dat z web serveru (1)

```
import java.io.*;
import java.net.*;

public class Hello
{
    public static void main(String[] args)
    {
        String address =
            "http://torment.usersys.redhat.com/openjdk/test.txt";
        URL url = null;
        BufferedReader bufReader = null;

        ...
    }
}
```

Čtení dat z web serveru (2)

```
...
try {
    url = new URL(address);
    try {
        bufReader = new BufferedReader(
            new InputStreamReader(url.openStream()));
        String line;
        while ((line = bufReader.readLine()) != null) {
            System.out.println(line);
        }
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
...
```

Čtení dat z web serveru (3)

```
...  
    catch (MalformedURLException e) {  
        e.printStackTrace();  
    }  
    try {  
        if (bufReader != null) {  
            bufReader.close();  
        }  
    }  
    catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
...
```

Část 17

Vlákna



Vlákna (threads) v Javě

- Multithreading se stává důležitou součástí aplikací
 - Programy s GUI
 - Moderní multiprocesorové systémy
 - I levné CPU mají více jader
- Podpora pro práci s vlákny přímo v syntaxi a sémantice jazyka Java
- Změna stavu vlákna
 - spuštění, zastavení
 - synchronizace
 - Nastavení priority po spuštění vlákna

Vlákna (threads) v Javě

- Dvě možnosti vytvoření třídy implementující vlákno
 - extends Thread
 - implements Runnable
- Spuštění vlákna
 - pomocí metody start()
 - interně se vytvoří nové vlákno a spustí se metoda run()
 - **nespouštět explicitně metodu run() !!!**
 - v tomto případě by se metoda nespustila v novém vlákně

Vlákna v Javě - demonstrační příklady

1. Rozšíření třídy Thread
2. Implementace rozhraní Runnable
3. Změna priority běžících vláken
4. Přístup ke sdíleným prostředkům
5. Synchronizace vláken
6. Deadlock

Příklad číslo 1 - rozšíření třídy Thread

Příklad číslo 1 - rozšíření třídy Thread

- Je vytvořena nová třída **VlaknoTyp1**
- Tato třída rozšiřuje třídu **Thread**
- V metodě **run()** je spuštěna programová smyčka, která vypíše hodnotu počítadla a ID vlákna
- Samotné vlákno je spuštěno zavoláním metody **start()**, nikoli **run()**

Rozšíření třídy Thread - deklarace a konstruktor

```
class VlaknoTyp1 extends Thread {  
    private int id;  
    private int counter = 5;  
  
    public VlaknoTyp1(int id) {  
        this.id = id;  
        System.out.println(  
            "vlakno #" + id + " vytvoreno");  
    }  
}
```

...

Rozšíření třídy Thread - metoda run()

```
@Override
public void run() {
    while (this.counter > 0) {
        try {
            System.out.println("vlakno #" +
                               this.id +
                               " pocitadlo: " +
                               this.counter);
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.counter--;
    }
}
```

Rozšíření třídy Thread - spuštění nových vláken

```
public class ThreadTest1 {  
    public static void main(String[] args) {  
        for (int id = 0; id < 10; id++) {  
            new VlaknoTyp1(id).start();  
        }  
    }  
}
```


Příklad číslo 2 - implementace rozhraní Runnable

Příklad číslo 2 - implementace rozhraní Runnable

- Je vytvořena nová třída `VlaknoTyp2`
- Tato třída implementuje rozhraní `Runnable`
- V metodě `run()` je spuštěna programová smyčka, která vypíše hodnotu počítadla a ID vlákna
- Z třídy `VlaknoTyp2` je vytvořeno vlastní vlákno pomocí `new Thread(VlaknoTyp2)`
- Samotné vlákno je spuštěno zavoláním metody `start()`, nikoli `run()`

Implementace rozhraní Runnable

- deklarace a konstruktor

```
class VlaknoTyp2 implements Runnable {  
    private int id;  
    private int counter = 5;  
  
    public VlaknoTyp2(int id) {  
        this.id = id;  
        System.out.println(  
            "vlakno #" + id + " vytvoreno");  
    }  
    ...  
}
```

Implementace rozhraní Runnable - metoda run

```
@Override
public void run() {
    while (this.counter > 0) {
        try {
            System.out.println("vlakno #" + this.id
+ " pocitadlo: " + this.counter);
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.counter--;
    }
}
```

Implementace rozhraní Runnable

- spuštění nových vláken (v.1)

```
public class ThreadTest2 {  
    public static void main(String[] args) {  
        for (int id = 0; id < 10; id++) {  
            // rozdíl oproti předchozímu příkladu  
            VlaknoTyp2 p = new VlaknoTyp2(id);  
            Thread t = new Thread(p);  
            t.start();  
        }  
    }  
}
```

Implementace rozhraní Runnable

- spuštění nových vláken (v.2)

```
public class ThreadTest2 {  
    public static void main(String[] args) {  
        for (int id = 0; id < 10; id++) {  
            // rozdíl oproti předchozímu příkladu  
            new Thread(new VlaknoTyp2(id)).start();  
        }  
    }  
}
```

Příklad číslo 3 - změna priority běžících vláken

Příklad číslo 3 - změna priority běžících vláken

- Jedná se z velké části o kopii příkladu číslo 1
- Jediný rozdíl spočívá v tom, že se prvnímu vláknu sníží priorita na minimum:
 - `v.setPriority(Thread.MIN_PRIORITY);`
- Pátému vláknu se naopak zvýší priorita na maximum:
 - `v.setPriority(Thread.MAX_PRIORITY);`

Rozšíření třídy Thread - deklarace a konstruktor

```
class VlaknoTyp3 extends Thread {  
    private int id;  
    private int counter = 5;  
  
    public VlaknoTyp3(int id) {  
        this.id = id;  
        System.out.println(  
            "vlakno #" + id + " vytvoreno");  
    }  
}
```

Rozšíření třídy Thread - metoda run()

```
@Override
public void run() {
    while (this.counter > 0) {
        try {
            System.out.println("vlakno #" +
                               this.id +
                               " pocitadlo: " +
                               this.counter);
            Thread.sleep(100);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.counter--;
    }
}
```

Vytvoření nových vláken a nastavení jejich priority

```
public class ThreadTest3 {  
    public static void main(String[] args) {  
        for (int id = 0; id < 10; id++) {  
            VlaknoTyp3 v = new VlaknoTyp3(id);  
            if (id == 0) {  
                v.setPriority(Thread.MIN_PRIORITY);  
            }  
            if (id == 5) {  
                v.setPriority(Thread.MAX_PRIORITY);  
            }  
            v.start();  
        }  
    }  
}
```

Příklad číslo 4 - přístup ke sdíleným prostředkům

Příklad číslo 4 - přístup ke sdíleným prostředkům

- Ve třídě **Vlakno** je vytvořena statická proměnná **sharedVariable=0**
- K této proměnné asynchronně přistupují jednotlivá běžící vlákna:
 - provádí její inkrementaci
 - ihned poté dekrementaci
- Vzhledem k tomu, že **chybí synchronizace**, může nastat situace, kdy je hodnota proměnné viditelná jiným vláknům rozdílná od nuly!

Třída se sdílenou proměnnou

```
class Vlakno extends Thread {  
    public static int sharedVariable = 0;  
  
    private void increment()  
    {  
        sharedVariable++;  
    }  
  
    private void decrement()  
    {  
        sharedVariable--;  
    }  
}
```

Metoda run() bez synchronizace

```
@Override
public void run() {
    for (int i = 0; i < 1000; i++) {
        increment(); // není synchronizace
        decrement(); // dtto
        if (sharedVariable != 0) {
            System.out.println(
"!synchronizace: " + this.getName() + " #" + i);
        }
    }
}
```

Spuštění více vláken paralelně

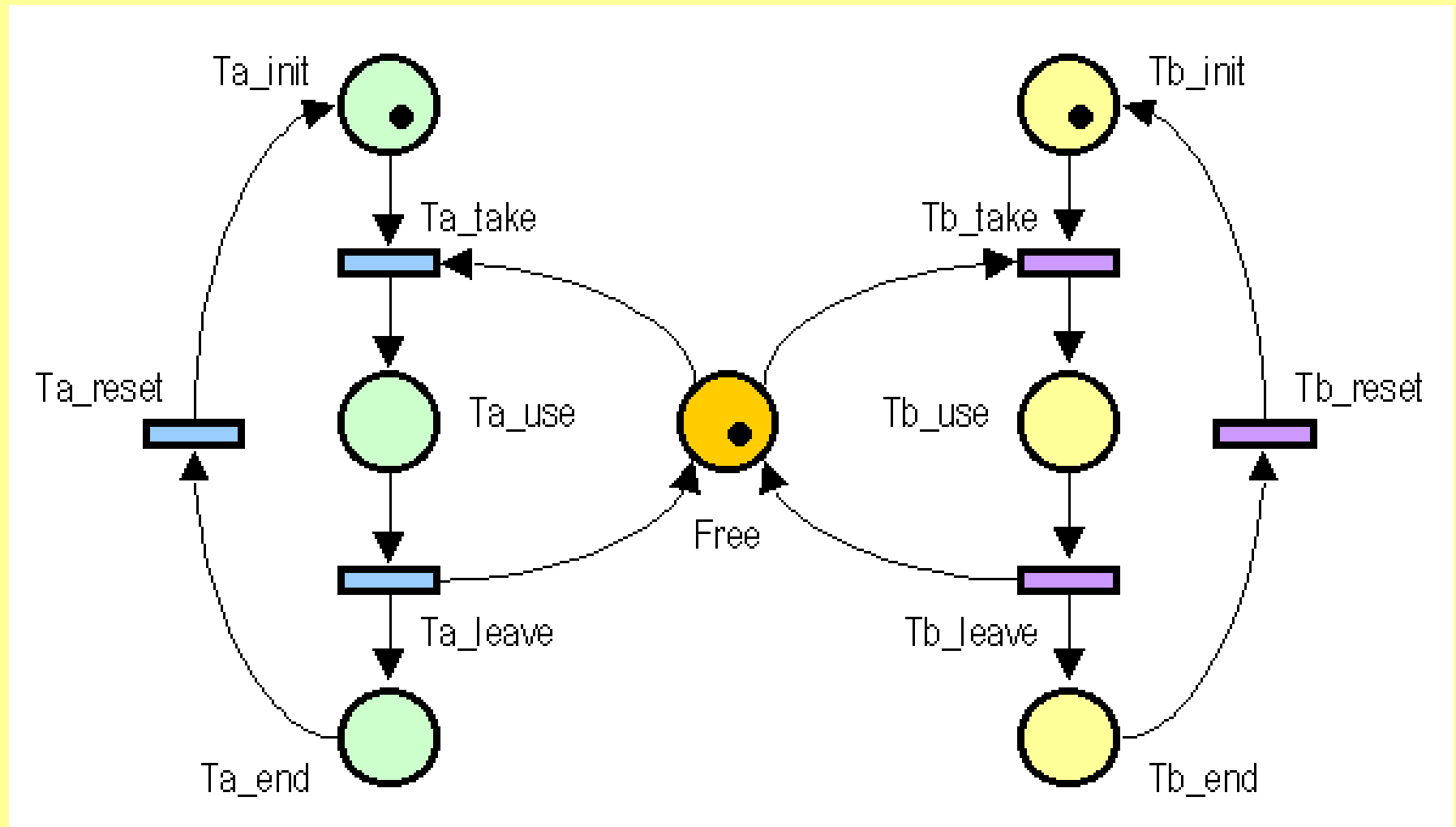
```
public class SoubehVlaken
{
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Vlakno().start();
        }
    }
}
```


Příklad číslo 5 - synchronizace vláken

Příklad číslo 5 - synchronizace vláken

- Prakticky stejný zdrojový kód, jako v případě příkladu číslo 4
- Jediná důležitá změna
 - tělo metody run() je synchronizováno
 - runtime Javy zajistí, že pouze jediné vlákno vstoupí do synchronizované části
 - synchronizace se provádí nad nějakým objektem sloužícím jako zámek

Dvě vlákna a zámek vymodelovaný pomocí Petriho sítě



Třída se sdílenou proměnnou a objektem pro synchronizaci

```
class Vlakno extends Thread {  
    private static final String syncObject = "";  
    public static int sharedVariable = 0;  
  
    private void increment()  
    {  
        sharedVariable++;  
    }  
  
    private void decrement()  
    {  
        sharedVariable--;  
    }  
}
```

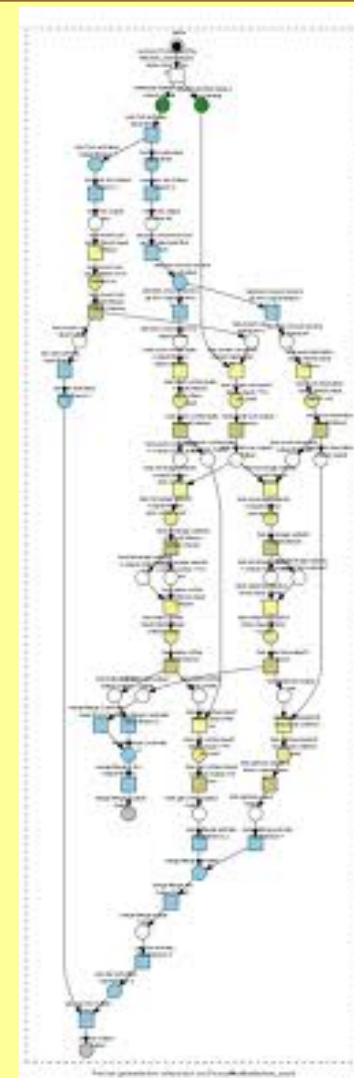
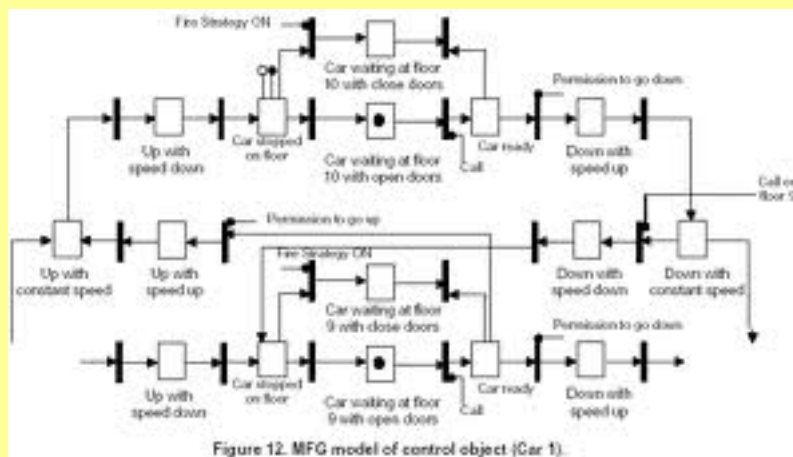
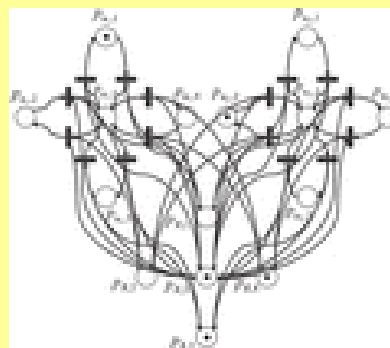
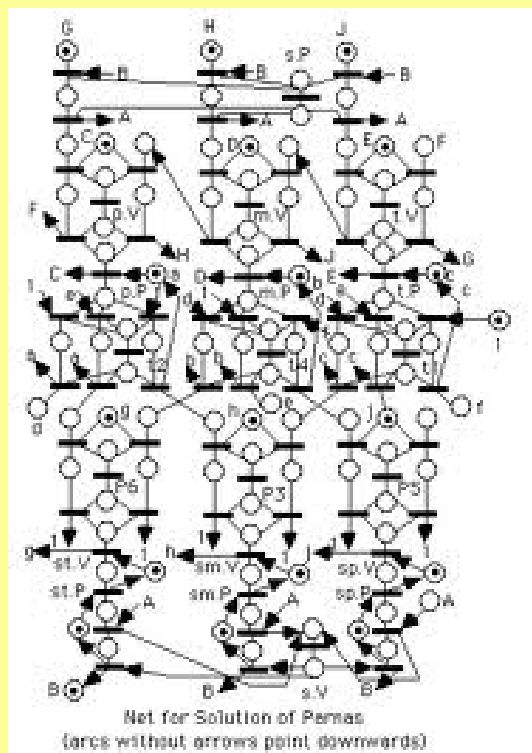
Metoda run() se synchronizací

```
@Override
public void run() {
    for (int i = 0; i < 1000; i++) {
        synchronized(syncObject) {
            increment(); // není synchronizace
            decrement(); // dtto
            if (sharedVariable != 0) {
                System.out.println(
"!synchronizace: " + this.getName() + " #" + i);
            }
        }
    }
}
```

Spuštění více vláken paralelně

```
public class SoubehVlaken
{
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            new Vlakno().start();
        }
    }
}
```

Synchronizace paralelních systémů je obecně velmi složitá



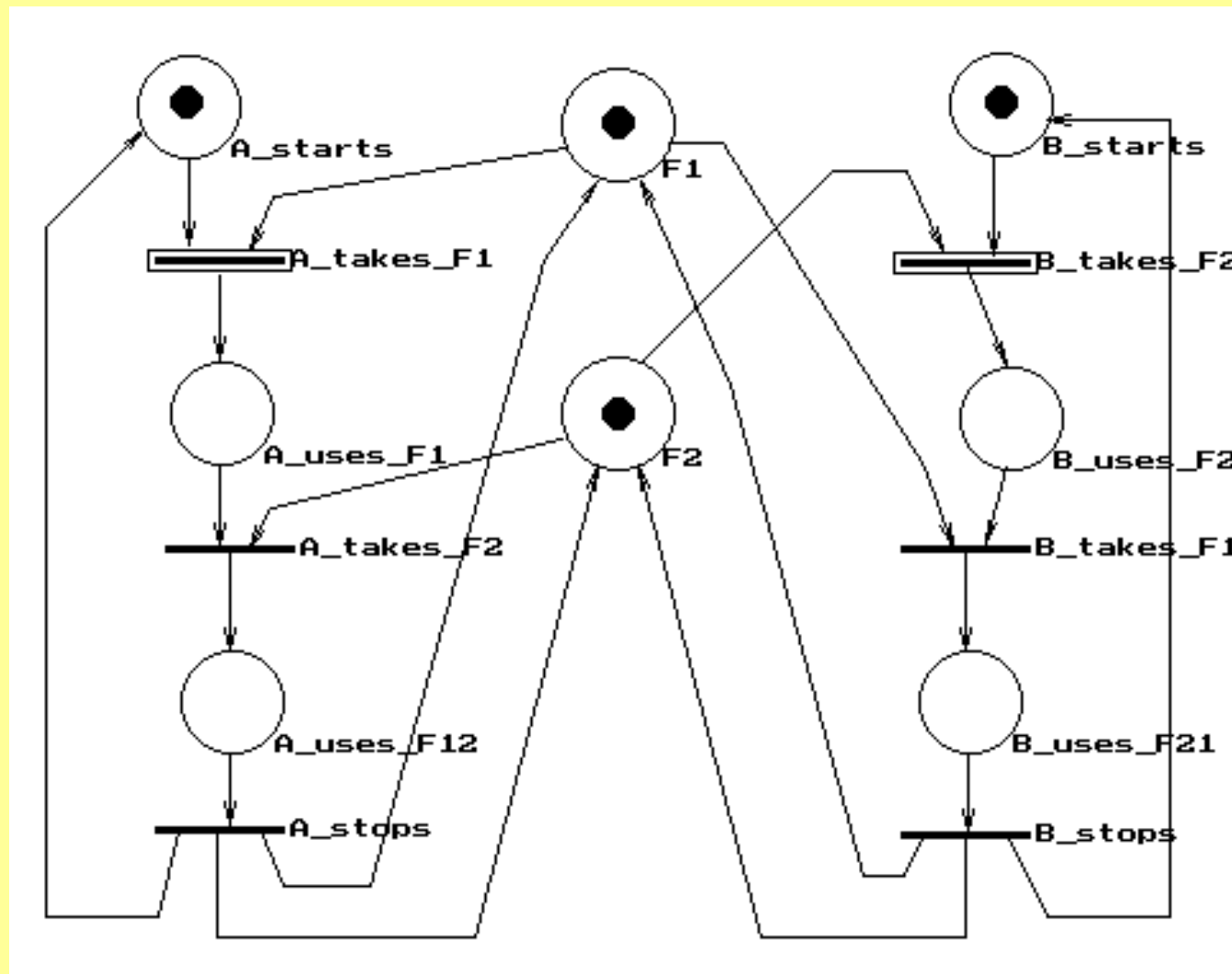
Příklad číslo 6 - deadlock



Vznik deadlocku

- Při synchronizaci může dojít k takzvanému deadlocku, kdy se jedno či více vláken zastaví ve stavu čekání na uvolnění nějakého zdroje
- Jedna z příčin vzniku deadlocku:
 - vlákno V1 získá zdroj Z1
 - vlákno V2 získá zdroj Z2
 - vlákno V1 čeká na uvolnění zdroje Z2
 - současně vlákno V2 čeká na uvolnění zdroje Z1

Deadlock v Petriho síti



Třída reprezentující vlákno

```
class DLVlakno extends Thread {  
    String s1;  
    String s2;  
  
    public DLVlakno(String s1, String s2) {  
        this.s1 = s1;  
        this.s2 = s2;  
    }  
}
```

Metoda run() s potenciálním deadlockem

```
@Override
public void run() {
    for (int i = 0; i < 1000; i++) {
        synchronized(this.s1) {
            synchronized(this.s2) {
                System.out.println(this.s1+this.s2);
            }
        }
    }
}
```

Spuštění dvou vláken a vznik deadlocku

```
public class DeadLock
{
    public static void main(String[] args) {
        String s1 = "Dead";
        String s2 = "Lock";
        new DLVlakno(s1, s2).start();
        new DLVlakno(s2, s1).start();
    }
}
```

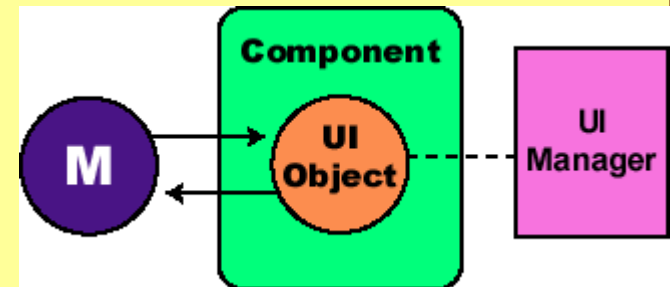
Část 18

Grafické uživatelské rozhraní



Grafické uživatelské rozhraní

- **AWT**
 - v Javě od jejího vzniku
 - využívá nativní komponenty (ovšem jen základní)
- **Swing**
 - založeno na návrhovém vzoru MVC (Model-View-Controller)
 - pluggable look and feel
 - tzv. lightweight komponenty - psané v Javě
- **SWT**
 - není součástí SDK ani JRE
 - taktéž využívá nativní komponenty jako AWT



AWT

- Tři typy objektů a rozhraní
 - Kontejnery
 - Mohou obsahovat další kontejnery či komponenty
 - Komponenty
 - Základní ovládací prvky GUI
 - Správci rozvržení (Layout Managers)
 - Určují rozmístění komponent na kontejneru

Kontejnery

- `java.awt.Container`
 - `add(Component)`
 - `add(Component, index)`
 - `add(Component, constraints)`
 - `setLayout(LayoutManager)`

Komponenty

- `java.awt.Component`
 - `Button`
 - `Canvas`
 - `Checkbox`
 - `Choise`
 - `Label`
 - `List`
 - `Scrollbar`
 - `TextComponent`
 - `TextArea`
 - `TextField`

Správci rozvržení

- `java.awt.LayoutManager`
 - `BorderLayout`
 - `CardLayout`
 - `FlowLayout`
 - `GridLayout`
 - `GridBagLayout`

Reakce na události

- `java.util.EventListener`
- Rozhraní, z něhož dědí další rozhraní
 - `ActionListener`
 - `AdjustmentListeter`
 - `ItemListener`
 - `KeyListener`
 - `MenuListener`
 - `MouseMotionListener`
 - `MouseWheelListener`
 - `PopupMenuListener`
 - `WindowFocusListener`

Anonymní třídy

- Třídy bez explicitně uvedeného názvu
- Není členem své obklopující třídy
- Deklarace a současně inicializace
- Lze je použít v programu všude tam, kde se může zapsat výraz
- Většinou tyto třídy implementují metody předepsané v jejich rozhraní
- Typické použití - implementace listenerů

ActionListener

- actionPerformed()
- Typická komponenta
 - Button

Příklad využití anonymní třídy implementující ActionListener

```
button.addActionListener(new ActionListener()  
{  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        // libovolný kód pracující s tlačítkem  
        // a/nebo obalující třídou  
    }  
});
```


ItemListener

- `itemStateChanged()`
- Typické komponenty
 - `List`
 - `Checkbox`

KeyListener

- keyPressed()
- keyReleased()
- keyTyped()

MouseListener

- `mouseClicked()`
- `mouseEntered()`
- `mouseExited()`
- `mousePressed()`
- `mouseReleased()`

MouseMotionListener

- `mouseDragged()`
- `mouseMoved()`

MouseWheelListener

- `mouseWheelMoved()`

WindowListener

- `windowActivated()`
- `windowClosed()`
- `windowClosing()`
- `windowDeactivated()`
- `windowDeiconified()`
- `windowIconified()`
- `windowOpened()`

WindowFocusListener

- `windowGainedFocus()`
- `windowLostFocus()`

Swing - widgety (1)

- JButton
- JCheckBox
- JComboBox
- JList
- JMenu
- JRadioButton
- JSlider
- JSpinner

Swing - widgety (2)

- JLabel
- JTextField
- JPasswordField
- JTextPane
- EditorPane
- JTextArea

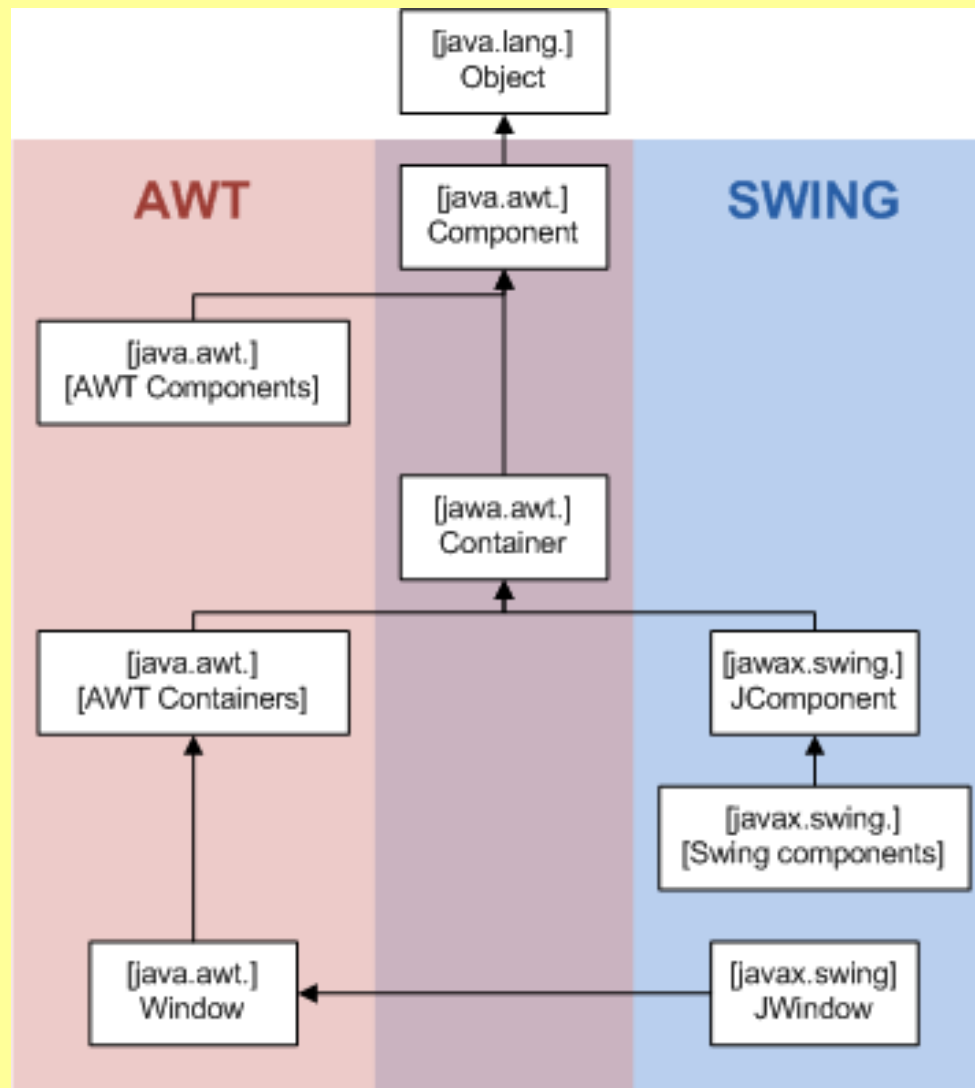
Swing - widgety (3)

- JColorChooser
- JFileChooser
- JTable
- JTree
- JProgressBar

Kontejnery

- JApplet
- JDialog
- JFrame
- JPanel
- JScrollPane
- JSplitPane
- JTabbedPane
- JToolBar
- JInternalFrame
- JLayeredPane

Vzájemné prolínání AWT a Swing



Tvorba GUI aplikací pomocí Swingu

- Demonstrační příklady
 1. Vytvoření jednoduchého prázdného okna
JFrame
 2. Okno a komponenty v něm umístěné
GridLayout manager
 3. Naprogramování reakcí na události
JButton
JTextField
JPasswordField

Příklad číslo 1 - vytvoření prázdného okna

Příklad číslo 1 - vytvoření prázdného okna

```
import javax.swing.*;

public class GuiTest1 extends JFrame
{
    public GuiTest1() {
        // nastavení vlastností okna
        this.setTitle("GuiTest1");
        this.setSize(450, 300);
        this.setLocation(100, 100);
        // důležité - aplikace má
        // při zavření okna skončit
        this.setDefaultCloseOperation(
            WindowConstants.DISPOSE_ON_CLOSE);
    }
}
```

Příklad číslo 1 - vytvoření prázdného okna

```
public static void main(String[] args)
{
    // třída GuiTest1 rozšiřuje třídu JFrame
    JFrame frame = new GuiTest1();
    // zobrazení okna
    frame.setVisible(true);
}
}
```


Příklad číslo 2 - použití layout manageru

Příklad číslo 2 - použití layout manageru

```
import java.awt.*;
import javax.swing.*;

public class GuiTest2 extends JFrame
{
    public GuiTest2() {
        // nastavení vlastností okna
        this.setTitle("GuiTest2");
        this.setSize(450, 200);
        this.setLocation(100, 100);
        // důležité - aplikace má
        // při zavření okna skončit
        this.setDefaultCloseOperation(
            WindowConstants.DISPOSE_ON_CLOSE);
    }
}
```

Příklad číslo 2 - použití layout manageru

```
// nastavení layout manageru
this.setLayout(new GridLayout(3, 2));

// přidání komponent na panel
this.add(new JLabel("Jmeno:"));
this.add(new JTextField());
this.add(new JLabel("Heslo:"));
this.add(new JPasswordField());
this.add(new JPanel());
this.add(new JButton("OK"));
```

Příklad číslo 2 - použití layout manageru

```
public static void main(String[] args)
{
    // třída GuiTest2 rozšiřuje třídu JFrame
    JFrame frame = new GuiTest2();
    // zobrazení okna
    frame.setVisible(true);
}
```

Příklad číslo 3 - reakce na události

Příklad číslo 3 - reakce na události

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

public class GuiTest3 extends JFrame
{
    private JTextField nameComponent =
        new JTextField();

    private JPasswordField passwordComponent =
        new JPasswordField();

    private JButton okComponent =
        new JButton("OK");

    private JButton closeComponent =
        new JButton("Close");
```

Příklad číslo 3 - reakce na události

```
// nastavení vlastnosti rámce
private void setFrame()
{
    this.setTitle("GuiTest3");
    this.setSize(450, 200);
    this.setLocation(100, 100);
    // chování aplikace při zavření rámce
    this.setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
}
```

Příklad číslo 3 - reakce na události

```
// nastavení layout manageru
private void setLayout()
{
    // volba layout manageru
    GridLayout layout = new GridLayout(3, 2);
    // mezery mezi komponentami
    layout.setHgap(20);
    layout.setVgap(20);
    // přiřazení layout manageru k panelu
    this.setLayout(layout);
}
```


Příklad číslo 3 - reakce na události

```
// konstruktor
public GuiTest3() {
    setFrame();
    setLayout();
    this.add(new JLabel("Jmeno:"));
    this.add(this.nameComponent);
    this.add(new JLabel("Heslo:"));
    this.add(this.passwordComponent);
    ...
    // následuje nastavení reakcí na události
```

Příklad číslo 3 - reakce na události

```
this.okComponent.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e) {
        String message = "Jmeno: " +
            nameComponent.getText()+"\n"+
            "Heslo: " +
            passwordComponent.getText();
        JOptionPane.showMessageDialog(
            GuiTest3.this, message, "Bylo zadano",
            JOptionPane.INFORMATION_MESSAGE);
    }
});

this.add(this.okComponent);
```

Příklad číslo 3 - reakce na události

```
this.closeComponent.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        if  
(JOptionPane.showConfirmDialog(GuiTest3.this, "Skutečně  
si přejete skončit?") == JOptionPane.OK_OPTION)  
        {  
            System.exit(0);  
        }  
    }  
});  
  
this.add(this.closeComponent);
```

Příklad číslo 3 - reakce na události

```
// metoda main s inicializací
public static void main(String[] args)
{
    // třída GuiTest3 rozšiřuje třídu JFrame
    JFrame frame = new GuiTest3();
    // zobrazení okna
    frame.setVisible(true);
}
}
```

Část 19

JDBC



JDBC

- Java Database Connectivity
- API pro přístup k relačním databázím
- V některých systémech JDBC-to-ODBC bridge
- Nezávislost programu na použité databázi

Balíčky s JDBC

- `java.sql`
 - Základní funkcionality DB
- `javax.sql`
 - Různé rozšiřující třídy a rozhraní

JDBC: základní třídy (1)

- Driver
 - rozhraní implementované každým databázovým ovladačem
- Connection
 - reprezentuje připojení k sezení vytvořeném v databázovém systému

JDBC: základní třídy (2)

- Statement
 - příkaz poslaný do databáze
- PreparedStatement
 - umožňuje vykonat příkaz vícekrát
 - tím je zaručena vyšší výkonnost
- Callable Statement
 - volání uložené procedury
 - předání parametrů proceduře

JDBC: základní třídy (3)

- **ResultSet**
 - většinou obsahuje výsledek příkazu **SELECT**
 - může a nemusí být povolen update záznamů
 - **ResultSet.next()**
 - základ pro čtení vrácených záznamů
 - **ResultSet.getXXX()**
 - čtení jednoho prvku ze záznamu
 - **ResultSet.updateXXX()**
 - update prvku v záznamu

JDBC: základní třídy (4)

- ResultSetMetadata
 - metadata o výsledku příkazu SELECT (typy sloupců atd.)

JDBC: příklad použití

- Všechny potřebné importy
- Získání driveru k databázi
- Připojení k databázi a vytvoření sezení
- Specifikace příkazu (typicky SELECT)
- Iterace nad výsledkem příkazu SELECT
- Ukončení (uzavření) příkazu
- Uzavření připojení k databázi

JDBC: potřebné importy

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Statement;
```

JDBC: získání driveru k databázi

```
// načtení driveru
Class.forName("oracle.jdbc.OracleDriver");

// databázově závislý řetězec s URL
String url = "jdbc:oracle:thin:@myhost:1521/orcl";

// připojení k databázi
Connection conn = DriverManager.getConnection(url,
"user", "password");
```

JDBC: vytvoření příkazu a získání jeho výsledku

```
String select = "select * from test_table");  
  
try {  
    Statement stmt = conn.createStatement();  
    try {  
        ResultSet resultSet =  
            stmt.executeQuery(select);
```

JDBC: zpracování výsledku SELECTu

```
while (resultSet.next()) {  
    int id = resultSet.getInt(1);  
    String name = resultSet.getString(2);  
    System.out.println(id + ": " + name);  
}
```


JDBC: ResultSet je nutné uzavřít

```
finally {  
    try {  
        resultSet.close();  
    }  
    catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

JDBC: i Statement je nutné uzavřít

```
finally {  
    try {  
        stmt.close();  
    }  
    catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

JDBC: nakonec se musí uzavřít i Connection

```
finally {  
    try {  
        conn.close();  
    }  
    catch (Exception ignore) {  
        e.printStackTrace();  
    }  
}
```

Část 20

Práce s Eclipse

