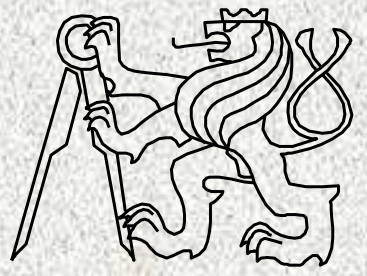
Třídy a objekty



A0B36PR1-Programování 1

Fakulta elektrotechnická
České vysoké učení technické

Třída

- Třída v jazyku Java je programová jednotka tvořená množinou identifikátorů, které mají třídou definovaný význam a které označují prostředky, které lze v téže či jiných třídách využít. Třída je zdrojem metod popisujících řešení problému rozkladem na podproblémy.
- Základem aplikace, tj. uživatelského programu, je třída, ve které
  - je deklarována spouštěcí metoda přesně takto:

    public static void main(String[ ] args) { ... }
  - mohou být deklarovány další pomocné metody
  - mohou být deklarovány statické proměnné, které jsou použitelné jako nelokální proměnné ve metodách dané třídy

Třída jako zdroj funkcionality

- Mnohé třídy nemají deklaraci metody *main,* např.:
  - knihovní třída *java.lang.Math* poskytující matematické funkce
- Metoda `main` je mnohdy využívána pro ukázku použití dané třídy
- V jiných jazycích, např. v C++, lze program vytvořit bez použití tříd

Třída jako datový typ

- Obecně je třída popisem strukturovaného datového typu, tzn. specifikuje
  - množinu hodnot datových objektů skládajících se ze složek, a
  - množinu operací s datovými objekty
- Datové objekty typu třída (zkráceně jen objekty) se nazývají též instancemi třídy
- V jazyku Java lze objekty (instance tříd) vytvářet pouze dynamicky pomocí operátoru *new* a přistupovat k nim pomocí referenčních proměnných

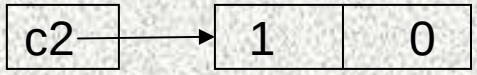
  (Pole jsou také objekty a tudíž i referenční typy)
- Třídy jako datové typy se nejprve naučíme používat pasivně

Konstruktory

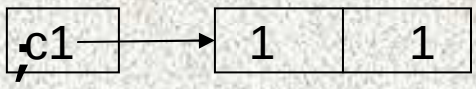
- Třída *Complex* umožňuje vytvořit a inicializovat objekt třemi způsoby:

```
Complex c1 = new Complex();
```


```
Complex c2 = new Complex(1);
```




```
Complex c3 = new Complex(1,1);
```



- Tyto tři možnosti jsou dány třemi konstruktory třídy *Complex*
  - konstruktor bez parametrů *Complex*( ) inicializuje vytvořený objekt hodnotami 0,0
  - konstruktor s jedním parametrem *Complex*(*double re*) inicializuje vytvořený objekt hodnotami *re*,0
  - konstruktor se dvěma parametry *Complex*(*double re*, *double im*) inicializuje vytvořený objekt hodnotami *re* a *im*

Přetěžování konstruktorů

Konstruktor je metoda, která vytvoří objekt a nastaví jeho počáteční hodnoty.

Budeme chtít 3 typy konstruktorů pro Obdelnik:

- bez parametrů - vytvoří obdélník o stranách 0x0,
- jeden parametr - vytvoří čtverec,
- 2 parametry - šířka x výška

```
public class Obdelnik {
        Obdelnik(){
            sirka =  vyska =0;
        }
        Obdelnik(int a){
            sirka =  vyska = a;
        }
        Obdelnik(int s, int v){
            sirka = s;
            vyska = v;
        }
    }
```

Přetěžování konstruktorů II

Budeme chtít konstruktor, který u vytvářeného obdélníku specifikuje jeho barvu.

```
public class Obdelnik {
    …
Obdelnik(int s, int v){
    sirka = s;
    vyska = v;
    }
Obdelnik(int s, int v, Color c){
    sirka = s;
    vyska = v;
    barva = c;
    }
}
```

Stejný kód, správné by bylo použití jednoho místa pro tento kód - jednodušší opravy.

Vzájemné volání konstruktorů

Vytvoříme jedem „univerzální“ konstruktor a ostatní jej budou volat

```
public class Obdelnik {
    …
Obdelnik(int s, int v){

    this(s,v,Color.BLACK);
    }

Obdelnik(int s, int v, Color c){
        sirka = s;
        vyska = v;
       barva = c;
    }
}
```

nastavení implicitní hodnoty

volání konstruktoru téže třídy

Vlastnosti konstruktorů

- Jméno konstruktoru je totožné se jménem třídy
- Konstruktor nemá návratovou hodnotu (ani `void`)
- Předčasně lze ukončit činnost konstruktoru `return` bez parametrů
- Konstruktor má parametrovou část jako metoda - může mít libovolný počet a typ parametrů
- V těle konstruktoru použít operátor **`this`**, který obsahuje odkaz na právě konstruovaný objekt
- Konstruktor je zpravidla vždy `public` (!)
  - //třída java.lang.Math jej má private – proč?
- Příslušný konstruktor je volán automaticky při vytváření objektu operátorem **`new`**

Instanční metody

- Operace s objekty se realizují pomocí instančních metod
- Metody mohou mít parametry a mohou vracet výsledek
- Volání, tj. užití, metody *m* na objekt referencovaný proměnnou *p* má tvar:
  - p.m(*seznam argumentů*)

  Zkráceně říkáme, že „metoda *m* se volá na objekt *p*“
- Příklady metod definovaných třídou *Complex*:
  - double abs() // *c.abs*() je absolutní hodnota komplexního čísla *c*
  - Complex plus(Complex y) // c1.*plus*(*c*2) je reference na nový objekt typu *Complex*, jehož hodnotou je součet komplexních čísel *c*1 a *c*2
  - String toString() //výsledkem volání *c.toString*( ) je řetěz znakové reprezentace komplexního čísla *c* (metodou je zavedena implicitní typová konverze z typu *Complex* na typ *String*)

Statické x instanční metody

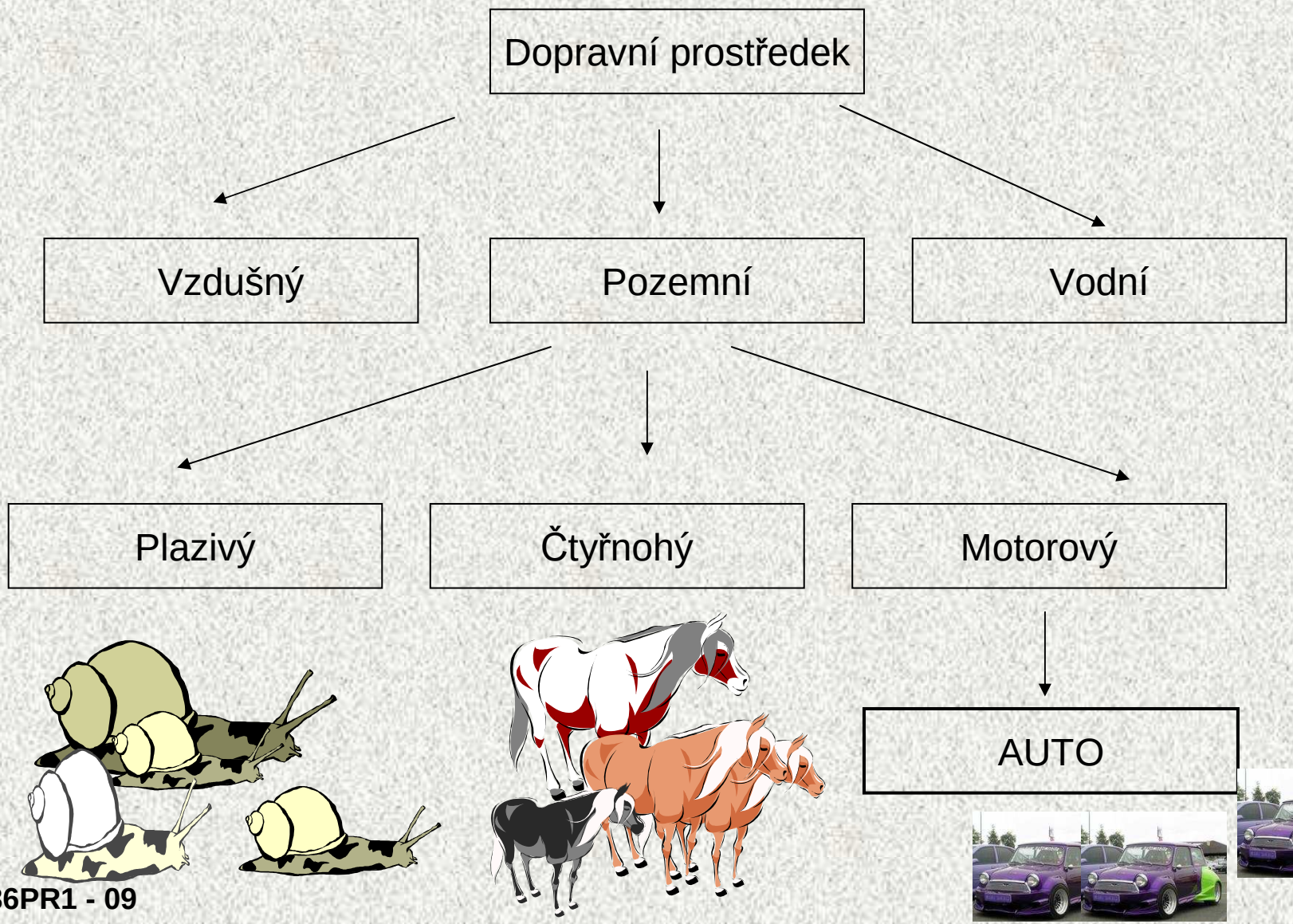
- Pod pojmem metoda se skrývají dva druhy metod:
  - statické metody
  - instanční metody
- všechny mohou mít parametry a mohou vracet výsledek
- Statická metoda je dostupná pomocí jména třídy - aniž je nutno vytvářet někaký objekt. Voláme ji tedy takto:

  `JménoTřídy.jménoMetody( seznam argumentů )`

  (lze využít i volání pomocí referenční prom., ale není doporučeno)
- Instanční metoda označuje operaci nad objektem čili instancí dané třídy. Je dostupná jen přes referenci na objekt. Voláme ji tedy takto:

  `referenčníProměnná.jménoMetody( seznam arg..ů )`

Třídy a dědičnost



Třída Object

Pokud v definici třídy není explicitně uvedena nadtřída, je jako nadtřída automaticky použita třída `Object`, metody z třídy Object jsou tedy zděděny ve všech třídách (definovaná v `java.lang`):

1. **`class X {}`** je ekvivalentní s **`class X extends Object {}`**

některé metody třídy Object:

```
2. public boolean equals(Object o);
        // porovnává objekt s parametrem
3. public String toString();
      // vrací textovou informaci o objektu
4. public int hashCode();
      // vrací jednoznačný hešovací kód objektu
5. protected Object clone();
        // vytváří kopii objektu
```

…

Metody `toString, equals`

```
public String toString(){
   return getClass().getName() + "@" +
                  Integer.toHexString(hashCode());
}
```

– výsledkem volání *x.toString*( ) je řetěz znakové reprezentace objektu *x*,

– metodou je zavedena implicitní typová konverze z typu objektu *x* na řetězec, použitá např. při výpisu objektu

```
public boolean equals(Object obj) {
 return (this == obj);
}
```

- standardní chování neporovnává položky

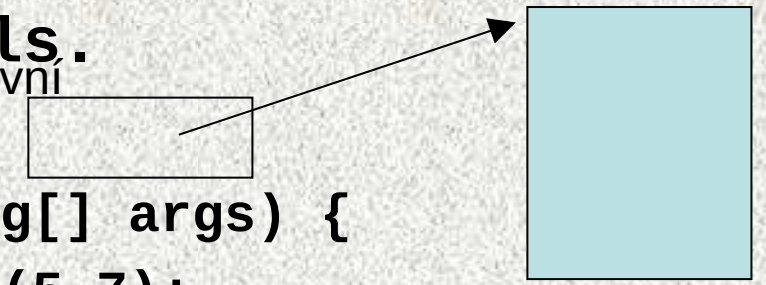
Zastínění metod `equals` a `toString`

Každá třída (kromě jediné) v Javě je potomkem třídy **Object,** která implementuje několik základních metod. Základní jsou metody **`toString`** a **`equals.`**

```
public String toString(){
 return "Obdelnik: "+sirka+" x "+vyska;
}
public boolean equals(Object o){
 if(!(o instanceof Obdelnik))return false;
                          // porovnam jen s obdelniky
 Obdelnik obd = (Obdelnik) o;  //pretypovani
 return (sirka == obd.sirka)&&(obd.vyska==vyska);
}
```

Vlastnosti metody `equals`

Každá třída (kromě jediné) v Javě je potomkem třídy **Object,** která implementuje několik základních metod. Základní jsou metody **`toString`** a **`equals`.**



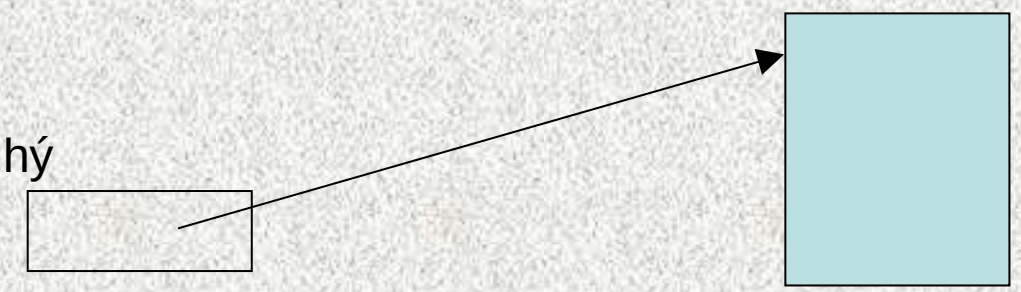

```
public class ObdelnikTest {
    public static void main(String[] args) {
    Obdelnik prvni = new Obdelnik(5,7);

    Obdelnik druhy = new Obdelnik(5,7);
    System.out.println("Prvni: " + prvni);
    System.out.println("Stejne? "+(prvni==druhy));
   System.out.println("Stejne?"+prvni.equals(druhy));
    }
}
```

Prvni: alg7.Obdelnik@11b86e7

false

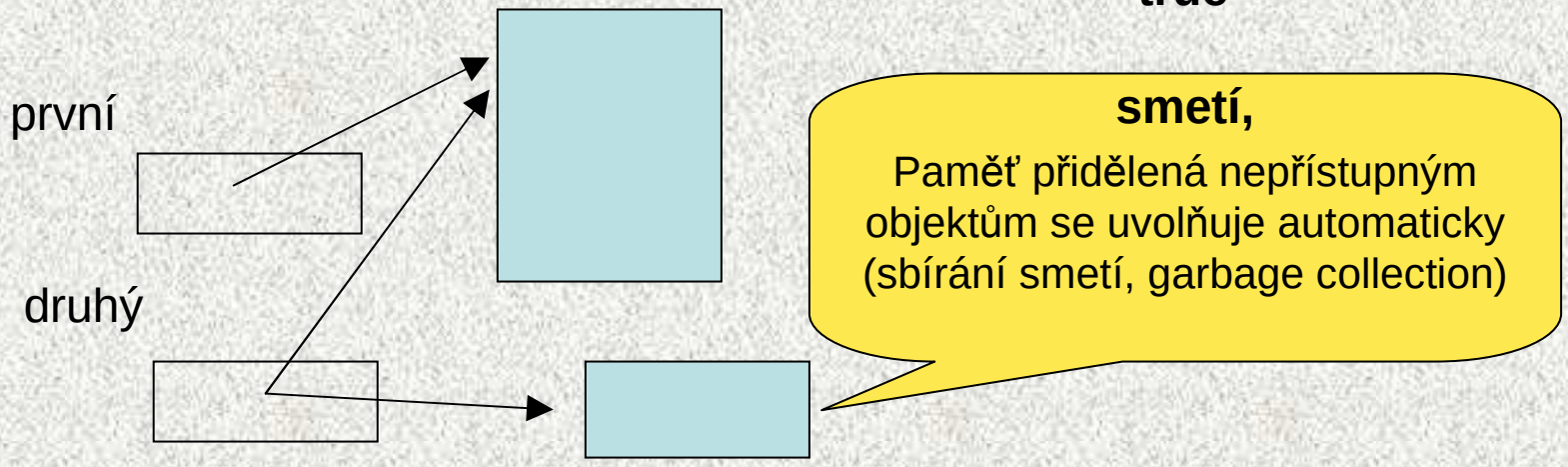
false



Více referencí na jeden objekt, smetí

```
public class ObdelnikTest {
    public static void main(String[] args) {
        Obdelnik prvni = new Obdelnik(5,7);
        Obdelnik druhy = new Obdelnik(5,2);
        druhy = prvni;
        System.out.println("Stejne? "+(prvni==druhy));
        System.out.println("Stejne?"+
                           prvni.equals(druhy));
    }
}
```

```
true
true
```



smetí,

Paměť přidělená nepřístupným objektům se uvolňuje automaticky (sbírání smetí, garbage collection)

Hierarchie tříd, definice

- Třída *Tpod,* která je podtřídou třídy *Tnad,* dědí vlastnosti nadtřídy *Tnad* a rozšiřuje je o nové vlastnosti; některé zděděné vlastnosti mohou být v podtřídě modifikovány

- Pro instanční metody to znamená:
  - každá metoda třídy *Tnad* je i metodou třídy *Tpod,* v podtřídě však může mít jinou implementaci (může být zastíněna - override)
  - v podtřídě mohou být definovány nové metody

- Pro strukturu objektu to znamená:
  - instance třídy *Tpod* mají všechny členy třídy *Tnad* a případně další

Hierarchie tříd, vlastnosti

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

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

Primitivní typy jako objekty

- Do kolekcí lze vkládat pouze reference na objekty, nikoli primitivní typy (které primitivní typy v Javě máme?)
- Primitivní typ např. čísla typu *int*, musíme je nejprve zabalit (wrap) do objektů typu java.lang.*Integer*

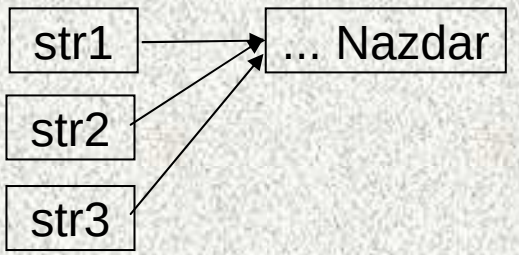
```
ArrayList cisla = new ArrayList();
cisla.add( new Integer(10));
cisla.add( new Integer(20));
System.out.println(cisla.get(0));      // vypíše se 10
System.out.println(cisla.get(1));     //  vypíše se 20
Integer prvni = (Integer)cisla.get(0);
```

- Číslo, které je v objektu typu *Integer* uloženo, získáme metodou *intValue*: int n = prvni.intValue();
- Podobné obalovací třídy (wrapper classes) jsou v Javě definovány pro všechny (celkem 8) primitivní typy.

Třída String

- Objekty knihovní třídy *java.lang.String* jsou řetězy znaků
- Od ostatních tříd se liší třemi specialitami:
  - objekt typu *String* lze vytvořit literálem (posloupnost znaků uzavřená mezi uvozovky, ”Pepa”, “Další řetězec”)
  - hodnotu objektu typu *String* nelze jakkoli změnit
  - operací konkatenace čili zřetězení je nejen realizována metodou concat, ale i přetíženým operátorem +
- Příklady referečních proměnných typu *String:*

```
String str1 = "Nazdar";
String str2 = str1;
String str3 = "Nazdar";
```



//Java si eviduje seznam všech vytvořených řetězců a pokud již stejný existuje, nevytváří jeho kopii

Operace s řetězy

- Spojení řetězců (konkatenace) +
  "abc" + "123" výsledek je "abc123"
- Jestliže jeden operand operátoru + je typu *String* a druhý je jiného typu, pak druhý operand se převede na typ *String* a výsledkem je konkatenace řetězů
  "abc" + 5 výsledek je "abc5"
  "a" + 1 + 2 výsledek je "a12"
  "a" + 1 + (-2) výsledek je "a1-2"
- Porovnávání řetězů
  - relační operátory == a != porovnávají reference, nikoliv obsah řetězů
  - pro porovnání řetězů na rovnost slouží metoda *equals*

```
String s1 = "abcd" ;
String s2 = "ab" ;
String s3 = s2 + "cd" ;
System.out.println(s1==s3); // vypíše false
System.out.println(s1.equals(s3))// vypíše true
```

Operace s řetězy - porovnávání

- Metoda *compareTo*:

```
String s = "abcd";
System.out.println(s1.compareTo("abdc"));//vypíše –1
System.out.println(s1.compareTo( "abcd"));//vypíše 0
System.out.println(“abdc”.compareTo(s1));// vypíše 1
String s = “nazdar”;
int delka = s.length();              // délka je 5
char znak = s.charAt(1);             // znak je ‘a’
String ss = s.substring(2,4);              // ss je “zd”
int z1 = s.indexOf(‘a’);             // z1 je 1
int z2 = s.lastIndexOf(‘a’);               // z2 je 4
int z3 = s.lastIndexOf(‘A’);               // z3 je -1
```

- Hodnotu referenční proměnné typu *String* lze změnit (odkazuje pak na jiný řetěz), vlastní řetěz změnit nelze

Příklad - palindrom

- Napišme program, který přečte jeden řádek a zjistí, zda se po vynechání mezer jedná o palindrom (čte se stejně zpředu jako zezadu, např. “kobyla ma maly bok”)
- funkce s parametrem typu *String* a výsledkem typu *boolean*:

```
static boolean jePalindrom(String str) {
        int i = 0, j = str.length()-1;
        while (i<j) {
            while (str.charAt(i)==' ') i++;
            while (str.charAt(j)==' ') j--;
            if (str.charAt(i)!=str.charAt(j))return
    false;
            i++; j--;
        }
        return true;
    }
```

Příklad - palindrom

```
public class Palindrom {
 public static void main(String[] args) {
 System.out.println( "Zadejte jeden řádek" );
 String radek = (new Scanner(System.in)).nextLine();
 String vysl;
 if (jePalindrom(radek)) vysl = "je" ;
    else vysl = "není" ;
    System.out.println("Na řádku"+vysl+ " palindrom"
  );
     }
 static boolean jePalindrom(String str) {
     ...
     }
}
```

Pole znaků a řetěz

- Příklad: funkce pro převod celého čísla na řetěz tvořený zápisem čísla v hexadecimální soustavě

```
final static String HEXA = "0123456789abcdef";
static String hexaToDecimal(int x) {
if (x==0) return "0" ;
char[ ] znaky = new char[9];
int y;
if (x<0) y= -x; else y = x;
int prvni = 9;
do {
prvni--;
znaky[prvni] = HEXA.charAt(y%16);
y = y / 16;
} while (y>0);
if (x<0) {
prvni--; znaky[prvni] = '-';
  }
return new String(znaky, prvni, 9-prvni);
}
```


Pro zájemce

Hierarchie tříd v dokumentaci

- V on-line dokumentaci jazyka Java týkající se třídy *ArrayList* najdeme následující obrázek:

```
java.lang.Object
    |
    +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
            |
            +--java.util.ArrayList
```

- Tento obrázek vyjadřuje, že:
  - třída *ArrayList* (definovaná v balíku *java.util*) je podtřídou třídy *AbstractList*
  - třída *AbstractList* je podtřídou třídy *AbstractCollection*
  - třída *AbstractCollection* je podtřídou nejvyšší třídy java.lang.*Object*