Dědičnost, polymorfismus, interface, práce se soubory

Dědičnost

- dovoluje vybudování hierarchie tříd, které se postupně z generace na generaci rozšiřují
- používá se v případech, kdy se chceme vyhnout opakování kódu

Dědění je význačný nástroj pro vytváření opakovaně využitelných programových modulů.

Programový modul by měl být zároveň uzavřený a otevřený.

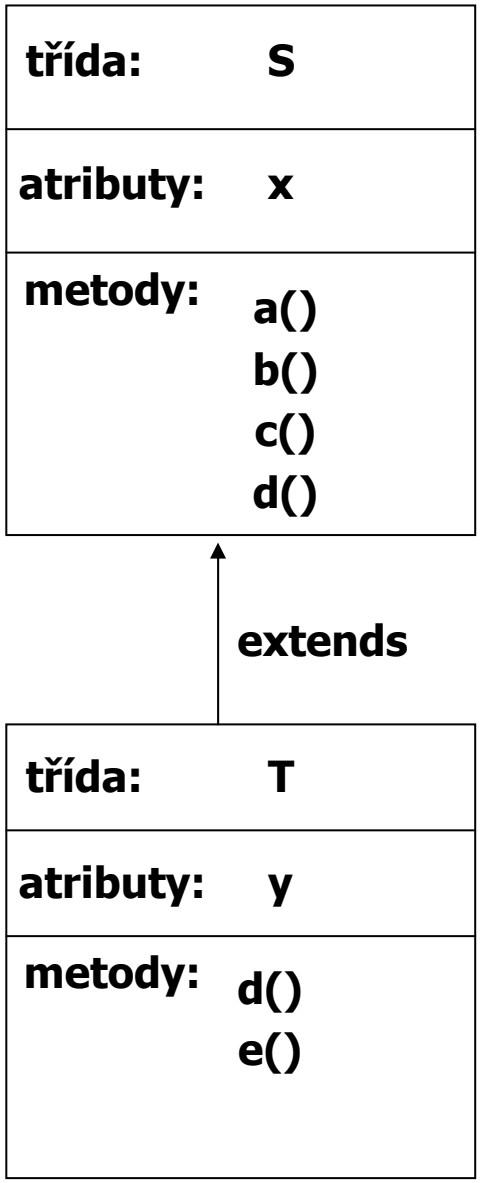
- **uzavřený** – pro jeho použití není potřeba nic přidávat, uživatel není oprávněn modul modifikovat
- **otevřený** – uživatel by měl mít možnost nevhodné věci modifikovat a nové přidávat

Dědění umožňuje v odvozené třídě:

- vše co bylo dobré v základní třídě, ponechat i v odvozené třídě
- vše co nám chybělo, jednoduše dodat
- vše co se nám nelíbilo, změnit

Realizace dědičnosti v Javě:

- musí existovat třída, která se stane rodičem (bázová třída, superclass), jméno této třídy se uvede v hlavičce třídy za klíčovým slovem **extends**
- je povolena pouze jednouchá dědičnost
- vícenásobná dědičnost pouze přes interface



- Objekt třídy **T** obsahuje:
  - atributy **x** a **y**
  - metody a(), b(), c(), d(), e()
  - metoda d(), překrývá metodu d() rodičovské třídy

Změnit vlastnosti metod v odvozené třídě je možné dvojím způsobem:

- **přetížením** (overloading) – použijeme stejné jméno metody, ale jiné parametry, popř. návratový typ
- **překrytím** (overriding, zasínění – hiding) – hlavička metody je identická s hlavičkou metody rodiče, ale metoda může dělat něco jiného

- Při využití dědičnosti pozor na konstruktor(y) potomka !!!! Během konstrukce musí být vždy umožněno volat konstruktor rodiče. Mohou nastat 2 případy:
  - Rodič má konstruktor bez parametrů nebo implicitní - potomek může mít konstruktor implicitní a nemusí být volán konstruktor rodiče
  - Rodič má konstruktor alespoň s jedním parametrem – konstruktor potomka **musí !!!** existovat a prvním příkazem musí být volání konstruktoru rodiče (příkaz super())

```
class Rodic {
  public int i;
  public Rodic(int parI) { i = parI; }
//  public Rodic() { i = 5; }
}

public class Potomek extends Rodic {
  public Potomek() {
    super(8);
  }
  public static void main(String[] args) {
    Potomek pot = new Potomek();
  }
}
```

- Finální metody třídy - používají se tehdy, pokud považujeme metodu za dokonalou a nechceme aby byla ve zděděných třídách překryta.

  Pozor !!! Tato metoda může být přetížena

```
class Rodic {
  public int i;
  public Rodic() { i = 1; }
  final int getI() { return i; }     // koncová metoda třídy  nelze překrýt
}

public class Potomek extends Rodic {
  //  int getI() { return i * 2; }  // chyba

  public static void main(String[] args) {
    Potomek pot = new Potomek();
    System.out.println("Hodnota je: " + pot.getI());
  }
}
```

- Abstraktní třídy – chceme-li aby byla metoda určitě překryta doplníme ji v rodičovské třídě klíčovým slovem *abstract*
  - Jakmile je jako abstract označena jedna z metod třídy musí být použito abstract i u třídy = abstraktní třída.
  - Ve zděděné třídě se musí přeprogramovat všechny metody označené jako abstraktní !!!

```
abstract class Rodic {
  public int i;
  public Rodic() { i = 1; }
  abstract int getI();
  final void setI(int novel) { i = novel; }
}

public class Potomek extends Rodic {
  int getI() { return i * 2; }
  void setI() { i = 5; }   // přetížená

  public static void main(String[] args) {
//    Rodic rod = new Rodic();    // chyba
    Potomek pot = new Potomek();
    pot.setI(3);
    System.out.println("Hodnota je: " + pot.getI());
    pot.setI();   // přetížená
    System.out.println("Hodnota je: " + pot.getI());
  }
}
```

- **Finální třídy – používají se tehdy, nechceme-li, aby byla třída zděděná (např. z důvodů optimalizace třídy během překladu)**
  - **Finální třída nesmí obsahovat abstraktní metody !!!!!**

- **Překrytí proměnné – u jednoduchých datových typů diskutabilní, používá se často u referenčních proměnných (viz. Herout)**

```
public class Progression {

    protected long first;
    protected long cur;

    /** Default constructor.  */
    Progression() {
      cur = first = 0;
    }

    /** Resets the progression to the first value.

    protected long firstValue() {
      cur = first;
      return cur;
    }

    /** Advances the progression to the next value.
     */
    protected long nextValue() {
      return ++cur; // default next value
    }

    /** Prints the first n values of the progression.
    public void printProgression(int n) {
      System.out.print(firstValue());
      for (int i = 2; i <= n; i++)
        System.out.print(" " + nextValue());
      System.out.println(); // ends the line
    }
  }
```

```
class ArithProgression extends Progression {

  /** Increment. */
  protected long inc;

  // Inherits variables first and cur.

  /** Default constructor setting a unit increment. */
  ArithProgression() {
    this(1);
  }

  /** Parametric constructor providing the increment. */
  ArithProgression(long increment) {
    inc = increment;
  }

  /** Advances the progression by adding the increment to the
      current value.
   *
   * @return next value of the progression
   */
  protected long nextValue() {
    cur += inc;
    return cur;
  }

  //  Inherits methods firstValue() and printProgression(int).
}
```

```
/**
*    Geometric Progression
*/


class GeomProgression extends Progression {

       // Inherits variables first and cur.

       /** Default constructor setting base 2. */
       GeomProgression() {
         this(2);
       }

       /** Parametric constructor providing the base.
        *
        * @param base base of the progression.
        */
       GeomProgression(long base) {
         first = base;
         cur = first;
       }

       /** Advances the progression by multiplying the base with
     the current value.
        *
        * @return next value of the progression
        */
       protected long nextValue() {
         cur *= first;
         return cur;
       }

       //  Inherits methods firstValue() and printProgression(int).
     }
}
```

```
/**
 * Fibonacci progression.
 */
class FibonacciProgression extends Progression {
  /** Previous value. */
  long prev;
  // Inherits variables first and cur.

  /** Default constructor setting 0 and 1 as the first two values.
*/
  FibonacciProgression() {
    this(0, 1);
  }
  /** Parametric constructor providing the first and second
values.
   *
   * @param value1 first value.
   * @param value2 second value.
   */
  FibonacciProgression(long value1, long value2) {
      first = value1;
      prev = value2 - value1; // fictitious value preceding the first
  }

  /** Advances the progression by adding the previous value to
the current value.
   *
   * @return next value of the progression
   */
  protected long nextValue() {
    long temp = prev;
    prev = cur;
    cur += temp;
    return cur;
  }
  // Inherits methods firstValue() and printProgression(int).
}
```

Polymorfizmus

- polymorfizmus=vícetvarost, mnohotvarost. Jedná se o možnost využívat v programovém textu stejnou syntaktickou podobu metody s různou vnitřní reprezentací (voláme stejnou metodu a ta pokaždé dělá něco jiného)
- polymorfizmus má smysl tehdy. když má nějaká třída více potomků (více typů) a my k nim přistupujeme jednotným (typově nezávislým) způsobem
- k využití polymorfizmu je výhodné použití abstraktní třídy nebo interface, kdy nadefinujeme abstraktní metody s jasně definovanými parametry a návratovým typem a donutíme programátory, aby je překryli (implementovali). Polymorfismus je ale možné využívat i u neabstraktních tříd

Příklady: využití abstraktní třídy

```
abstract class Zivocich {
  String typ;
  Zivocich(String typ) { this.typ = new String(typ); }

  public void vypisInfo() {
    System.out.print(typ + ", ");
    vypisDelku();
  }

  public abstract void vypisDelku();
}

class Ptak extends Zivocich {
  int delkaKridel;

  Ptak(String typ, int delka) {
    super(typ);
    delkaKridel = delka;
  }

  public void vypisDelku() {
    System.out.println("delka kridel: " + delkaKridel);
  }
}

class Slon extends Zivocich {
  int delkaChobotu;

  Slon(String typ, int delka) {
    super(typ);
    delkaChobotu = delka;
  }

  public void vypisDelku() {
    System.out.println("delka chobotu: " + delkaChobotu);
  }
}
```

```
class Had extends Zivocich {
  int delkaTela;

  Had(String typ, int delka) {
    super(typ);
    delkaTela = delka;
  }

  public void vypisDelku() {
    System.out.println("delka tela: " + delkaTela);
  }
}

public class PolymAbstr {
  public static void main(String[] args) {
    Zivocich[] z = new Zivocich[6];
    for (int i = 0;  i < z.length;  i++) {
      switch ((int) (1.0 + Math.random() * 3.0)) {
        case 1: z[i] = new Ptak("ptak", i); break;
        case 2: z[i] = new Slon("slon", i); break;
        case 3: z[i] = new Had("had", i); break;
      }
    }

    Zivocich t;
    for (int i = 0;  i < z.length;  i++) {
      t = z[i];      // zbytecne, staci z[i].vypisInfo();
      t.vypisInfo();
    }
  }
}
```

- **použití neabstraktních tříd**

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

class Ptak extends Zivocich {
  int delkaKridel;

  Ptak(int delka) { delkaKridel = delka; }

  public void vypisInfo() {
    super.vypisInfo();
    System.out.println("delka kridel: " + delkaKridel);
  }
}

class Slon extends Zivocich {
  int delkaChobotu;

  Slon(int delka) { delkaChobotu = delka; }

  public void vypisInfo() {
    super.vypisInfo();
    System.out.println("delka chobotu: " + delkaChobotu);
  }
}
```

```
class Had extends Zivocich {
  int delkaTela;

  Had(int delka) { delkaTela = delka; }

  public void vypisInfo() {
    super.vypisInfo();
    System.out.println("delka tela: " + delkaTela);
  }
}

public class PolymDeden {
  public static void main(String[] args) {
    Zivocich[] z = new Zivocich[6];
    for (int i = 0;  i < z.length;  i++) {
      switch ((int) (1.0 + Math.random() * 3.0)) {
        case 1: z[i] = new Ptak(i); break;
        case 2: z[i] = new Slon(i); break;
        case 3: z[i] = new Had(i); break;
      }
    }

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

- **použití rozhraní**

```
interface Vazitelny {
  public void vypisHmotnost();
}

class Clovek implements Vazitelny {
  int vaha;
  String profese;

  Clovek(String povolani, int tiha) {
    profese = new String(povolani);
    vaha = tiha;
  }

  public void vypisHmotnost() {
    System.out.println(profese + ": " + vaha);
  }

  public int getHmotnost() { return vaha; }
}

class Kufr implements Vazitelny {
  int vaha;

  Kufr(int tiha) { vaha = tiha; }

  public void vypisHmotnost() {
    System.out.println("kufr: " + vaha);
  }
}
```

```
public class PolymRozhra {
  public static void main(String[] args) {
    int vahaLidi = 0;
    Vazitelny[] kusJakoKus = new Vazitelny[3];

    kusJakoKus[0] = new Clovek("programator", 100);
    kusJakoKus[1] = new Kufr(20);
    kusJakoKus[2] = new Clovek("modelka", 51);

    System.out.println("CD - individualni pristup");
    for (int i = 0;  i < kusJakoKus.length;  i++) {
      kusJakoKus[i].vypisHmotnost();
      if (kusJakoKus[i] instanceof Clovek == true)
//        vahaLidi += kusJakoKus[i].getHmotnost();
        vahaLidi += ((Clovek) kusJakoKus[i]).getHmotnost();
    }
    System.out.println("Ziva vaha: " + vahaLidi);
  }
}
```

Soubory v Javě

- Soubor je častým nástrojem pro komunikaci programu s okolním světem. Slouží k uchování informace na energeticky nezávislém médiu (disk, páska, CD/DVD, flash disk).

- Různé operační systémy využívají různé systémy správy souborů tzv. souborové systémy (file systems = způsob organizace dat na médiích), to se mimo jiné projeví i v různém způsobu v oddělování adresářů a ve specifikaci cesty k souboru např.:
  - Windows: FAT, NTFS – používá více disků, oddělovač adresářů je znak **/**
  - Unix: UFS, Ext2 – používá jediný disk (jeden adresářový strom), používá více disků, oddělovač adresářů je znak ****

- Java je nezávislá na platformě – obsahuje nástroje pro práci se soubory v jednotlivých operačních systémech. Kromě toho java podporuje distribuovaný a vícevláknový výpočet takže informace, kterou program čte nebo zapisuje může ležet principiálně kdekoliv – v souboru na disku, na síti, v paměti. Tato informace může mít podobu znaků, skupiny bytů, objektů apod.

Základní kroky při práci se soubory:

1. **Vytvořit instanci třídy File.** Třída File slouží jako manažer souborů popř. adresářů

konstruktory třídy File :

public File(String filename)

public File(String directory,String filename)

důležité metody:

boolean createNewFile() –vytvoří soubor

boolean delete() – zruší soubor

boolean exist() – true, pokud soubor existuje

boolean isDirectory() – true, pokud je to adresář

boolean isFile() – true, pokud je to soubor

boolean mkdir() – vytvoření adresáře

ong length() –

String getName() – vrací jméno souboru/adresáře

String getParent() – vrací jméno adresáře, ve kterém je soubor umístěn, popř. jméno nadřazeného adresáře

statické proměnné: - zajišťují nezávislost na platformě OS

char File.separatorChar

String FileSeparator

char File.pathSeparatorChar

String File.pathSeparator

```
String aktDir = System.getProperty("user.dir");
File soubAbs = new File(aktDir, "a.txt");
File soubRel = new File("TMP" + File.separator + "a.txt");
File soub = new File("a.txt");
```

2. **Otevřít proud** (stream, kanál), kterým „proudí" informace do/z programu, a nastavit vlastnosti proudu (bufferování, čtení po řádcích, formátované čtení apod.)

Proudy mohou být:

- znakově orientované – základní jednotkou dat je 16 bitový Unicode znak (abstraktní třídy Reader a Writer)
- bajtově orientované – základní jednotka dat je osmibitová (abstraktní třídy InputStream a OutputStream)

Hlavičky metod abstraktních tříd:

– **Třída Reader**

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

– **Třída Writer**

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

– **Třída InputStream**

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

– **Třída *OutputStream***

```
void write(int i)
void write(char[] pole)
void write(char[] pole, int index, int pocet)
```

Od těchto tříd jsou odvozeny:

- **třídy pro fyzický přesun dat –** pro práci se soubory jsou to třídy:
  - FileReader, FileWriter **-** přesun znaků
  - FileInputStream, FileOutputStream **-** přesun bytů
- **třídy vlastností (filtry)**
  - BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream – využití vyrovnávací paměti
  - PrintWriter, PrintStream – formátovaný výstup
  - DataInputStream, DataOutputStream – binární čtení/zápis základních datových typů
  - ObjectInputStream, ObjectOutputStream – binární čtení/zápis libovolných objektů

Příklady práce se soubory:

- **Vstup a výstup znaků**

```
import java.io.*;

public class IoZnaky {
  public static void main(String[] args) throws
     IOException {
    File frJm = new File("a.txt");
    File fwJm = new File("b.txt");

    if (frJm.exists() == true) {
      FileReader fr = new FileReader(frJm);
      FileWriter fw = new FileWriter(fwJm);
      int c;

      while ((c = fr.read()) != -1)
        fw.write(c);

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

- **Vstup a výstup bajtů**

```
import java.io.*;

public class IoBajty {
  public static void main(String[] args) throws
     IOException {
    File frJm = new File("a.txt");
    File fwJm = new File("c.txt");

    if (frJm.exists() == true) {
      FileInputStream fr = new
     FileInputStream(frJm);
      FileOutputStream fw = new
     FileOutputStream(fwJm);

      int c;

      while ((c = fr.read()) != -1)
        fw.write(c);

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

- **čtení po řádcích - využívá vyrovnávací paměť**

```
import java.io.*;

public class PoRadcich {
  public static void main(String[] argv) throws
     IOException {
    FileReader fr = new FileReader("a.txt");
    BufferedReader in = new BufferedReader(fr);
    FileWriter fw = new FileWriter("b.txt");
    BufferedWriter out = new BufferedWriter(fw);
    String radka;

    while((radka = in.readLine()) != null) {
      System.out.println(radka);
      out.write(radka);
      out.newLine();
    }

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

- formátovaný vstup - čte čísla, předpokládá každé číslo na nové řádce

```
import java.io.*;

public class FormatovanyVstup {
  public static void main(String[] args) throws
     IOException {
    FileReader fr = new FileReader("buf.txt");
    BufferedReader in = new BufferedReader(fr);
    String radka;
    int k, suma = 0;

    while((radka = in.readLine()) != null) {
      k = Integer.valueOf(radka).intValue();
      suma += k;
    }

    System.out.println("Soucet je: " + suma);
    fr.close();
  }
}
```

- neformátovaný binární vstup/výstup základních datových typů

```
import java.io.*;

public class BinarniZapis {
  public static void main(String[] args) throws
    IOException {
    FileOutputStream fwJm = new
     FileOutputStream("data.bin");
    DataOutputStream fw = new
     DataOutputStream(fwJm);
    int k, pocet;

    pocet = 2 + (int) (Math.random() * (10 - 2));
    fw.writeInt(pocet);

    for (int i = 0;  i < pocet;  i++) {
      k = (int) (1000.0 * Math.random());
      System.out.print(k + " ");
      fw.writeInt(k);
    }

    fw.writeDouble(Math.PI);
    fw.writeDouble(Math.E);
    System.out.println("\n" + Math.PI + " " +
     Math.E);
    fwJm.close();
```

```
FileInputStream frJm = new
    FileInputStream("data.bin");
   DataInputStream fr = new
    DataInputStream(frJm);

   pocet = fr.readInt();
   for (int i = 0;  i < pocet;  i++) {
     k = fr.readInt();
     System.out.print(k + " ");
   }

   double pi = fr.readDouble();
   double e = fr.readDouble();

   System.out.println("\n" + pi + " " + e);
   frJm.close();
  }
}
```

- formátovaný vstup – textový vstup základních datových typů

  vstup ve tvaru:

  10 AAAAAA BBBBBBB 12.5
  12 CCCCCCC DDDDDDD 23.2

```
import java.io.*;
import java.util.*;
public class soubory {

    public static void main(String[] args) throws IOException {
            FileReader fr = new FileReader("inp.txt");
            BufferedReader in = new BufferedReader(fr);
            String radka,jm="",prjm="",cislo ;
            float x,suma;
            int k=0;
            x=0; suma=0;
            while ((radka=in.readLine()) != null){
                    StringTokenizer st=new StringTokenizer(radka);
                    while (st.hasMoreTokens()){
                            cislo=st.nextToken();
                            k=Integer.valueOf(cislo).intValue();
                            jm=st.nextToken();
                            prjm=st.nextToken();
                            cislo=st.nextToken();
                            x=Float.valueOf(cislo).floatValue();
                            }
                System.out.println(k+" "+jm+" "+prjm+" "+x);
                    suma+=x;

            }
            System.out.println("suma="+suma);
    fr.close();
            }
    }
```

Rozhraní (interface)

- Definuje soubor metod, které v něm nejsou implementovány. V deklaraci rozhraní jsou pouze hlavičky metod (jako u abstraktní třídy), třída, která rozhraní implementuje musí překrýt všechny jeho metody.

- Rozhraní se používá v případech kdy:
  - chceme třídě pomocí implementace rozhraní vnutit konkrétní metody
  - vidíme podobnost v různých třídách, ale pomocí dědění jde tato podobnost jen těžko realizovat (není např.. Možné vytvořit společného předka)
  - požadujeme vícenásobnou dědičnost

konstrukce rozhraní: - podobá se zápisu třídy

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

Použití jednoho rozhraní

- každá třída, která implementuje rozhraní ho musí uvést za klíčové slovo *implements*

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

public class Koule implements Info {
  int polomer;
  Koule(int polomer) { this.polomer = polomer; }
  public void kdoJsem() {
    System.out.println("Koule");
  }
}

public class TestKoule {
  public static void main(String[] args) {
    Usecka u = new Usecka(5);
    Koule  k = new Koule(3);
    u.kdoJsem();
    k.kdoJsem();
  }
}
```

- Použití rozhraní jako typu referenční proměnné

  – pokud je rozhraní implementováno nějakou třídou lze deklarovat referenční proměnnou typu rozhraní, pomocí které lze přistupovat k instancím všech tříd, které toto rozhrani implementují

```
public class TestKoule {
  public static void main(String[] args) {
    Koule k = new Koule(3);
    Info i = new Usecka(5);
    i.kdoJsem();
    i = k;
    i.kdoJsem();
  }
}
```

- **Implementace více rozhraní jedinou třídou**
  - **třída může implementovat více jak jedno rozhraní = vicenásobná dědičnost**

```
public interface InfoDalsi {
  public void vlastnosti();
}
```

```
class Usecka implements Info, InfoDalsi {
  int delka;
  Usecka(int delka) { this.delka = delka; }
  public void kdoJsem() {
    System.out.print("Usecka");
  }
  public void vlastnosti() {
    System.out.println(" = " + delka);
  }
}

public class TestDvou {
  public static void main(String[] args) {
    Usecka u = new Usecka(5);
    u.kdoJsem();
    u.vlastnosti();
  }
}
```

- **Instance rozhraní může využívat jen metody rozhraní**
  - **referenční proměnná typu rozhraní umožňuje přístup jen k metodám rozhraní, ostatní metody třídy nejsou z pohledu rozhraní známy**

```
class Usecka implements Info, InfoDalsi {
  int delka;

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

  public void kdoJsem() {
    System.out.print("Usecka");
  }

  public void vlastnosti() {
    System.out.println(" = " + delka);
  }

  public int getDelka() { return delka; }
}

public class Test {
  public static void main(String[] args) {
    Info info = new Usecka(2);
    info.kdoJsem();
    // info.vlastnosti();                    // chyba
    // System.out.println(info.getDelka());   // chyba
  }
}
```

- **Implementované rozhraní se dědí beze změny**

Zdědíme=li třídu, která implementovala rozhraní, bude metoda z rozhraní přístupná v obou třídách, ale bude se jednat o tutéž metodu – metodu z rodičovské třídy

```
class Usecka implements Info {
  int delka;

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

  public void kdoJsem() {
    System.out.println("Usecka");
  }
}

class Obdelnik extends Usecka {
  int sirka;

  Obdelnik(int delka, int sirka) {
    super(delka);
    this.sirka = sirka;
  }
}

public class Test {
  public static void main(String[] args) {
    Usecka u = new Usecka(5);
    Obdelnik o = new Obdelnik(2, 4);
    Info iu = new Usecka(6);
    Info io = new Obdelnik(3, 6);
    u.kdoJsem();     // vypíše Usecka
    o.kdoJsem();     //       -""-
    iu.kdoJsem();    //
    io.kdoJsem();    //
  }
}
```